SIAM J. COMPUT.
Vol. 26, No. 1, pp. 1–14, February 1997

THE AVERAGE-CASE COMPLEXITY OF DETERMINING THE MAJORITY*

LAURENT ALONSO†, EDWARD M. REINGOLD‡, AND RENÉ SCHOTT§

Abstract. Given a set of n elements each of which is either red or blue, it is known that in the worst case $n - \nu(n)$ pairwise equal/not equal color comparisons are necessary and sufficient to determine the majority color, where $\nu(n)$ is the number of 1-bits in the binary representation of n. We prove that $\frac{2n}{3} - \sqrt{\frac{8n}{9\pi}} + O(\log n)$ such comparisons are necessary and sufficient in the *average case*.

Key words. algorithm analysis, decision trees, lower bounds, average case

AMS subject classifications. 68Q25, 68P10, 68Q20, 68R05, 05A10, 11A63

PII. S0097539794275914

1. Introduction.

Given a set $\{x_1, x_2, \ldots, x_n\}$, each element of which is colored either red or blue, we must determine an element of the majority color by making equal/not equal color comparisons $x_u : x_v$; when n is even, we must report that there is no majority if there are equal numbers of each color. How many such questions are necessary and sufficient?

In the worst case, exactly $n - \nu(n)$ questions are necessary and sufficient, where, following [4], $\nu(n)$ is the number of 1-bits in the binary representation of n. This result was first proved by Saks and Werman [9], who expressed the problem in terms of games and graphs and gave an intricate, technical argument based on generating functions to prove the lower bound. In [2] we gave a short, elementary proof of this result.

The present paper concerns the *average case*. We show in section 3 that any algorithm that correctly determines the majority must on the average use at least

$$\frac{2n}{3} - \sqrt{\frac{8n}{9\pi}} + \Theta(1) \tag{1}$$

color comparisons, assuming all 2^n distinct colorings of the n elements are equally probable. Furthermore, in section 4 we describe an algorithm that uses an average of

$$\frac{2n}{3} - \sqrt{\frac{8n}{9\pi}} + O(\log n) \tag{2}$$

color comparisons. Together these bounds imply that $\frac{2n}{3} - \sqrt{\frac{8n}{9\pi}} + O(\log n)$ such comparisons are necessary and sufficient in the average case to determine the majority. Some open problems are discussed in section 5.

* Received by the editors October 24, 1994; accepted for publication (in revised form) February 2, 1995. This research was supported in part by INRIA and the NSF through grants NSF INT 90-16958 and NFS INT 95-07248.
http://www.siam.org/journals/sicomp/26-1/27591.html

† CRIN, INRIA-Lorraine, Université de Nancy I, 54506 Vandoeuvre-lès-Nancy, France and ENS, 45 Rue d'Ulm, 75005 Paris, France (alonso@loria.crin.fr).

‡ Department of Computer Science, University of Illinois at Urbana–Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801 (reingold@cs.uiuc.edu). The research of this author was supported in part by NSF grants CCR-93-20577 and CCR-95-30297.

§ CRIN, INRIA-Lorraine, Université de Nancy I, 54506 Vandoeuvre-lès-Nancy, France (rene.schott@ loria.fr).

2. The decision tree. The derivation of the average-case lower bound (1) is, as is the derivation of the worst-case lower bound in [2], based on analyzing the underlying binary decision tree (see [8], for example) of any algorithm based on equal/not equal comparisons. Each node of such a decision tree corresponds to a subset of the 2^n possible colorings of the n elements—the largest subset for which the answers to the questions posed are consistent with the coloring. The lower bound for the average case follows from a detailed analysis of the structure of such a tree.

At any node of the decision tree, the state of information obtained about the coloring can be described by a partition of $\{x_1, x_2, \ldots, x_n\}$ into disjoint sets $A_1 \cup B_1 \cup A_2 \cup B_2 \cup \cdots \cup A_m \cup B_m$, for some m, with the meaning that for all i, $1 \le i \le m$, all the elements in A_i are known to have one color and all the elements in B_i are known to have the opposite color. There are clearly 2^m colorings of the elements that are consistent with this partition. At the root of the decision tree we have $m = n$, $A_i = \{x_i\}$, and $B_i = \emptyset$, $1 \le i \le n$, and hence, 2^n consistent colorings. A leaf of the decision tree contains enough information for the algorithm to identify one of the subsets A_i or B_i as being of the majority color.

If we assume that no redundant questions are asked, each internal node of the decision tree reflects a color comparison $x_u : x_v$, $x_u \in A_i \cup B_i$ and $x_v \in A_j \cup B_j$, $i \ne j$. The answer to the question causes the two pairs of sets A_i, B_i and A_j, B_j to be replaced by a single pair of sets, either the pair $A_i \cup A_j$, $B_i \cup B_j$ or the pair $A_i \cup B_j$, $A_j \cup B_i$, depending on the answer to the question. Thus at each internal node the number of colorings splits half and half between the two children of that node, and therefore a node at depth d (the root having depth zero) corresponds to 2^{n-d} colorings.

Observe that a color comparison $x_u : x_v$, $x_u \in A_i \cup B_i$ and $x_v \in A_j \cup B_j$, $i \ne j$, is irrelevant if either $|A_i| = |B_i|$ or $|A_j| = |B_j|$ because, in either case, the cardinalities of the replacement pair of sets for the two pairs of sets A_i, B_i and A_j, B_j do not depend on the outcome of the color comparison. Hence the state of information at a node need not include pairs of sets A_i, B_i for which $|A_i| = |B_i|$, so we will assume that $|A_i| \ne |B_i|$, $1 \le i \le m$.

In considering the state of information at a node in the decision tree described by the partition $A_1 \cup B_1 \cup A_2 \cup B_2 \cup \cdots \cup A_m \cup B_m$, it is convenient to relabel these $2m$ sets so that

$$|A_i| > |B_i|,$$

for $1 \le i \le m$, and

$$|A_i| - |B_i| \ge |A_{i+1}| - |B_{i+1}|,$$

for $1 \le i < m$. Thus by defining, for $1 \le i \le m$,

$$\Delta_i = |A_i| - |B_i|,$$

we can encode all relevant information about the state at a node of the decision tree with the nonincreasing sequence of positive integers

$$\Delta_1 \ge \Delta_2 \ge \cdots \ge \Delta_m > 0.$$

The state of the algorithm at the root of the tree is thus described by a sequence of n ones. At an internal node $(\Delta_1, \Delta_2, \ldots, \Delta_m)$, a color comparison $x_u : x_v$, $x_u \in A_i \cup B_i$

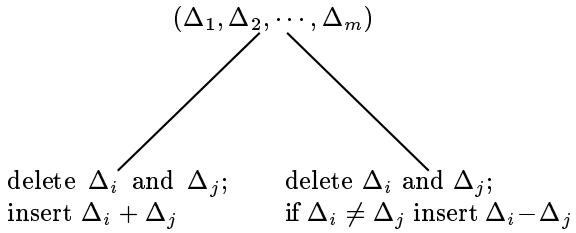


FIG. 1. *The effect on the* Δ*-vector of a color comparison at an internal node of the decision tree with the color comparison* $x_u : x_v$*,* $x_u \in A_i \cup B_i$ *and* $x_v \in A_j \cup B_j$*. If* $\Delta_i = \Delta_j$*, the node is* ordinary*; if* $\Delta_i \neq \Delta_j$*, the node is* unusual.

and $x_v \in A_j \cup B_j$, results in the two values Δ_i and Δ_j being deleted from the sequence and $\Delta_i + \Delta_j$ or $|\Delta_i - \Delta_j| > 0$ then being inserted into place in the sequence, one in the left subtree the other in the right subtree, respectively, depending on the result of the color comparison—in this case the sequence shrinks in length by 1 in passing to subtrees. However, zeroes are never in the sequence (since $\Delta_i = 0$ would mean $|A_i| = |B_i|$), so if $|\Delta_i - \Delta_j| = 0$, the sequence for that subtree shrinks in length by 2. (See Figure 1.) In other words, each internal node of the decision tree can have either both of its children with a Δ-vector one shorter in length than its own Δ-vector or its left child with a Δ-vector one shorter in length than its own and its right child with a Δ-vector two shorter in length than its own; in the former case we call the node *unusual*, and in the latter case we call it *ordinary*.

A leaf in the decision tree corresponds to an outcome of the algorithm, and hence the associated Δ-vector contains enough information to determine the majority. There are two cases: If the Δ-vector is empty ($m = 0$), there is no majority. Otherwise, we must have

$$\Delta_1 > \sum_{i \geq 2} \Delta_i, \tag{3}$$

in which case any element from the set A_1 is a member of the majority. Of course each $\Delta_i \geq 1$, so (3) implies that

$$\Delta_1 \geq m, \tag{4}$$

for every leaf in which there is a majority element; we consider that (4) holds vacuously when $m = 0$ and there is no majority. Thus (4) holds for every leaf in the tree.

3. The lower bound. Given an algorithm for the majority problem, consider its corresponding decision tree T and let $C(T)$ be the average number of color comparisons, assuming all 2^n colorings are equally likely. From the discussion in the previous section, we have

$$C(T) = \sum_{\substack{\text{leaves} \\ l \in T}} \frac{2^{n - \text{depth}(l)}}{2^n} \text{depth}(l)$$

because there are $2^{n-\text{depth}(l)}$ colorings in a leaf at depth depth(l), hence

$$C(T) = \sum_{\substack{\text{leaves} \\ l \in T}} \frac{\text{depth}(l)}{2^{\text{depth}(l)}},$$

and

$$C(T) = \sum_{\substack{\text{internal} \\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}} \tag{5}$$

by induction on the height of the tree.

We will evaluate the sum in (5) by relating it to the average length of the Δ-vector when the algorithm ends (that is, at the leaves, weighted by the number of colorings in a leaf),

$$L(T) = \sum_{\substack{\text{leaves} \\ l \in T}} \frac{\text{length}(l)}{2^{\text{depth}(l)}}, \tag{6}$$

where length(l) is the length of the Δ-vector at leaf l. This, in turn, we will express in terms of the classic ballot problem [3].

Observe that for any integer-valued function f of nodes, we have the identity

$$\sum_{\substack{\text{internal} \\ \text{nodes } v \in T}} [f(v) - f(\text{left}(v)) - f(\text{right}(v))] = f(\text{root}) - \sum_{\substack{\text{leaves} \\ l \in T}} f(l); \tag{7}$$

this follows by induction on the height of the tree (the sum on the left telescopes). Applying (7) to the function

$$f(v) = \frac{\text{length}(v)}{2^{\text{depth}(v)}},$$

we find that $f(\text{root}) = n$,

$$\sum_{\substack{\text{leaves} \\ l \in T}} f(l) = L(T),$$

and

$$f(v) - f(\text{left}(v)) - f(\text{right}(v)) = \begin{cases} \frac{3/2}{2^{\text{depth}(v)}}, & v \text{ ordinary}, \\ \frac{1}{2^{\text{depth}(v)}}, & v \text{ unusual}. \end{cases}$$

So, (7) becomes

$$\sum_{\substack{\text{ordinary internal} \\ \text{nodes } v \in T}} \frac{3/2}{2^{\text{depth}(v)}} + \sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}} = n - L(T).$$

Because every internal node is either ordinary or unusual, we can rewrite this last equation as

$$\frac{3}{2} \sum_{\substack{\text{internal} \\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}} - \frac{1}{2} \sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}} = n - L(T).$$

But by (5), this becomes

$$\frac{3}{2}C(T) - \frac{1}{2}\sum_{\substack{\text{unusual internal}\\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}} = n - L(T),$$

or

$$C(T) = \frac{2n}{3} - \frac{2}{3}L(T) + \frac{1}{3}\sum_{\substack{\text{unusual internal}\\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}} \tag{8}$$

$$\geq \frac{2n}{3} - \frac{2}{3}L(T)$$

$$\geq \frac{2n}{3} - \frac{2}{3}E(\Delta_1), \tag{9}$$

where $E(\Delta_1)$ is the expected value of Δ_1, assuming all 2^n colorings are equally likely, because (4) tells us that $L(T) \leq E(\Delta_1)$ and thus $-L(T) \geq -E(\Delta_1)$.

$E(\Delta_1) = M(n)$, the expected margin of victory in the n-vote ballot problem [3], for consider

$$M(n) = \frac{1}{2^n}\sum_{\substack{\text{colorings}\\ C}} (|\text{majority in } C| - |\text{minority in } C|)$$

$$= \frac{1}{2^n}\sum_{\substack{\text{leaves}\\ l \in T}}\sum_{\substack{\text{colorings}\\ C \in l}} (|\text{majority in } C| - |\text{minority in } C|).$$

But for a leaf l with Δ-vector $(\Delta_1, \Delta_2, \ldots, \Delta_m)$,

$$|\text{majority}| - |\text{minority}| = \Delta_1 \pm \Delta_2 \pm \cdots \pm \Delta_m,$$

where the sign of Δ_i is $+$ if $\text{color}(A_1) = \text{color}(A_i)$ and $-$ otherwise. A given leaf l contains the colorings with all 2^{m-1} sign combinations, so when these are added all terms cancel *except* the Δ_1 term. Thus

$$\sum_{\substack{\text{colorings}\\ C \in l}} (|\text{majority in } C| - |\text{minority in } C|) = 2^{n-\text{depth}(l)}\Delta_1,$$

since there are $2^{n-\text{depth}(l)}$ colorings associated with l, and so

$$M(n) = \frac{1}{2^n}\sum_{\substack{\text{leaves}\\ l \in T}} 2^{n-\text{depth}(l)}\Delta_1$$

$$= \sum_{\substack{\text{leaves}\\ l \in T}} \frac{\Delta_1}{2^{\text{depth}(l)}}$$

$$= E(\Delta_1),$$

as claimed. Inequality (9) now becomes

$$C(T) \geq \frac{2}{3}n - \frac{2}{3}M(n). \tag{10}$$

The value of $M(n)$ was the subject of problem A-4 in the 1974 Putnam competition [7] (see also [1, pp. 22 and 87]). We have,

$$\begin{aligned}
M(n) &= \frac{1}{2^{n-1}} \sum_{k \leq n/2} (n-2k)\binom{n}{k} \\
&= \frac{1}{2^{n-1}} \sum_{k \leq n/2} \left[(n-k)\binom{n}{k} - k\binom{n}{k}\right] \\
&= \frac{1}{2^{n-1}} \sum_{k \leq n/2} \left[n\binom{n-1}{k} - n\binom{n-1}{k-1}\right] \\
&= \frac{n}{2^{n-1}} \sum_{k \leq n/2} \left[\binom{n-1}{k} - \binom{n-1}{k-1}\right] \\
&= \frac{n}{2^{n-1}} \binom{n-1}{\lfloor \frac{n-1}{2} \rfloor} \\
&= \sqrt{\frac{2n}{\pi}} + \Theta(1) \qquad (11)
\end{aligned}$$

by Stirling's formula (see [4, equation (4.15)], for example). Therefore, (10) becomes

$$C(T) \geq \frac{2}{3}n - \sqrt{\frac{8n}{9\pi}} + \Theta(1),$$

as claimed.

4. An algorithm. Equation (8) holds for *any* tree and allows us to analyze the expected behavior of any algorithm by computing $L(T)$ and

$$\sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}}$$

for that algorithm. The algorithm from [2] has no unusual internal nodes and each leaf has a Δ-vector that is the binary representation of an integer between 0 and n, so the expected length must be $O(\log n)$. That algorithm, therefore, has an expected $2n/3 - O(\log n)$ color comparisons—in other words, although exactly optimal in the worst case, is only within an additive $O(\sqrt{n})$ of being optimal in the average case. We can do better, however.

For optimal average-case behavior, we seek to maximize

$$\frac{2}{3}L(T) - \frac{1}{3} \sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}}.$$

We will describe an algorithm (that is, a tree $T(n)$) for which this is $\sqrt{\frac{8n}{9\pi}} - O(\log n)$; this algorithm is thus within an additive $O(\log n)$ of being optimal.

Consider the tree $T(n)$ formed by starting with the Δ-vector $(1, 1, \ldots, 1)$ of n ones at the root and proceeding as follows. Suppose the Δ-vector has the form

$$((2t+1)2^{\delta_1}, 2^{\delta_2}, \ldots, 2^{\delta_m}), \tag{12}$$

$\delta_1 \geq \delta_2 \geq \cdots \geq \delta_m$ and $t \geq 0$; certainly, the initial Δ-vector $(1, 1, \ldots, 1)$ has this form. If $\Delta_1 > \Delta_2 + \cdots + \Delta_m$, it is a leaf. Otherwise, if $\Delta_1 \leq \Delta_2 + \cdots + \Delta_m$, that is, $(2t+1)2^{\delta_1} \leq 2^{\delta_2} + \cdots + 2^{\delta_m}$, the δ_i's cannot all be distinct, so let i be the smallest integer such that for which $\delta_i = \delta_{i+1}$; in other words, let i be the smallest integer such that Δ_i/Δ_{i+1} is odd. If

$$\sum_{j \leq i} \Delta_j > \sum_{j > i} \Delta_j, \tag{13}$$

we make a color comparison between an element from A_1 and one from A_2; otherwise, if

$$\sum_{j \leq i} \Delta_j \leq \sum_{j > i} \Delta_j,$$

we make a color comparison between an element from A_i and one from A_{i+1}. The resulting Δ-vector has the same form as (12) and it follows by induction on the depth in $T(n)$ that each Δ-vector thus obtained satisfies the invariant relationship

$$\text{sum of the non-ones} \leq 2 + \text{number of ones}. \tag{14}$$

Inequality (14) is a critical property of $T(n)$, as we shall see. Figure 2 shows the tree $T(8)$.

The only possible Δ-vectors in the leaves of $T(n)$ are

$$(\ ), \qquad (2), \qquad (i, \underbrace{1, 1, \ldots, 1}_{j}), \quad j = i-1 \text{ or } j = i-2. \tag{15}$$

For suppose, by way of contradiction, that a leaf l has $\Delta_2 \geq 2$. By (14),

$$\Delta_1 + \Delta_2 \leq \text{sum of the non-ones} \leq 2 + \text{number of ones},$$

or

$$\Delta_1 \leq (2 - \Delta_2) + \text{number of ones}.$$

But $2 - \Delta_2 \leq 0$, so

$$\Delta_1 \leq \text{number of ones}.$$

Hence the algorithm cannot correctly conclude at leaf l that A_1 is of the majority color, a contradiction.

Because each leaf has one of the forms in (15), the length of the Δ-vector in a leaf satisfies

$$\text{length(leaf)} = \Delta_1 + O(1). \tag{16}$$

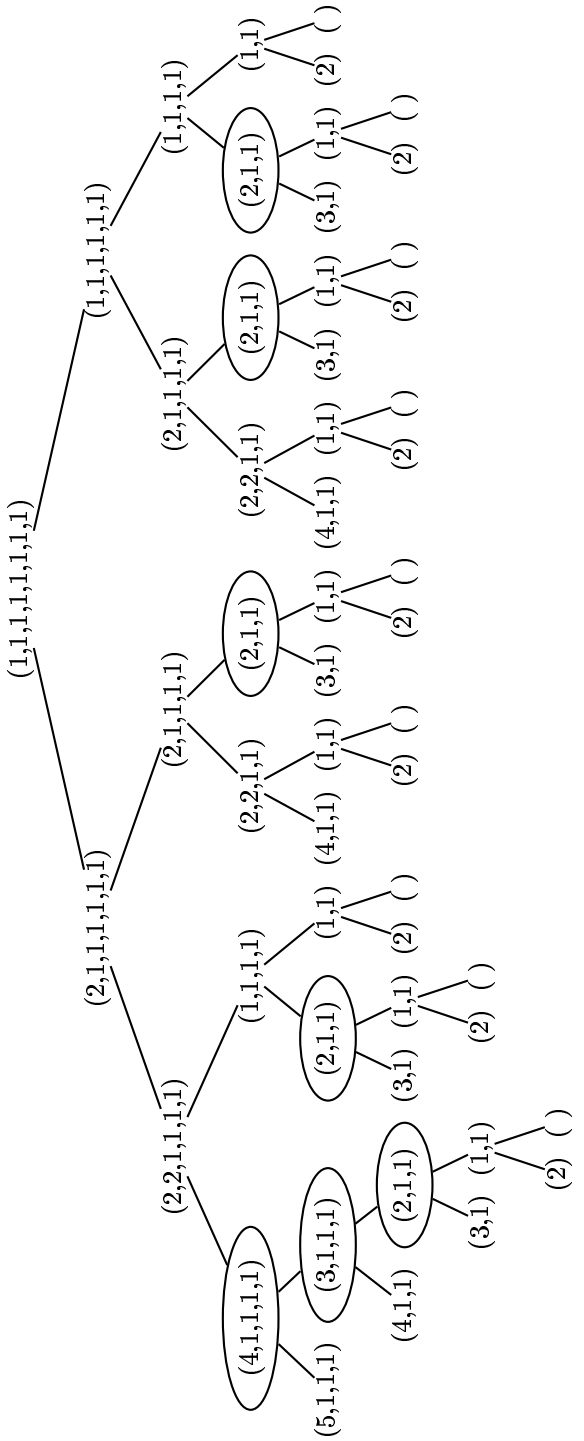


FIG. 2. *The tree $T(8)$. Unusual nodes are enclosed in ovals.*

So,

$$\begin{aligned} L(T(n)) &= E(\text{length(leaf)}) \\ &= E(\Delta_1) + O(1) \\ &= M(n) + O(1) \\ &= \sqrt{\frac{2n}{\pi}} + \Theta(1) \end{aligned}$$

by (11).

It remains to compute

$$\sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T(n)}} \frac{1}{2^{\text{depth}(v)}}.$$

Since $\sum_{j\geq 1} \Delta_j \leq n$, (12) tells us that the only possible values for Δ_2 are 2^i, $0 \leq i \leq \lfloor \lg n \rfloor$, so

$$\sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T(n)}} \frac{1}{2^{\text{depth}(v)}} = \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T(n) \\ \text{with } \Delta_2 = 2^i}} \frac{1}{2^{\text{depth}(v)}}.$$

From inequality (20) in Corollary 4.5 below, we see that the inner sum here is $O(1)$. This implies that the outer sum is $O(\log n)$ and hence that

$$C(T(n)) = \frac{2n}{3} - \sqrt{\frac{8n}{9\pi}} + O(\log n),$$

whence we can conclude that $C(T(n))$ is within an additive $O(\log n)$ of being optimal in the average case. To reach the needed corollary we must study the arrangement of unusual internal nodes in $T(n)$.

PROPOSITION 4.1. *Let v with Δ-vector $(d_1, d_2, \ldots)$ be an ancestor of $\hat{v}$ with Δ-vector $(\hat{d}_1, \hat{d}_2, \ldots)$ in $T(n)$. Then*

$$\text{sum}(\hat{v}) \leq \text{sum}(v) - 2(d_1 - \hat{d}_1), \tag{17}$$

where $\text{sum}(x)$ *is the sum of all values in the Δ-vector of a node x.*

Proof. The proof is by induction on $\text{depth}(\hat{v}) - \text{depth}(v)$ because $\text{sum}(x) - 2\Delta_1(x)$, an integer-valued function of a node x, is nonincreasing as we go from parent to child in $T(n)$. □

PROPOSITION 4.2. *For any k, the highest unusual node in $T(n)$ with $\Delta_2 \leq k$ has Δ_1/Δ_2 even.*

Proof. It suffices to prove that any ancestor of such a node has a Δ-vector $(d_1, d_2, \ldots)$ in which d_1 is either a power of two or is a multiple of $2\Delta_2$. This is proved by induction on the depth of a node. □

PROPOSITION 4.3. *If $v \in T(n)$ is a unusual node with $\Delta_2 = k$, then* (a) *there is no unusual node with $\Delta_2 > k$ in the subtree rooted at v, and* (b) *there is no unusual node with $\Delta_2 = k$ in the subtree rooted at* $\text{left}(v)$.

Proof. Suppose there is such a node v. Let x be the highest ancestor of v that is unusual and has $\Delta_2 \leq k$ (of course, we could have $x = v$), and let $(d_1, d_2, \ldots)$ be the Δ-vector of x. Because x is the highest unusual node with $\Delta_2 \leq k$, we must have d_1/d_2 even—if it were odd, it would contradict Proposition 4.2—and hence if i is the smallest integer such that d_i/d_{i+1} is odd, then $i \geq 2$, and by (13),

$$\sum_{j \leq i} d_j > \sum_{j > i} d_j.$$

Furthermore, because $d_2/d_3, \ldots, d_{i-1}/d_i$ are all even, $d_2 > d_3 > \cdots > d_i$; since these are all powers of two, we have

$$d_2 \geq 1 + \sum_{3 \leq j \leq i} d_j$$

and hence

$$d_1 + 2d_2 \geq 1 + \sum_{j \leq i} d_j.$$

The inequalities above combine to tell us that

$$\begin{aligned} \sum_{j \geq 1} d_j &= \sum_{j \leq i} d_j + \sum_{j > i} d_j \\ &< 2 \sum_{j \leq i} d_j \\ &\leq 2d_1 + 4d_2 - 2. \end{aligned}$$

Thus

$$\operatorname{sum}(x) < 2d_1 + 4d_2 - 2. \tag{18}$$

For part (a), let y be an unusual descendant of v with $\Delta_2 > k$ and let $(\hat{d}_1, \hat{d}_2, \ldots)$ be the Δ-vector of y, $\hat{d}_2 > k \geq d_2$; we have $\hat{d}_1 > \hat{d}_2$ since y is unusual and, since Δ_2 must be a power of two, we have $\hat{d}_2 \geq 2d_2$. (See Figure 3(a).) Because y is also a descendant of x, we have by Proposition 4.1 that

$$\operatorname{sum}(y) \leq \operatorname{sum}(x) - 2(d_1 - \hat{d}_1).$$

Therefore, by (18),

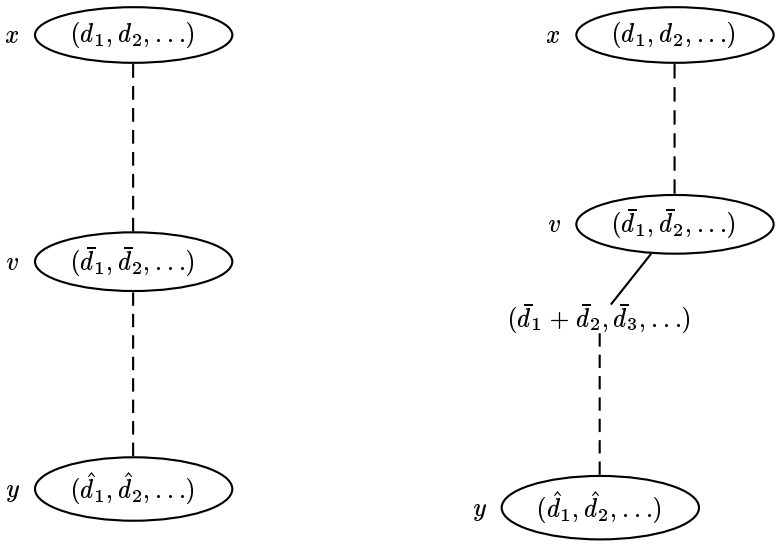
$$\begin{aligned} \operatorname{sum}(y) &< 2d_1 + 4d_2 - 2 - 2(d_1 - \hat{d}_1) \\ &= 2\hat{d}_1 + 4d_2 - 2 \\ &\leq 2\hat{d}_1 + 2\hat{d}_2 - 2. \end{aligned}$$

But

$$\text{number of ones in } y \leq \operatorname{sum}(y) - \hat{d}_1 - \hat{d}_2,$$

since $\hat{d}_1 > \hat{d}_2 \geq 2d_2 \geq 2$, and so

$$\text{number of ones in } y < \hat{d}_1 + \hat{d}_2 - 2,$$



(a) $\hat{d}_1 > \hat{d}_2 \geq 2d_2$ (b) $\hat{d}_1 > \hat{d}_2 = \bar{d}_2 \geq d_2$

FIG. 3. *The structure of the tree $T(n)$ in the two parts of Proposition* 4.3. *Unusual nodes are enclosed in ovals.*

contradicting (14).

For part (b), let y be an unusual descendant of left(v) with $\Delta_2 = k$. Let $(\bar{d}_1, \bar{d}_2, \ldots)$ be the Δ-vector of v and let $(\hat{d}_1, \hat{d}_2, \ldots)$ be the Δ-vector of y; we have $\hat{d}_1 > \hat{d}_2$ since y is unusual and $\hat{d}_2 = \bar{d}_2 \geq d_2$ by hypothesis. (See Figure 3(b).) Because y is a descendant of left(v), we have by Proposition 4.1 that

$$\begin{aligned} \text{sum}(y) &\leq \text{sum}(\text{left}(v)) - 2(\bar{d}_1 + \bar{d}_2 - \hat{d}_1) \\ &\leq \text{sum}(v) - 2(\bar{d}_1 + d_2 - \hat{d}_1) \end{aligned}$$

because $\text{sum}(v) = \text{sum}(\text{left}(v))$ and $\bar{d}_2 \geq d_2$. However, v is a descendant of x, so we have

$$\text{sum}(v) \leq \text{sum}(x) - 2(d_1 - \bar{d}_1)$$

and hence

$$\begin{aligned} \text{sum}(y) &\leq \text{sum}(x) - 2(d_1 - \bar{d}_1) - 2(\bar{d}_1 + d_2 - \hat{d}_1) \\ &= \text{sum}(x) - 2(d_1 + d_2 - \hat{d}_1). \end{aligned}$$

Therefore, by (18),

$$\begin{aligned} \text{sum}(y) &< 2d_1 + 4d_2 - 2 - 2(d_1 + d_2 - \hat{d}_1) \\ &= 2\hat{d}_1 + 2d_2 - 2 \\ &\leq 2\hat{d}_1 + 2\hat{d}_2 - 2, \end{aligned}$$

and we then reach the same contradiction to (14). □

PROPOSITION 4.4. *For any node $x \in T(n)$ and for any constant k,*

$$\sum_{\substack{\text{unusual internal nodes}\\ v \text{ with } \Delta_2 = k \text{ in the}\\ \text{subtree rooted at } x}} \frac{1}{2^{\text{depth}(v)}} \leq \frac{1}{2^{\text{depth}(x)-1}} \tag{19}$$

Proof. The proof is by induction on the height of the subtree rooted at x. If the x is a leaf, the result is immediate. Suppose the proposition holds for subtrees of height at most $h \geq 0$ and consider a subtree of height $h+1$ rooted at x. There are two cases. When the node x is an unusual node with $\Delta_2 = k$, by Proposition 4.3 we know that any unusual node in the left subtree of x has $\Delta_2 < k$ and any unusual node in the right subtree of x has $\Delta_2 \leq k$. Hence

$$\begin{aligned}
\sum_{\substack{\text{unusual internal nodes}\\ v \text{ with } \Delta_2 = k \text{ in the}\\ \text{subtree rooted at } x}} \frac{1}{2^{\text{depth}(v)}} &= \frac{1}{2^{\text{depth}(x)}} + \sum_{\substack{\text{unusual internal nodes}\\ v \text{ with } \Delta_2 = k \text{ in the}\\ \text{subtree rooted at left}(x)}} \frac{1}{2^{\text{depth}(v)}} \\
&\qquad + \sum_{\substack{\text{unusual internal nodes}\\ v \text{ with } \Delta_2 = k \text{ in the}\\ \text{subtree rooted at right}(x)}} \frac{1}{2^{\text{depth}(v)}} \\
&\leq \frac{1}{2^{\text{depth}(x)}} + 0 + \frac{1}{2^{\text{depth}(\text{right}(x))-1}} \\
&= \frac{1}{2^{\text{depth}(x)}} + \frac{1}{2^{\text{depth}(x)}} \\
&= \frac{1}{2^{\text{depth}(x)-1}},
\end{aligned}$$

as desired. Otherwise, direct application of the induction hypothesis to the two subtrees gives

$$\begin{aligned}
\sum_{\substack{\text{unusual internal nodes}\\ v \text{ with } \Delta_2 = k \text{ in the}\\ \text{subtree rooted at } x}} \frac{1}{2^{\text{depth}(v)}} &= \sum_{\substack{\text{unusual internal nodes}\\ v \text{ with } \Delta_2 = k \text{ in the}\\ \text{subtree rooted at left}(x)}} \frac{1}{2^{\text{depth}(v)}} \\
&\qquad + \sum_{\substack{\text{unusual internal nodes}\\ v \text{ with } \Delta_2 = k \text{ in the}\\ \text{subtree rooted at right}(x)}} \frac{1}{2^{\text{depth}(v)}} \\
&\leq \frac{1}{2^{\text{depth}(\text{left}(x))-1}} + \frac{1}{2^{\text{depth}(\text{right}(x))-1}} \\
&= \frac{1}{2^{\text{depth}(x)}} + \frac{1}{2^{\text{depth}(x)}} \\
&= \frac{1}{2^{\text{depth}(x)-1}}. \qquad \square
\end{aligned}$$

COROLLARY 4.5. *For any constant k,*

$$\sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T(n) \\ \text{with } \Delta_2 = k}} \frac{1}{2^{\text{depth}(v)}} \le 2. \tag{20}$$

Proof. Apply Proposition 4.4 to the root of the tree, which has depth zero. □

Corollary 4.5 can be interpreted probabilistically: The sum (20), which resembles Kraft's inequality, is the expected number of unusual internal nodes with $\Delta_2 = k$ encountered as we follow a random path down the tree from the root to a leaf. Proposition 4.4 says that if we encounter such a node, then half the time—whenever we go left—there will be no further such nodes as we continue, so the expected number of such nodes is at most $1 + 1/2 + 1/4 + \cdots = 2$.

5. Open problems. We conjecture that for our tree $T(n)$,

$$\sum_{\substack{\text{unusual internal} \\ \text{nodes } v \in T}} \frac{1}{2^{\text{depth}(v)}} = \Theta(\log n)$$

and that the algorithm departs by an additive logarithmic term from optimality as n increases. Moreover, the worst-case behavior of this algorithm is worse than the best possible $n - \nu(n)$: for $n = 12$, it can take 11 color comparisons. Two open questions are thus suggested:

- What are the exact average-case and worst-case analyses of our algorithm?
- Is there an algorithm that is optimal in *both* the average and worst cases?

Consider the tree formed by starting with the Δ-vector $(1, 1, \ldots, 1)$ at the root and proceeding as follows. Suppose the Δ-vector has the form (12). Let i be the smallest integer such that Δ_i/Δ_{i+1} is odd; if

$$\Delta_1 + \Delta_2 > \sum_{j \ge 3} \Delta_j,$$

make a color comparison between an element from A_1 and one from A_2; otherwise, make a color comparison between an element of A_i and one of A_{i+1}. We conjecture that this algorithm is optimal in the worst case and departs by at most an additive logarithmic term from optimality in the average case, but we are unable to analyze either case. Explicit computation of the trees verifies the optimality of this algorithm for $n \le 52$ in the worst case. In the average case, a similar computation verifies that this algorithm is optimal for $n \le 20$ but not for $n = 50$.

The majority problem is closely related to problems in system diagnosis [6], [10]. It is also similar to coin-weighing problems [5] and knight/knave problems [11]. These relationships deserve exploration.

Acknowledgments. We are indebted to Donald L. Burkholder for suggesting the method we originally used to calculate $M(n)$ in section 3. We are grateful to an anonymous referee who pointed out references [1] and [7] and who suggested several stylistic improvements.

REFERENCES

[1] G. L. Alexanderson, L. F. Klosinski, and L. C. Larson, *The William Lowell Putnam Mathematical Competition, Problems and Solutions:* 1965–1984, Mathematical Association of America, Washington, DC, 1985.

[2] L. Alonso, E. M. Reingold, and R. Schott, *Determining the majority*, Inform. Process. Lett., 47 (1993), pp. 253–255.

[3] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. I, 3rd ed., John Wiley, New York, 1968.

[4] D. H. Greene and D. E. Knuth, *Mathematics for the Analysis of Algorithms*, 3rd ed., Birkhäuser, Boston, 1990.

[5] R. K. Guy and R. J. Nowakowski, *Coin-weighing problems*, Amer. Math. Monthly, 102 (1995), pp. 164–167.

[6] S. L. Hakimi and E. F. Schmeichel, *An adaptive algorithm for system level diagnosis*, J. Algorithms, 5 (1984), pp. 526–530.

[7] A. P. Hillman, *The William Lowell Putnam mathematical competition*, Amer. Math. Monthly, 82 (1975), pp. 905–912.

[8] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms*: *Theory and Practice*, Prentice–Hall, Englewood Cliffs, NJ, 1977.

[9] M. E. Saks and M. Werman, *On computing majority by comparisons*, Combinatorica, 11 (1991), pp. 383–387.

[10] E. Schmeichel, S. L. Hakimi, M. Otsuka, and G. Sullivan, *A parallel fault identification algorithm*, J. Algorithms, 11 (1990), pp. 231–241.

[11] R. Smullyan, *What Is the Name of This Book*?: *The Riddle of Dracula and Other Logical Puzzles*, Prentice–Hall, Englewood Cliffs, NJ, 1978.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 15–38, February 1997

002

AMPLIFICATION BY READ-ONCE FORMULAS*

MOSHE DUBINER† AND URI ZWICK‡

Abstract. Moore and Shannon have shown that relays with arbitrarily high reliability can be built from relays with arbitrarily poor reliability. Valiant used similar methods to construct monotone read-once formulas of size $O(n^{\alpha+2})$ (where $\alpha = \log_{\sqrt{5}-1} 2 \simeq 3.27$) that amplify $(\psi - \frac{1}{n}, \psi + \frac{1}{n})$ (where $\psi = (\sqrt{5}-1)/2 \simeq 0.62$) to $(2^{-n}, 1-2^{-n})$ and deduced as a consequence the existence of monotone formulas of the same size that compute the majority of n bits. Boppana has shown that any monotone read-once formula that amplifies $(p - \frac{1}{n}, p + \frac{1}{n})$ to $(\frac{1}{4}, \frac{3}{4})$ (where $0 < p < 1$ is constant) has size $\Omega(n^{\alpha})$ and that any monotone, not necessarily read-once, contact network (and in particular any monotone formula) that amplifies $(\frac{1}{4}, \frac{3}{4})$ to $(2^{-n}, 1-2^{-n})$ has size $\Omega(n^2)$.

We extend Boppana's results in two ways. We first show that his two lower bounds hold for general read-once formulas, not necessarily monotone, that may even include exclusive-or gates. We are then able to join his two lower bounds together and show that any read-once, not necessarily monotone, formula that amplifies $(p - \frac{1}{n}, p + \frac{1}{n})$ to $(2^{-n}, 1-2^{-n})$ has size $\Omega(n^{\alpha+2})$. This result does not follow from Boppana's arguments, and it shows that the amount of amplification achieved by Valiant is the maximal achievable using read-once formulas.

In a companion paper we construct monotone read-once contact networks of size $O(n^{2.99})$ that amplify $(\frac{1}{2} - \frac{1}{n}, \frac{1}{2} + \frac{1}{n})$ to $(\frac{1}{4}, \frac{3}{4})$. This shows that Boppana's lower bound for the first amplification stage does not apply to contact networks, even if they are required to be both monotone and read-once.

Key words. circuit complexity, Boolean formula, amplification

AMS subject classifications. Primary, 94C10; Secondary, 06E30, 68Q05

PII. S009753979223633X

1. Introduction.

In a classical paper, Moore and Shannon [14] use what is now called the *amplification method* to show that relays with arbitrarily high reliability can be built from (so-called *crummy*) relays with arbitrarily poor reliability.

The amplification method was next used by several researchers to show that particular Boolean functions have small Boolean circuits and formulas. Bennett and Gill [3], extending a result of Adleman [1], used it to show that every language in the complexity class BPP has polynomial-size circuits. Ajtai and Ben-Or [2] used the amplification method to show that probabilistic constant-depth circuits can be simulated by deterministic constant-depth circuits with only a polynomial increase in size.

The main focus of attention in this paper is the elegant application of the amplification method by Valiant [19] to the construction of monotone formulas of size $O(n^{\alpha+2})$ (where $\alpha = \log_{\sqrt{5}-1} 2 \simeq 3.27$) for the majority function and its extension by Boppana [4] to the construction of $O(k^{\alpha+1} n \log n)$-size monotone formulas for the kth threshold function of n variables.

* Received by the editors August 19, 1992; accepted for publication (in revised form) March 31, 1995. A preliminary version of this paper appeared in *Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 258–267 [5].

http://www.siam.org/journals/sicomp/26-2/23633.html

† School of Mathematical Sciences, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel.

‡ School of Mathematical Sciences, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel (zwick@math.tau.ac.il). The research of this author was supported in part by the Basic Research Foundation administrated by the Israel Academy of Sciences and Humanities.

To show the existence of $O(n^{\alpha+2})$-size monotone formulas for the majority function, Valiant first constructs monotone read-once formulas that amplify $(\psi-\frac{1}{n},\psi+\frac{1}{n})$ (where $\psi=(\sqrt{5}-1)/2$) to $(2^{-n},1-2^{-n})$. Formal definitions of all of these terms will appear in the next section. The existence of monotone formulas of the same size for majority follows from a simple probabilistic argument.

Boppana [4] considered the question of whether Valiant had obtained an optimal amount of amplification in his construction and came very close to answering it positively. He observed that Valiant had actually constructed $O(n^{\alpha})$-size monotone read-once formulas that amplify $(\psi-\frac{1}{n},\psi+\frac{1}{n})$ to $(\frac{1}{4},\frac{3}{4})$ and $O(n^2)$-size monotone read-once formulas that amplify $(\frac{1}{4},\frac{3}{4})$ to $(2^{-n},1-2^{-n})$. The $O(n^{\alpha+2})$-size monotone read-once formulas that amplify $(\psi-\frac{1}{n},\psi+\frac{1}{n})$ to $(2^{-n},1-2^{-n})$ are easily obtained by combining these two subconstructions. Boppana was able to show that each one of these subconstructions achieved an optimal amount of amplification. However, it does not seem to follow from Boppana's arguments that the combined construction is also optimal.

We are able to join together the two lower bounds of Boppana and show that any monotone read-once formula that amplifies $(p-\frac{1}{n},p+\frac{1}{n})$ to $(2^{-n},1-2^{-n})$ (where $0<p<1$ is fixed) does indeed have to be of size $\Omega(n^{\alpha+2})$. This gives a complete positive answer to the question of whether Valiant's construction obtains an optimal amount of amplification.

We are also able to strengthen Boppana's results in another respect. We show that the combined $\Omega(n^{\alpha+2})$ lower bound applies even if the read-once formulas are allowed to use negations and exclusive-or (XOR) gates (as well as the monotone AND and OR gates). We deal with negations directly. XOR gates are dealt with by showing that for every formula with XOR gates there exists a probabilistic formula without XOR gates of the same size that achieves the same amount of amplification.

Boppana had used two different (we are tempted to say incompatible) methods to get his two lower bounds for the two amplification stages. To obtain the combined lower bound, we have to slightly strengthen his first lower bound and exhibit an alternative proof for his second. In particular, we obtain a slightly stronger version of what we call *Boppana's inequality* (see section 2) and present an analytical proof of it. (Boppana resorted to numerical experimentation in the proof of his inequality.) We also generalize Boppana's bound on the derivatives of univariate amplification functions to bounds on the partial derivatives of multivariate amplification functions.

As mentioned, Boppana's second lower bound applies to general monotone contact networks. Using a result of Lupanov [13], it can be shown that it does not apply to nonmonotone contact networks. Thus in the contact-networks model, negations do help amplify. More details can be found in [5] and [6]. We also note that negations and XOR gates seem to help in the construction of formulas for the majority function. In [15] and [16], nonmonotone formulas of size $O(n^{4.57})$ without XOR gates and of size $O(n^{3.13})$ with XOR gates were constructed.

In a companion paper [6] (see also [5]), we show that there exist monotone undirected contact networks of size $O(n^{2.99})$ that amplify $(\frac{1}{2}-\frac{1}{n},\frac{1}{2}+\frac{1}{n})$ to $(\frac{1}{4},\frac{3}{4})$. Certain conjectures in percolation theory imply that the size of the amplifying networks can be further reduced to $O(n^{8/3+o(1)})$ and perhaps even further. This extends the results of Moore and Shannon [14] and those of Valiant [19] and shows that Boppana's first lower bound does not apply to the contact networks, even if they are required to be both monotone and read-once. It also implies the existence of *undirected* monotone contact networks of size $O(n^{4.99})$ (or $O(n^{4.67})$, relying on the percolation conjectures)

that compute the majority of n bits. Smaller *directed* monotone networks for majority have recently been constructed by Radhakrishnan and Subrahmanyam [18].

Other works relevant to the subjects considered in this paper are [8], [9], [10], and [17].

The rest of the paper is organized as follows. Section 2 is mainly composed of definitions. In section 3 we present the strengthened version of Boppana's inequality. In section 4, we obtain the bounds on the derivatives of multivariate and univariate amplification functions. These bounds are used in section 5 to strengthen Boppana's bound for the first amplification stage. In section 6, we put forth a new approach to proving amplification lower bounds. This approach is based on a simple functional inequality. Using this approach, we present an alternative proof to Boppana's lower bound for the second amplification stage. In section 7, we show how to combine the methods of the two preceding sections and obtain our unified lower bound. In section 8, we show that read-once formulas that include XOR gates can be simulated (as far as amplification is concerned) by probabilistic read-once formulas without XOR gates with no increase in size. All the results of sections 5, 6, and 7 are valid for probabilistic, not only deterministic, formulas. Thus all of these results, proved so far only for formulas without XOR gates, remain valid even if XOR gates are used. We conclude in section 9 with some open problems.

Although the basic ideas used in this work are fairly simple, many proofs, especially those of inequalities, are extremely technical. To maintain the readability of the paper, the more technical and lengthy proofs have been put in the appendices.

A preliminary version of this paper (and of [6]) appeared in [5].

2. Preliminaries. Following Moore and Shannon [14] and Boppana [4], we introduce the following two definitions

DEFINITION 2.1 (amplification functions). *Given a Boolean function* $f : \{0,1\}^n \to \{0,1\}$*, we define its* multivariate amplification function $f : [0,1]^n \to [0,1]$ *as follows:* $f(p_1, p_2, \ldots, p_n) = \Pr[f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) = 1]$*, where* $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ *are independent random variables and* $\mathbf{x}_i$ *assumes the value* 1 *with probability* p_i *and the value* 0 *with probability* $1 - p_i$*. We define the* univariate amplification function *of* f *to be* $f(p) = f(p, p, \ldots, p)$*. Since the (multivariate) amplification function is an extension of the original Boolean function, we use the same notation for both.*

DEFINITION 2.2 (amplification from/to). *A Boolean function* $f : \{0,1\}^n \to \{0,1\}$ amplifies (p, q) *to* (p', q') *if* $f(p) = p'$ *and* $f(q) = q'$*. It amplifies* (p, q) *to* at least (p', q') *if* $f(p) \le p'$ *and* $f(q) \ge q'$*.*

The main objects considered in this paper are formulas.

DEFINITION 2.3 (formulas). *A* formula *of* n *variables is defined recursively as follows:* (i) *for* $1 \le i \le n$*, the variables* x_i *and their negations* $\overline{x_i}$ *are formulas*; (ii) *if* f *and* g *are formulas, then so are* $(\neg f)$*,* $(f \wedge g)$*,* $(f \vee g)$*, and* $(f \oplus g)$*. A formula in which no XOR gates are used is called* unate *or* de Morgan*. A unate formula in which no negations are used is called* monotone*. Formulas define Boolean functions in the obvious way. The size of a formula* f*, denoted by* $size(f)$*, is the number of occurrences of variables in it.*

DEFINITION 2.4 (probabilistic formulas). *A* probabilistic *(monotone, unate) formula is simply a discrete probability distribution over deterministic (monotone, unate) formulas. The size of a probabilistic formula* F *is defined to be the maximum size of a formula* f *whose probability according to the distribution induced by* F *is positive. The amplification function* $F(p_1, \ldots, p_n)$ *of a probabilistic formula* F *is simply*

$F(p_1, \ldots, p_n) = E_{f \in F} f(p_1, \ldots, p_n)$, where E denotes expectation relative to the distribution induced by F.

It is easy to see that monotone and unate formulas correspond to monotone and nonmonotone *series-parallel* contact networks. For more details the reader is referred to Boppana [4].

Most of the formulas encountered in this work will be read-once.

DEFINITION 2.5 (read-once). *A formula is said to be* read-once *if every variable appears in it at most once.*

We investigate the minimal size of formulas required to achieve given amplification goals.

DEFINITION 2.6 (amplification complexity). *If $0 < p_0, q_0, p_1, q_1 < 1$, we denote by $N(p_1, q_1 \mid p_0, q_0)$ the minimum size of a unate read-once formula that amplifies (p_0, q_0) to (p_1, q_1). We denote by $N_\oplus(p_1, q_1 \mid p_0, q_0)$ the corresponding measure for general read-once formulas.*

In what follows, we refer to the following two inequalities.

THEOREM 2.7 (Hölder's inequality). *If $\alpha, \beta > 1$ and $\frac{1}{\alpha} + \frac{1}{\beta} = 1$, then for any two real vectors x and y, we have*

$$\langle |x|, |y| \rangle = \sum_{i=1}^{n} |x_i y_i| \leq \left(\sum_{i=1}^{n} |x_i|^\alpha \right)^{1/\alpha} \cdot \left(\sum_{i=1}^{n} |y_i|^\beta \right)^{1/\beta} = \|x\|_\alpha \cdot \|y\|_\beta .$$

THEOREM 2.8 (Boppana's inequality). *Let $H(x) = -x \log_2 x - (1-x) \log_2 (1-x)$ be the usual binary entropy function and let $\beta = \log_{(\sqrt{5}+1)/2} 2 \simeq 1.44$. Then for every $0 < x, y \leq 1$, we have*

$$\left(\frac{H(x)}{x} \right)^\beta + \left(\frac{H(y)}{y} \right)^\beta \leq \left(\frac{H(xy)}{xy} \right)^\beta .$$

In what follows we say that a function F satisfies Boppana's inequality iff $F(x) = F(1-x) \geq 0$ for every $0 \leq x \leq 1$ and F (substituting for H) satisfies the inequality in the above theorem.

3. Strengthening Boppana's inequality. Boppana's inequality, stated in the previous section, forms the basis of his lower bound for the first amplification stage. To get some of our extended lower bounds, we need a slightly stronger version of his inequality. This version is obtained by replacing the entropy function $H(x)$ by a function $G(x)$ with the tightest possible asymptotic behavior near $x = 0$ and $x = 1$.

To prove his inequality, Boppana had to make many numerical checks involving functions of two variables. (In one point of the argument, for example, he has to estimate the third derivatives of the function $\widetilde{H}(x, y) = (H(xy)/xy)^\beta - (H(x)/x)^\beta - (H(y)/y)^\beta$ and show that the Hessian matrix of $\widetilde{H}$ is positive definite on the entire region $[0.55, 0.65] \times [0.55, 0.65]$.) We are able to exhibit a much simpler proof to our strengthened inequality. We show that it follows from some relatively simple one-variable inequalities.

Our new function $G(x)$ is defined as follows:

$$G(x) = \begin{cases} x \left(\ln \frac{\gamma}{x} \right)^{1/\beta} & \text{if } 0 \leq x \leq \frac{1}{2}, \\ (1-x) \left(\ln \frac{\gamma}{1-x} \right)^{1/\beta} & \text{if } \frac{1}{2} \leq x \leq 1, \end{cases}$$

where

$$\ln \gamma = \frac{2 - \psi - \ln 4}{\beta} \simeq -0.003005.$$

We claim that $G(x)$ satisfies Boppana's inequality.

LEMMA 3.1. *For every $0 < x, y \leq 1$, we have $(\frac{G(x)}{x})^\beta + (\frac{G(y)}{y})^\beta \leq (\frac{G(xy)}{xy})^\beta$.*

Proof. See Appendix A. □

The somewhat peculiar constant γ in the definition of $G(x)$ was chosen so that $\widehat{G}(\psi) = \widehat{G}(1-\psi)$ where $\widehat{G}(x) = x \cdot \frac{d}{dx}(\frac{G(x)}{x})^\beta$. As shown in the next lemma, this is a necessary condition that must be satisfied by any (symmetric) function that wishes to satisfy Boppana's inequality.

LEMMA 3.2. *If $G(x)$ is a symmetric function that satisfies Boppana's inequality and if $G(x)$ is differentiable at $x = \psi$, then $\widehat{G}(\psi) = \widehat{G}(1-\psi)$, where $\widehat{G}(x) = x \cdot \frac{d}{dx}(\frac{G(x)}{x})^\beta$.*

Proof. Since $\psi^2 = 1 - \psi$, $\psi^\beta = \frac{1}{2}$, and $G(\psi^2) = G(1-\psi) = G(\psi)$, we get that $2(\frac{G(\psi)}{\psi})^\beta = (\frac{G(\psi^2)}{\psi^2})^\beta$. It follows that Boppana's inequality is always satisfied with equality at the point (ψ, ψ). As a consequence, the point (ψ, ψ) must be a local minimum of the function $\widetilde{G}(x,y) = \widetilde{G}(xy) - \widetilde{G}(x) - \widetilde{G}(y)$, where $\widetilde{G}(x) = (\frac{G(x)}{x})^\beta$. It is easy to verify that

$$\frac{\partial \widetilde{G}(x,y)}{\partial x} = \frac{1}{x}\left[xy\widetilde{G}'(xy) - x\widetilde{G}'(x)\right] = \frac{1}{x}\left[\widehat{G}(xy) - \widehat{G}(x)\right].$$

Since this partial derivative must vanish at (ψ, ψ), we get that $\widehat{G}(1-\psi) = \widehat{G}(\psi^2) = \widehat{G}(\psi)$. □

Graphs of the function $G(x)$ are discussed in the appendices. It can be seen from these graphs that $G(x)$ has a small cusp at $x = \frac{1}{2}$. This will not cause us any trouble. A variant of the function $G(x)$ in which this cusp is replaced by a straight line connecting the two local maxima also satisfies Boppana's inequality. This variant is used by us in [7].

It is easy to see that for small values of x, we have $G(x) \approx x(\ln \frac{1}{x})^{1/\beta}$ while $H(x) \approx x \ln \frac{1}{x}$ and thus $G(x) \ll H(x)$. This allows us to get the improved amplification bounds.

4. Bounds on derivatives of amplification functions. The basic theorem from which all the results of this section follow is the following.

THEOREM 4.1. *If f is a unate read-once formula (or a read-once series-parallel contact network) that depends on n variables and if $I(x)$ satisfies Boppana's inequality, then*

$$\left(\sum_{i=1}^{n} \left| I(x_i) \cdot \frac{\partial f}{\partial x_i} \right|^\beta \right)^{1/\beta} \leq I(f).$$

Proof. The proof is by induction on the structure of f. If $f = x_i$ or $f = \overline{x_i}$, then the inequality is easily verified since $|\frac{\partial f}{\partial x_i}| = 1$ and $I(x_i) = I(1-x_i) = I(f)$. If $f = \neg g$, then the inequality follows easily from the induction hypothesis since $f = 1 - g$ and therefore $|\frac{\partial f}{\partial x_i}| = |\frac{\partial g}{\partial x_i}|$ and $I(f) = I(1-f) = I(g)$.

If $f = f_1 \wedge f_2$ and if $f_1(x_1, \ldots, x_n)$ and $f_2(y_1, \ldots, y_m)$ are the amplification functions of f_1 and f_2, then the amplification function of f is $f(x_1, \ldots, x_n, y_1, \ldots, y_m) =$

$f_1(x_1,\ldots,x_n)f_2(y_1,\ldots,y_m)$. As a consequence, $\frac{\partial f}{\partial x_i} = \frac{\partial f_1}{\partial x_i}\cdot f_2$ and $\frac{\partial f}{\partial y_j} = f_1\cdot\frac{\partial f_2}{\partial y_j}$. For convenience, we let $x_{n+1} = y_1,\ldots,x_{n+m} = y_m$.

Using the induction hypothesis and then Boppana's inequality, we therefore get

$$\begin{aligned}\sum_{i=1}^{n+m}\left|I(x_i)\cdot\frac{\partial f}{\partial x_i}\right|^\beta &\le f_2^\beta\cdot\sum_{i=1}^{n}\left|I(x_i)\cdot\frac{\partial f_1}{\partial x_i}\right|^\beta + f_1^\beta\cdot\sum_{i=n+1}^{n+m}\left|I(x_i)\cdot\frac{\partial f_2}{\partial x_i}\right|^\beta\\ &\le (f_2\cdot I(f_1))^\beta + (f_1\cdot I(f_2))^\beta \;=\; (f_1f_2)^\beta\cdot\left[\left(\frac{I(f_1)}{f_1}\right)^\beta + \left(\frac{I(f_2)}{f_2}\right)^\beta\right]\\ &\le (f_1f_2)^\beta\cdot\left(\frac{I(f_1f_2)}{f_1f_2}\right)^\beta \;=\; (I(f))^\beta.\end{aligned}$$

This completes the proof since any unate gate can be obtained by combining an $\wedge$-gate with negations. □

If in addition to satisfying Boppana's inequality, the function $I(x)$ also satisfies the inequality

$$\left(\frac{I\left(\frac{1-x}{2}\right)}{x}\right)^\beta + \left(\frac{I\left(\frac{1-y}{2}\right)}{y}\right)^\beta \le \left(\frac{I\left(\frac{1-xy}{2}\right)}{xy}\right)^\beta$$

for every $0 < x, y \le 1$, then we can show directly that the inequality in the above lemma holds for read-once formulas that may include XOR gates. Both the entropy function $H(x)$ and the function $G(x)$ introduced in section 3 satisfy this inequality. We omit the details since the same amplification bounds will follow from the more general arguments of section 8.

We denote by $\nabla f(x)$ the *gradient* $\nabla f = (\frac{\partial f}{\partial x_1},\ldots,\frac{\partial f}{\partial x_n})$ of f evaluated at the point $(x,\ldots,x)$. As an immediate corollary to the previous lemma, we get the following.

COROLLARY 4.2. *If f is a unate read-once formula, then*

$$\|\nabla f(x)\|_\beta \le \frac{I(f(x))}{I(x)},$$

and if p_c is the critical probability *for which $f(p_c) = p_c$, then*

$$\|\nabla f(p_c)\|_\beta \le 1.$$

As a second corollary to Theorem 4.1, we get the following result obtained by Boppana [4] for monotone read-once formulas.

COROLLARY 4.3. *If f is a unate read-once formula that depends on n variables, then*

$$|f'(x)| \le n^{1/\alpha}\cdot\frac{I(f(x))}{I(x)}.$$

Proof. A simple application of Hölder's inequality yields

$$\begin{aligned}|f'(x)| \le \sum_{i=1}^{n}\left|\frac{\partial f}{\partial x_i}(x)\right| &\;\le\; \left(\sum_{i=1}^{n}1\right)^{1/\alpha}\cdot\left(\sum_{i=1}^{n}\left|\frac{\partial f}{\partial x_i}(x)\right|^\beta\right)^{1/\beta}\\ &= n^{1/\alpha}\cdot\|\nabla f(x)\|_\beta \;\le\; n^{1/\alpha}\cdot\frac{I(f(x))}{I(x)}.\end{aligned}$$

□

It was pointed out by one of the referees that the inequality $\sum_{i=1}^{n} |x_i| \le n^{1/\alpha} \cdot (\sum_{i=1}^{n} |x_i|^{\beta})^{1/\beta}$ used in the proof above follows immediately from Jensen's inequality, which is more elementary than Hölder's inequality. This is interesting since it shows that our proof, in contrast to Boppana's proof, does not use the full power of Hölder's inequality.

Again, if $I(x)$ satisfies the additional inequality mentioned above, as G and H do, then Corollary 4.3 holds for general, not necessarily unate, read-once formulas. We claim that the inequality in Corollary 4.3 with $G(x)$ plugged into it is, up to constant factors, the strongest valid inequality of its kind. We do not elaborate on it here.

5. Local amplification bounds. Boppana [4] used Corollary 4.3 with the entropy function $H(x)$ plugged into it together with the mean-value theorem to get a lower bound on the size of the formulas required to amplify (p_0, q_0) to (p_1, q_1). A stronger lower bound is obtained by replacing the entropy function by the new function $G(x)$ and by integrating the upper bound obtained for $|f'(x)|$.

Let

$$M(x,y) = \left| \int_x^y \frac{du}{G(u)} \right|^{\alpha}.$$

THEOREM 5.1. *Any read-once formula that amplifies* (p_0, q_0) to (p_1, q_1) has size at least

$$N(p_1, q_1 \mid p_0, q_0) \ge \frac{M(p_1, q_1)}{M(p_0, q_0)}.$$

Proof. Let f be a read-once formula that depends on n variables and amplifies (p_0, q_0) to (p_1, q_1). Using Corollary 4.3 and a simple change of variables, we get that

$$\left| \int_{p_1}^{q_1} \frac{dx}{G(x)} \right| \le \left| \int_{p_0}^{q_0} \frac{f'(x)dx}{G(f(x))} \right| \le \left| \int_{p_0}^{q_0} \frac{n^{1/\alpha} dx}{G(x)} \right| = n^{1/\alpha} \left| \int_{p_0}^{q_0} \frac{dx}{G(x)} \right|. \qquad \square$$

Since

$$\int \frac{dx}{x \left(\ln \frac{\gamma}{x}\right)^{1/\beta}} = -\alpha \left(\ln \frac{\gamma}{x}\right)^{1/\alpha},$$

it is easily verified that

$$M(x,y) = \left| \int_x^y \frac{du}{G(u)} \right|^{\alpha} = C \cdot \begin{cases} m(x,y) & \text{if } x \le y \le \frac{1}{2}, \\ (m(x,\frac{1}{2})^{1/\alpha} + m(1-y,\frac{1}{2})^{1/\alpha})^{\alpha} & \text{if } x \le \frac{1}{2} \le y, \\ m(1-y,1-x) & \text{if } \frac{1}{2} \le x \le y, \\ M(y,x) & \text{if } y < x, \end{cases}$$

where $C = \alpha^{\alpha}$ and

$$m(x,y) = \left[\left(\ln \frac{\gamma}{x}\right)^{1/\alpha} - \left(\ln \frac{\gamma}{y}\right)^{1/\alpha} \right]^{\alpha}.$$

Clearly, $M(x,y) = M(y,x) = M(1-x,1-y)$.

In section 7, the function $M(x,y)$ will be compared with some other functions. It will be convenient to scale $M(x,y)$ before these comparisons and assume that $C = 10^6$. Of course, this will not affect the validity of Theorem 5.1.

As an immediate corollary of Theorem 5.1, we get the following.

COROLLARY 5.2. *Any unate read-once formula that amplifies* $(p-\frac{1}{n},p+\frac{1}{n})$ *to at least* $(\frac{1}{4},\frac{3}{4})$ *(where* $0<p<1$ *is fixed) is of size* $\Omega(n^{\alpha})$ *and any unate read-once formula that amplifies* $(p-\frac{1}{n},p+\frac{1}{n})$ *to at least* $(2^{-n},\frac{1}{2})$ *or to at least* $(\frac{1}{2},1-2^{-n})$ *is of size* $\Omega(n^{\alpha+1})$.

Proof. The proof follows easily since $M(p-\frac{1}{n},p+\frac{1}{n})=\Theta(n^{-\alpha})$ and $M(2^{-n},\frac{1}{2})$, $M(\frac{1}{2},1-2^{-n})=\Theta(n)$. □

It can be checked that both of these bounds are optimal. Valiant [19] obtained monotone formulas of size $O(n^{\alpha})$ that amplify $(p-\frac{1}{n},p+\frac{1}{n})$ to $(\frac{1}{4},\frac{3}{4})$. Boppana [4] obtained monotone formulas of size $O(n^{\alpha+1})$ that amplify $(p-\frac{1}{n},p+\frac{1}{n})$ to $(2^{-n},\frac{1}{2})$ or to $(\frac{1}{2},1-2^{-n})$. The fact that the second lower bound obtained using the function $G(x)$ is tight shows that the asymptotic behavior of $G(x)$ near $x=0$ and $x=1$ cannot be improved.

Since $M(2^{-n},1-2^{-n})$ is still $\Theta(n)$, the methods used so far imply only an $\Omega(n^{\alpha+1})$ lower bound on amplification from $(p-\frac{1}{n},p+\frac{1}{n})$ to $(2^{-n},1-2^{-n})$. This will be improved in section 7 to a tight $\Omega(n^{\alpha+2})$ lower bound.

We note that the use of the function $G(x)$ in place of $H(x)$ is essential in obtaining the $\Omega(n^{\alpha+1})$ lower bounds of Corollary 5.2. Had we used $H(x)$, we would have obtained only $\Omega(n^{\alpha}\log n)$ lower bounds.

6. Global amplification bounds. The main result of this section is the following simple yet powerful theorem. Let $(0,1)$ denote the open unit interval and let $(0,1)^2$ denote the open unit square in the plane.

THEOREM 6.1. *If* $L(x,y)$ *is defined on* $(0,1)^2$ *and satisfies the following two conditions,*

$$\begin{aligned} L(x_1x_2,y_1y_2) &\le L(x_1,y_1)+L(x_2,y_2) \quad \forall\ 0<x_1,x_2,y_1,y_2<1,\\ L(x,y) &= L(1-x,1-y) \qquad\qquad \forall\ 0<x,y<1, \end{aligned}$$

then for every $0<p_0,q_0,p_1,q_1<1$, *we have*

$$N(p_1,q_1 \mid p_0,q_0) \ge \frac{L(p_1,q_1)}{L(p_0,q_0)}\ .$$

Proof We prove by induction on the structure of f that if f amplifies (p_0,q_0) to (p_1,q_1), then $\mathit{size}(f)\ge L(p_1,q_1)/L(p_0,q_0)$. The basis of the induction is easily established.

If $f=\neg g$, then g amplifies (p_0,q_0) to $(1-p_1,1-q_1)$. Since $\mathit{size}(f)=\mathit{size}(g)$ and $L(1-p_1,1-q_1)=L(p_1,q_1)$, the required inequality follows immediately from the induction hypothesis.

If $f=f_1\wedge f_2$ and if f_1 and f_2 amplify (p_0,q_0) to (x_1,y_1) and (x_2,y_2), respectively, then $p_1=x_1x_2$ and $q_1=y_1y_2$. By the induction hypothesis and the first condition of L, we have

$$\begin{aligned} \mathit{size}(f) &= \mathit{size}(f_1)+\mathit{size}(f_2)\\ &\ge \frac{L(x_1,y_1)+L(x_2,y_2)}{L(x_0,y_0)} \ge \frac{L(x_1x_2,y_1y_2)}{L(p_0,q_0)} = \frac{L(p_1,q_1)}{L(p_0,q_0)}. \end{aligned}$$ □

Note that in both Theorems 5.1 and 6.1, the lower bounds obtained were of a similar form. Both involved a quotient of some two-variable *potential* function evaluated at the pre- and post-amplification probabilities.

The approach of Theorem 6.1 seems to be more general than the approach of the previous section. Numerical tests seem to suggest that the function $M(x, y)$ of section 5 satisfies the condition $M(x_1x_2, y_1y_2) \le M(x_1, y_1)+M(x_2, y_2)$ of Theorem 6.1. If this were indeed the case, then Theorem 5.1 would follow immediately from Theorem 6.1.

With some additional work, the numerical tests that we performed can probably be turned into a proof that is similar in spirit to the original proof that Boppana had given for his inequality—that the function $M(x, y)$ satisfies the inequality $M(x_1x_2, y_1y_2) \le M(x_1, y_1) + M(x_2, y_2)$. This would be tedious, however, since four, and not just two, variables are involved this time. We did not carry out this extra work since the direct proof given to Theorem 5.1 seems to be more informative.

We now use Theorem 6.1 to obtain an alternative proof to the lower bounds for the second amplification stage obtained by Moore and Shannon [14] and Boppana [4]. The new proof works only for read-once formulas and not for general read-once networks. However, it does work for general, not necessarily monotone, read-once formulas.

Perhaps the simplest function satisfying the conditions of Theorem 6.1 is the following function:

$$L'(x, y) = \begin{cases} \ln \frac{1}{x} \cdot \ln \frac{1}{1-y} & \text{if } x \le y, \\ \ln \frac{1}{1-x} \cdot \ln \frac{1}{y} & \text{if } x \ge y. \end{cases}$$

LEMMA 6.2. *The function $L'(x, y)$ satisfies the conditions of Theorem* 6.1.

Proof. The condition $L'(x, y) = L'(1 - x, 1 - y)$ is easily verified. Let $\ell'(x, y) = \ln \frac{1}{x} \cdot \ln \frac{1}{1-y}$. Note that $\ell'(1 - x, 1 - y) = \ell'(y, x)$. When $x \le y$, we have $\frac{1}{x} \ge \frac{1}{y}$ and $\frac{1}{1-y} \ge \frac{1}{1-x}$ and therefore

$$\ell'(x, y) = \ln \tfrac{1}{x} \cdot \ln \tfrac{1}{1-y} \ge \ln \tfrac{1}{1-x} \cdot \ln \tfrac{1}{y} = \ell'(y, x).$$

When $x \ge y$, the opposite inequality holds. Thus $L'(x, y) = \max\{\ell'(x, y), \ell'(y, x)\}$. We now have

$$\begin{aligned} \ell'(x_1x_2, y_1y_2) &= \ln \tfrac{1}{x_1x_2} \cdot \ln \tfrac{1}{1-y_1y_2} \\ &= \left(\ln \tfrac{1}{x_1} + \ln \tfrac{1}{x_2}\right) \cdot \ln \tfrac{1}{1-y_1y_2} \le \ln \tfrac{1}{x_1} \cdot \ln \tfrac{1}{1-y_1} + \ln \tfrac{1}{x_2} \ln \tfrac{1}{1-y_2} \\ &= \ell'(x_1, y_1) + \ell'(x_2, y_2) \le L'(x_1, y_1) + L'(x_2, y_2) \end{aligned}$$

and

$$\ell'(y_1y_2, x_1x_2) \le \ell'(y_1, x_1) + \ell'(y_2, x_2) \le L'(x_1, y_1) + L'(x_2, y_2).$$

Consequently,

$$L'(x_1x_2, y_1y_2) = \max\{\ell'(x_1x_2, y_1y_2), \ell'(y_1y_2, x_1x_2)\} \le L'(x_1, y_1) + L'(x_2, y_2),$$

as required. □

As an immediate corollary, we extend the lower bound of Moore and Shannon [14] to include nonmonotone read-once formulas.

COROLLARY 6.3. *Every unate read-once formula that amplifies $(\frac{1}{4}, \frac{3}{4})$ to at least $(2^{-n_1}, 1 - 2^{-n_2})$ is of size $\Omega(n_1n_2)$.*

Boppana [4] obtained a stronger version of this inequality when n_1 and n_2 are not within an exponent of each other. To extend his result, we use Theorem 6.1 in

conjunction with the more complicated function

$$L(x,y) = \begin{cases} \ell(x,y) & \text{if } x \le y \le 1-x, \\ \ell(1-y,1-x) & \text{if } x, 1-x \le y, \\ \ell(1-x,1-y) & \text{if } 1-x \le y \le x, \\ \ell(y,x) & \text{if } y \le x, 1-x, \end{cases}$$

where

$$\ell(x,y) = \ln\frac{1}{x} \cdot \ln\frac{\ln\frac{1}{x}}{\ln\frac{1}{y}}.$$

LEMMA 6.4. *The function $L(x,y)$ satisfies the conditions of Theorem* 6.1.

Proof. The proof is tedious and it is presented in Appendix B. □

Using $L(x,y)$, we extend Boppana's second lower bound to nonmonotone read-once formulas.

COROLLARY 6.5. *Every unate read-once formula that amplifies $(\frac{1}{4},\frac{3}{4})$ to at least $(2^{-n_1}, 1-2^{-n_2})$ is of size $\Omega(n_1 n_2 + n_1 \log n_1 + n_2 \log n_2)$.*

The results of this section are stated for unate read-once formulas. The validity of all the results for general read-once formulas will follow from the results of section 8.

7. A unified lower bound for amplification. Corollary 5.2 gives in particular an $\Omega(n^{\alpha+1})$ lower bound on the size of unate read-once formulas that amplify $(p-\frac{1}{n}, p+\frac{1}{n})$ to at least $(2^{-n}, 1-2^{-n})$ (for a fixed $0<p<1$). In this section, we combine the proof techniques of sections 5 and 6 and improve this to a tight $\Omega(n^{\alpha+2})$ lower bound. We begin with the following generalization of Theorem 6.1.

THEOREM 7.1. *If $K(x,y)$ is defined on $(0,1)^2$ and satisfies the conditions*

$$\begin{aligned} K(x_1x_2, y_1y_2) &\le \max\{K(x_1,y_1) + K(x_2,y_2),\ M(x_1x_2, y_1y_2)\}, \\ K(x,y) &\ge M(x,y), \\ K(x,y) &= K(1-x,1-y) \end{aligned}$$

for every $0 < x_1, x_2, y_1, y_2 < 1$ and $0 < x, y < 1$, and if it is already known for every $0 < p_0, p_1, q_0, q_1 < 1$ that

$$N(p_1,q_1 \mid p_0,q_0) \ge \frac{M(p_1,q_1)}{M(p_0,q_0)},$$

then we also have for every $0 < p_0, p_1, q_0, q_1 < 1$ that

$$N(p_1,q_1 \mid p_0,q_0) \ge \frac{K(p_1,q_1)}{K(p_0,q_0)}.$$

Proof. The proof is a trivial modification of the proof of Theorem 6.1. □

To get our unified lower bound, we apply Theorem 7.1 to a function $K(x,y)$ obtained by stitching together $M(x,y)$ of section 5 (with $C = 10^6$), which did well for the first amplification stage, and $L(x,y)$ of section 6, which did well for the second amplification stage. The function $K(x,y)$ will equal $M(x,y)$ when x and y are very close to one another and it will equal $L(x,y)$ when they are far apart.

The function $K(x,y)$ is defined as follows:

$$K(x,y) = \begin{cases} M(x,y) & \text{if } (x,y) \in \mathcal{A}, \\ \max\{M(x,y), L(x,y)\} & \text{if } (x,y) \notin \mathcal{A}, \end{cases}$$

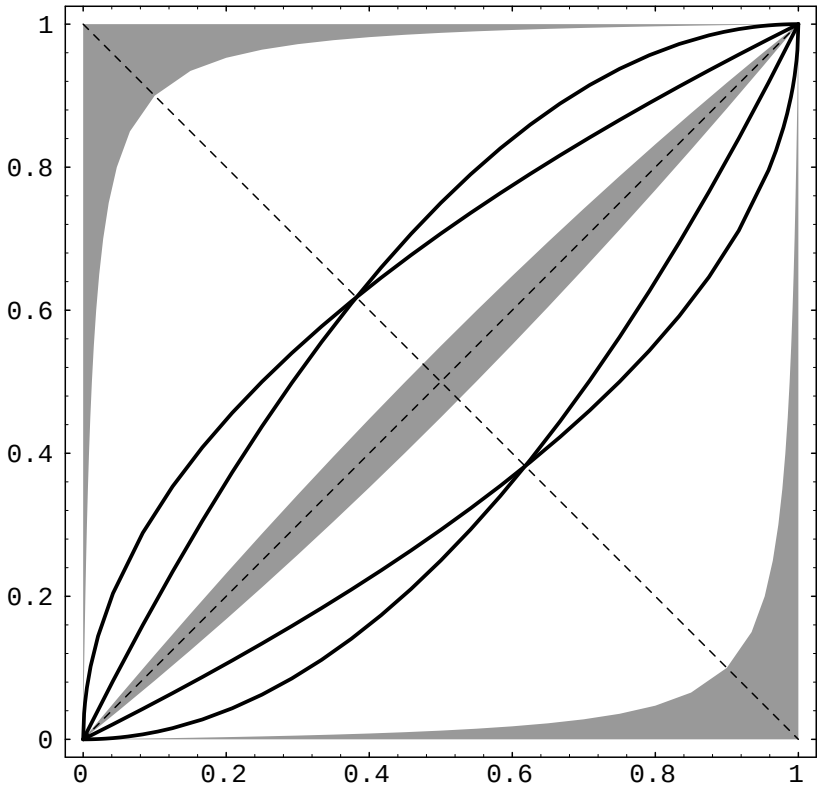


FIG. 1. *The region $\mathcal{A}$ (composed of $\mathcal{A}'$ and $\mathcal{A}''$) and the regions in which $L(x,y) \geq M(x,y)$.*

where

$$\mathcal{A} = \mathcal{A}' \cup \mathcal{A}'', \qquad \mathcal{A}' = \mathcal{R}'_2, \qquad \mathcal{A}'' = \mathcal{R}''_2$$

and

$$\mathcal{R}'_d = \left\{ (x,y) : \frac{1}{d} \leq \left| \frac{\ln \frac{1}{x}}{\ln \frac{1}{y}} \right| \leq d \right\}, \qquad \mathcal{R}''_d = \left\{ (x,y) : \frac{1}{d} \leq \left| \frac{\ln \frac{1}{1-x}}{\ln \frac{1}{1-y}} \right| \leq d \right\}.$$

Figure 1 gives a schematic description of the region $\mathcal{A}$ (composed of $\mathcal{A}'$ and $\mathcal{A}''$) and the two regions (shown shaded) in which $L(x,y) \geq M(x,y)$. These two extremely thin regions stretch along the diagonal and the boundary of the unit square. It can be checked (see the proof of Claim 7.4(a) in Appendix C) that when $x \leq y$ and when both x and y are very small, then $L(x,y)/M(x,y) \approx \ln u / C[1 - (1/u)^{1/\alpha}]^{\alpha}$, where $u = \ln \frac{1}{x} / \ln \frac{1}{y}$. Thus $L(x,y) \geq M(x,y)$ roughly when $u \leq 1.01$ or when $u \geq e^{10^6}$, i.e., when $y^{1.01} \leq x \leq y$ or when $y^{e^{10^6}} \geq x$. The definition of $\mathcal{A}$ was conveniently chosen to lie between these two regions. Note that although it is impossible to see it in Figure 1, the upper boundary of the inner region in which $L(x,y) \geq M(x,y)$ is also tangent to the y-axis.

It is immediate that $K(x,y)$ satisfies the last two conditions of Theorem 7.1. To show that it also satisfies the first, we define the following additional regions:

$$\begin{array}{lll} \mathcal{B}' = \mathcal{R}'_4, & \mathcal{B}'' = \mathcal{R}''_4, & \mathcal{B} = \mathcal{B}' \cup \mathcal{B}'', \\ \mathcal{C}' = \mathcal{R}'_6, & \mathcal{C}'' = \mathcal{R}''_6, & \mathcal{C} = \mathcal{C}' \cup \mathcal{C}''. \end{array}$$

Clearly, $\mathcal{A} \subseteq \mathcal{B} \subseteq \mathcal{C}$. Note that if $(x_1x_2, y_1y_2) \in \mathcal{A}$, then $K(x_1x_2, y_1y_2) = M(x_1x_2, y_1y_2)$. The fact that the function $K(x, y)$ satisfies the first condition of Theorem 7.1 will therefore follow from the following lemma.

LEMMA 7.2.

$$L(x_1x_2, y_1y_2) \leq \begin{cases} K(x_1, y_1) + K(x_2, y_2) & \text{if } (x_1x_2, y_1y_2) \notin \mathcal{C}, \\ M(x_1x_2, y_1y_2) & \text{if } (x_1x_2, y_1y_2) \in \mathcal{C} \setminus \mathcal{A}. \end{cases}$$

As an immediate corollary of Theorem 7.1 and Lemma 7.2, we get the following.

COROLLARY 7.3. (a) *Any unate read-once formula that amplifies* $(p - \frac{1}{n_1}, p + \frac{1}{n_1})$ (*where* $0 < p < 1$ *is fixed*) *to at least* $(2^{-n_2}, 1 - 2^{-n_3})$ *is of size* $\Omega(n_1^{\alpha}(n_2n_3 + n_2 \log n_2 + n_3 \log n_3))$. (b) *Any unate read-once formula that amplifies* $(p - \frac{1}{n}, p + \frac{1}{n})$ (*where* $0 < p < 1$ *is fixed*) *to at least* $(2^{-n}, 1 - 2^{-n})$ *is of size* $\Omega(n^{\alpha+2})$.

To prove Lemma 7.2 we need the following technical claims whose proofs may be found in Appendix C.

CLAIM 7.4. (a) $M(x, y) \geq L(x, y)$ *for* $(x, y) \in \mathcal{C} \setminus \mathcal{A}$. (b) $M(x, y) \geq \ell(x, y)$ *for* $(x, y) \in \mathcal{A}'' \setminus \mathcal{A}'$.

CLAIM 7.5. (a) $(\mathcal{A}')^t \subseteq \mathcal{A}'$. (b) $(\mathcal{A}'')^t \subseteq \mathcal{A}''$ *for every* $0 \leq t \leq 1$.

CLAIM 7.6. (a) $\mathcal{A}'\mathcal{C}' \subseteq \mathcal{C}'$. (b) $(\mathcal{A}'' \cap \{x + y \geq 1\})(\mathcal{B}'' \cap \{x + y \geq 1\}) \subseteq \mathcal{C}''$.

In Claims 7.5 and 7.6, we use the definitions

$$\mathcal{R}^t = \{(x^t, y^t) : (x, y) \in \mathcal{R}\},$$
$$\mathcal{RS} = \{(x_1x_2, y_1y_2) : (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\}.$$

CLAIM 7.7. (a) *If* $(x_1, y_1) \in \mathcal{A}'$ *and* $(x_2, y_2) \in \{x \leq y\} \setminus \mathcal{C}'$, *then*

$$\frac{d}{dt}\ell({x_1}^t x_2, {y_1}^t y_2)\Big|_{t=0} \leq 0.$$

(b) *If* $(x_1, y_1) \in \mathcal{A}''$ *and* $(x_2, y_2) \in \{x + y \geq 1, x \leq y\} \setminus \mathcal{B}''$, *then*

$$\frac{d}{dt}\ell(1 - {y_1}^t y_2, 1 - {x_1}^t x_2)\Big|_{t=0} \leq \begin{cases} (2 \ln \frac{1}{y_1})^{1/4} & \text{always}, \\ 0 & \text{if } \frac{\ln \frac{1}{x_1}}{\ln \frac{1}{y_1}} \leq 1.15. \end{cases}$$

CLAIM 7.8. *If* $\ln \frac{1}{x} / \ln \frac{1}{y} \geq 1.15$ *and* $x + y \geq 1$, *then* $M(x, y) \geq (2 \ln \frac{1}{y})^{1/4}$.

The choice $C = 10^6$ was made to insure the validity of Claims 7.4 and 7.8. In the proofs of these claims, found in Appendix C, the main emphasis is on showing that there exists a value of C for which these claims are valid. The proof that the choice $C = 10^6$ is sufficient is a simple drudgery and will not be presented in full.

Relying on the preceding claims, we now prove Lemma 7.2.

Proof of Lemma 7.2. If $(x_1x_2, y_1y_2) \in \mathcal{C} \setminus \mathcal{A}$, then the claim follows from 7.4(a). We therefore assume that $(x_1x_2, y_1y_2) \notin \mathcal{C}$, in which case we have to show that $L(x_1x_2, y_1y_2) \leq K(x_1, y_1) + K(x_2, y_2)$.

Since all the functions and regions involved in the statement of the lemma are invariant under reflection by the line $y = x$ (which corresponds to switching the x and the y coordinates), we may assume without loss of generality that $x_1x_2 \leq y_1y_2$. We may further assume without loss of generality that $x_1 \leq y_1$. It is enough to prove the inequality for the case where $x_2 \leq y_2$. To see this, we note that if $x \leq x' \leq y' \leq y$, then $L(x', y') \leq L(x, y)$. Assume that we had proven the inequality for every $x_1 \leq y_1$ and $x_2 \leq y_2$. We now show how to deal with the case where $x_1 \leq y_1$ and $x_2 \geq y_2$.

Since $x_1y_2 \le x_1x_2 \le y_1y_2 \le y_1x_2$ and since $(x_1x_2, y_1y_2) \notin \mathcal{C}$ implies $(x_1y_2, y_1x_2) \notin \mathcal{C}$, we have

$$\begin{aligned} L(x_1x_2, y_1y_2) &\le L(x_1y_2, y_1x_2) \\ &\le K(x_1, y_1) + K(y_2, x_2) \\ &= K(x_1, y_1) + K(x_2, y_2) \end{aligned}$$

as required. Therefore, we will assume henceforth that $x_2 \le y_2$.

We now split the proof into two cases depending on whether (x_1x_2, y_1y_2) is below or above the line $x + y = 1$.

Case 1. $x_1x_2 + y_1y_2 \le 1$.

Note that $\mathcal{C} \cap \{x + y \le 1\} = \mathcal{C}'$. We therefore know that $(x_1x_2, y_1y_2) \notin \mathcal{C}'$.

If $(x, y) \notin \mathcal{A}'$, then $\ell(x, y) \le K(x, y)$. If $(x, y) \notin \mathcal{A}$, this follows from the definition of $K(x, y)$ and the fact that $\ell(x, y) \le L(x, y)$ (see Lemma B.1 in Appendix B), and if $(x, y) \in \mathcal{A}'' \setminus \mathcal{A}'$ this follows from Claim 7.4(b).

If $(x_1, y_1), (x_2, y_2) \notin \mathcal{A}'$, then using the inequality $\ell(x_1x_2, y_1y_2) \le \ell(x_1, y_1) + \ell(x_2, y_2)$, which is always valid (see Lemma B.1 in Appendix B) and the fact that in this case we have $L(x_1x_2, y_1y_2) = \ell(x_1x_2, y_1y_2)$, we get

$$\begin{aligned} L(x_1x_2, y_1y_2) &= \ell(x_1x_2, y_1y_2) \\ &\le \ell(x_1, y_1) + \ell(x_2, y_2) \\ &\le K(x_1, y_1) + K(x_2, y_2) \end{aligned}$$

as required.

We therefore assume without loss of generality that $(x_1, y_1) \in \mathcal{A}'$. Consider the continuous movement from (x_2, y_2) to (x_1x_2, y_1y_2) described parametrically by $({x_1}^t x_2, {y_1}^t y_2)$, where t ranges from 0 to 1. If for some $0 \le t \le 1$ we have $({x_1}^t x_2, {y_1}^t y_2) \in \mathcal{C}'$, then since $({x_1}^{1-t}, {y_1}^{1-t}) \in \mathcal{A}'$ (this follows from Claim 7.5(a)) we get using Claim 7.6(a) that $(x_1x_2, y_1y_2) = ({x_1}^{1-t} \cdot {x_1}^t x_2, {y_1}^{1-t} \cdot {y_1}^t y_2) \in \mathcal{C}'$, which is a contradiction.

The remaining possibility is therefore that $({x_1}^t x_2, {y_1}^t y_2) \notin \mathcal{C}'$ for every $0 \le t \le 1$. In particular $(x_2, y_2) \notin \mathcal{C}'$ and $\ell(x_2, y_2) \le K(x_2, y_2)$. We now rely on Claim 7.7(a), which states that in this case $\frac{d}{dt}\ell({x_1}^t x_2, {y_1}^t y_2) \le 0$ for every $0 \le t \le 1$ (since $\frac{d}{dt}\ell({x_1}^t x_2, {y_1}^t y_2)|_{t=t_0} = \frac{d}{dt}\ell({x_1}^t \cdot {x_1}^{t_0} x_2, {y_1}^t \cdot {y_1}^{t_0} y_2)|_{t=0}$), and get that

$$\begin{aligned} L(x_1x_2, y_1y_2) &= \ell(x_1x_2, y_1y_2) \\ &\le \ell(x_2, y_2) \ \le \ K(x_2, y_2), \end{aligned}$$

which is more than required.

We are left with the case where (x_1x_2, y_1y_2) is above the line $x + y = 1$.

Case 2. $x_1x_2 + y_1y_2 \ge 1$.

Note that in this case we also have $x_1 + y_1 \ge 1$, $x_2 + y_2 \ge 1$, and ${x_1}^t x_2 + {y_1}^t y_2 \ge 1$ for every $0 \le t \le 1$. We also know that $L(x_1x_2, y_1y_2) = \ell(1 - y_1y_2, 1 - x_1x_2)$.

We now repeat some of the reasonings used in the previous case. However, certain complications arise since we can rely only on Claim 7.7(b), which is weaker than its counterpart Claim 7.7(a).

If $(x, y) \notin \mathcal{A}$ (which is equivalent in this case to $(x, y) \notin \mathcal{A}''$), then by the definition of $K(x, y)$, we get that $L(x, y) \le K(x, y)$.

If $(x_1, y_1), (x_2, y_2) \notin \mathcal{A}$, then using the fact that $L(x, y)$ satisfies the condition $L(x_1x_2, y_1y_2) \leq L(x_1, y_1) + L(x_2, y_2)$ (which is the first condition of Theorem 6.1), we get that

$$\begin{aligned} L(x_1x_2, y_1y_2) &\leq L(x_1, y_1) + L(x_2, y_2) \\ &= K(x_1, y_1) + K(x_2, y_2) \end{aligned}$$

as required.

We may therefore assume without loss of generality that $(x_1, y_1) \in \mathcal{A}''$. Consider again the continuous movement from (x_2, y_2) to (x_1x_2, y_1y_2) described parametrically by $({x_1}^t x_2, {y_1}^t y_2)$, where t ranges from 0 to 1. Note that in this case the whole curve is above the line $x + y = 1$.

If for some $0 \leq t \leq 1$ we have $({x_1}^t x_2, {y_1}^t y_2) \in \mathcal{B}''$, then since $({x_1}^{1-t}, {y_1}^{1-t}) \in \mathcal{A}''$ (this follows from Claim 7.5(b)), we get using Claim 7.6(b) that $(x_1x_2, y_1y_2) \in \mathcal{C}''$, which is a contradiction.

The remaining possibility is therefore that $({x_1}^t x_2, {y_1}^t y_2) \notin \mathcal{B}''$ for every $0 \leq t \leq 1$. In particular, $(x_2, y_2) \notin \mathcal{B}''$. If $\ln \frac{1}{x_1} / \ln \frac{1}{y_1} \leq 1.15$, then using the lower part of Claim 7.7(b) and essentially the same argument as before, we are done.

Assume therefore that $\ln \frac{1}{x_1} / \ln \frac{1}{y_1} \geq 1.15$. From the upper part of Claim 7.7(b), we get that

$$\frac{d}{dt} L({x_1}^t x_2, {y_1}^t y_2) \leq (2 \ln \tfrac{1}{y_1})^{1/4} \quad \text{for every } 0 \leq t \leq 1,$$

which integrates to

$$L(x_1x_2, y_1y_2) - L(x_2, y_2) \leq (2 \ln \tfrac{1}{y_1})^{1/4} .$$

Using Claim 7.8 and the facts that $L(x_2, y_2) \leq K(x_2, y_2)$ (since $(x_2, y_2) \notin \mathcal{A}$) and $M(x_1, y_1) = K(x_1, y_1)$ (since $(x_1, y_1) \in \mathcal{A}$), we get that

$$\begin{aligned} L(x_1x_2, y_1y_2) &\leq (2 \ln \tfrac{1}{y_1})^{1/4} + L(x_2, y_2) \\ &\leq M(x_1, y_1) + K(x_2, y_2) \\ &= K(x_1, y_1) + K(x_2, y_2) \end{aligned}$$

as required. This completes the proof. □

8. Exclusive-or gates as convex combinations of unate gates. The main result of this section is the following theorem.

THEOREM 8.1. *If f is a read-once formula that amplifies (p, q) to (p', q'), then there exists a probabilistic unate read-once formula F with $size(F) \leq size(f)$ that also amplifies (p, q) to (p', q').*

Proof. The proof is by induction on the structure of f. If f is a variable, then the result is clear. If $f = \neg f_1$ or $f = f_1 \wedge f_2$, then the result follows immediately from the induction hypothesis. The interesting case is, of course, if $f = f_1 \oplus f_2$. Assume that f_1 and f_2 amplify (p, q) to (x_1, y_1) and (x_2, y_2), respectively. Let F_1 and F_2 be two probabilistic unate formulas with $size(F_1) \leq size(f_1)$ and $size(F_2) \leq size(f_2)$ that also amplify (p, q) to (x_1, y_1) and (x_2, y_2), respectively. The existence of F_1 and F_2 follows from the induction hypothesis. The next lemma proves the existence of a unate connective $\circ$ such that the point $(x_1, y_1) \oplus (x_2, y_2)$ lies in the convex hull of the points $(0, 0), (x_1, y_1) \circ (x_2, y_2)$ and $(1, 1)$. It follows that there exist three constants

$0 \le \alpha, \beta, \gamma \le 1$ with $\alpha+\beta+\gamma = 1$ such that $(x_1, y_1) \oplus (x_2, y_2) = \alpha \cdot (0,0) + \beta \cdot (x_1, y_1) \circ (x_2, y_2) + \gamma \cdot (1,1)$. Thus the unate probabilistic formula

$$F = \begin{cases} 0 & \text{with prob. } \alpha, \\ F_1 \circ F_2 & \text{with prob. } \beta, \\ 1 & \text{with prob. } \gamma \end{cases}$$

achieves the same amplification as f. Clearly, $size(F) \le size(F_1) + size(F_2) \le size(f_1) + size(f_2) = size(f)$. This completes the proof of the theorem. □

A unate connective here is one of the twelve connectives $(x^a \wedge y^b)^c$, x^a, and y^b, where $x^a = x \oplus a$, that may be obtained using one AND gate and some negations.

LEMMA 8.2. *For every $0 \le x_1, x_2, y_1, y_2 \le 1$, there exists a unate connective $\circ$ such that $(x_1, y_1) \oplus (x_2, y_2)$ is in the convex hull of the points $(0,0), (x_1, y_1) \circ (x_2, y_2)$ and $(1,1)$.*

Proof. We may assume without loss of generality that $x_1 \oplus x_2 \le y_1 \oplus y_2$ since if this is not the case, we can switch the roles of the x's and y's. By this assumption, the point $(x_1 \oplus x_2, y_1 \oplus y_2)$ is above the line $y = x$. It is therefore contained in the convex hull of the points $(0,0), (x_1, y_1) \circ (x_2, y_2)$, and $(1,1)$ iff

$$\frac{y_1 \circ y_2}{x_1 \circ x_2} \ge \frac{y_1 \oplus y_2}{x_1 \oplus x_2} \quad \text{and} \quad \frac{1 - y_1 \circ y_2}{1 - x_1 \circ x_2} \le \frac{1 - y_1 \oplus y_2}{1 - x_1 \oplus x_2}.$$

We consider four different cases.

Case 1. $x_1 \le y_1$ and $x_2 \le y_2$.

We take $\circ$ to be the OR operator, i.e., $a \circ b = a \vee b$. A simple manipulation shows that the first inequality is equivalent to the inequality

$$x_1 y_1 (y_2 - x_2) + x_2 y_2 (y_1 - x_1) \ge 0,$$

which is easily seen to hold since $x_1 \le y_1$ and $x_2 \le y_2$. A similar manipulation shows that the second inequality is equivalent in this case to the inequality

$$y_1 y_2 (1 - x_1 \vee x_2) \ge x_1 x_2 (1 - y_1 \vee y_2),$$

which is also easily seen to hold since $x_1 x_2 \le y_1 y_2$ and $x_1 \vee x_2 \le y_1 \vee y_2$.

Case 2. $x_1 \le y_1$ and $x_2 \ge y_2$.

We consider the points $(\overline{y}_1, \overline{x}_1)$ and (y_2, x_2). It is easy to check that $\overline{y}_1 \le \overline{x}_1$, $y_2 \le x_2$, and $\overline{y}_1 \oplus y_2 \le \overline{x}_1 \oplus x_2$. By Case 1, we get that the point $(\overline{y}_1 \oplus y_2, \overline{x}_1 \oplus x_2)$ lies in the convex hull of the points $(0,0), (\overline{y}_1 \vee y_2, \overline{x}_1 \vee x_2)$, and $(1,1)$. It follows immediately that the point $(x_1, x_2) \oplus (y_1, y_2) = (1,1) - (\overline{x}_1, x_2) \oplus (\overline{y}_1, y_2)$ lies in the convex hull of the points $(0,0), (x_1 \wedge \overline{x}_2, y_1 \wedge \overline{y}_2)$, and $(1,1)$. We can thus take $a \circ b = a \wedge \overline{b}$.

Case 3. $x_1 \ge y_1$ and $x_2 \le y_2$.

By switching the roles of (x_1, y_1) and (x_2, y_2), we are back in Case 2. We can thus take $a \circ b = \overline{a} \wedge b$.

Case 4. $x_1 \ge y_1$ and $x_2 \ge y_2$.

The points $(\overline{x}_1, \overline{y}_1)$ and $(\overline{x}_2, \overline{y}_2)$ satisfy the conditions of Case 1. We can thus take $a \circ b = \overline{a} \vee \overline{b}$.

This completes the proof of the lemma. □

It is easy to check that the lower bounds of the previous sections apply to probabilistic and not only to deterministic read-once formulas. All of the lower bounds claimed for unate read-once formulas are therefore valid for general read-once formulas. We also have the following connection between probabilistic and nonprobabilistic formulas.

THEOREM 8.3. *If F is a probabilistic formula that amplifies (p,q) to (p',q'), then there exists a deterministic formula f that amplifies (p,q) to (p'',q''), where $p'' \le 2p'$ and $1-q'' \le 2(1-q')$. Furthermore, if F is read-once, unate, or monotone, then so is f.*

Proof. Recall that $p' = F(p) = E_{f\in F} f(p)$, where the expectation is according to the distribution that F induces on the deterministic formulas that it assumes. Since for every nonnegative random variable X we have $\Pr[X > 2E(X)] < \frac{1}{2}$, we get that $\Pr[f(p) > 2p'] < \frac{1}{2}$, where again f is chosen according to the distribution induced by F. Applying a similar argument to $q' = F(q)$, we get that $\Pr[(1-f(q)) > 2(1-q')] < \frac{1}{2}$. As a consequence,

$$Pr[\ f(p) > 2p' \ \vee\ (1-f(q)) > 2(1-q')\ ] < 1,$$

and therefore a deterministic formula chosen according to the distribution of F will satisfy the required conditions with a positive probability. □

COROLLARY 8.4. $N_{\oplus}(p_1,q_1|p_0,q_0) \ge \min_{\substack{p'_1\le 2p_1 \\ 1-q'_1\le 2(1-q_1)}} N(p'_1,q'_1 \mid p_0,q_0)$.

9. Open problems. At least two major open problems are left in connection with the subjects discussed in this paper:

1. Do the lower bounds on amplifying formulas given in this paper apply to general, not necessarily read-once, formulas? Do they hold for general monotone formulas? Do they hold, say, for read-twice formulas?

2. Are the optimal monotone formulas for majority obtained by the use of the amplification method? Is it possible to obtain an $\Omega(n^{\alpha+2})$ lower bound on the monotone formula complexity of the majority function? The currently best lower bound on the monotone or unate formula size of the majority function is an $\Omega(n^2)$ lower bound obtained by Khrapchenko [11], [12].

One less important problem is the following:

3. Is there a simple and less technical proof of the unified amplification lower bound? Is there a simple and natural function that satisfies the conditions of Theorem 6.1 and by the use of which a direct simple proof of the unified lower bound may be obtained?

Appendix A. Proving the strengthened Boppana's inequality. Graphs of the function $G(x)$ are given in Fig. 2.

Proof of Lemma 3.1. Define

$$\widetilde{G}(x) = \left(\frac{G(x)}{x}\right)^{\beta} = \begin{cases} \ln\frac{\gamma}{x} & \text{if } \ 0 \le x \le \frac{1}{2}, \\ \left(\frac{1}{x}-1\right)^{\beta} \ln\frac{\gamma}{1-x} & \text{if } \ \frac{1}{2} \le x \le 1. \end{cases}$$

We have to show that for every $0 < x, y \le 1$ we have

$$\widetilde{G}(x,y) = \widetilde{G}(xy) - \widetilde{G}(x) - \widetilde{G}(y) \ge 0.$$

If $0 < y \le \frac{1}{2}$, then $0 < xy \le \frac{1}{2}$ and

$$\begin{aligned} \widetilde{G}(x,y) &= \ln\frac{\gamma}{xy} - \widetilde{G}(x) - \ln\frac{\gamma}{y} \\ &= \ln\frac{1}{x} - \widetilde{G}(x) = \begin{cases} \ln\frac{1}{\gamma} & \text{if } \ 0 \le x \le \frac{1}{2}, \\ \ln\frac{1}{x} - \left(\frac{1}{x}-1\right)^{\beta} \ln\frac{\gamma}{1-x} & \text{if } \ \frac{1}{2} \le x \le 1. \end{cases} \end{aligned}$$

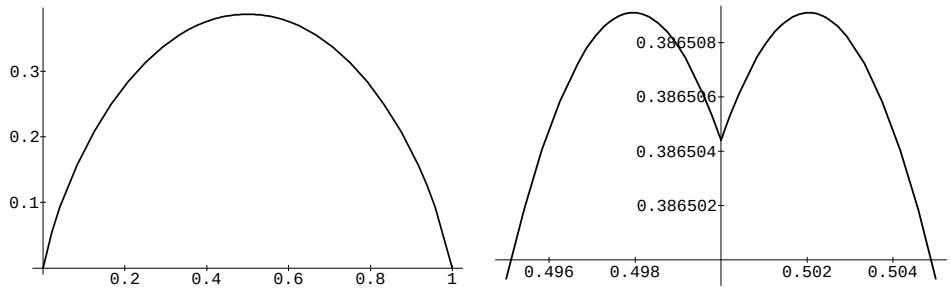


FIG. 2. *The function $G(x)$.*

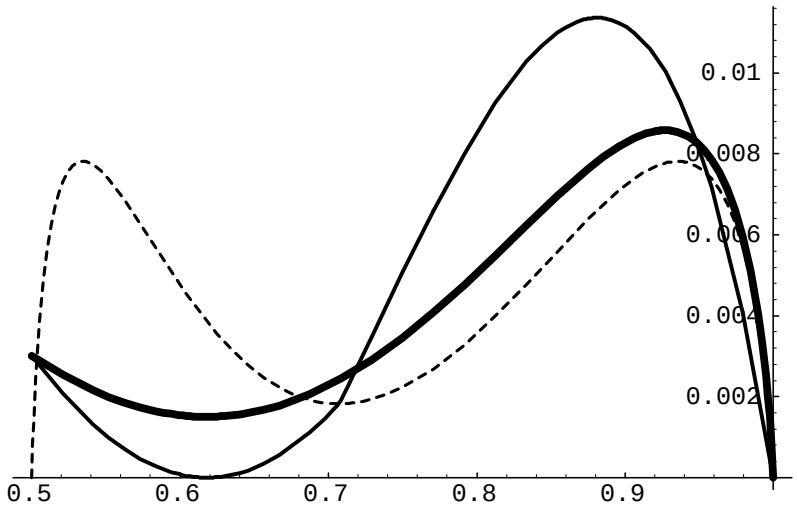


FIG. 3. *The three one-variable functions whose nonnegativity imply the strengthened Boppana's inequality.*

Since $\ln \frac{1}{\gamma} \simeq 0.003 > 0$, the inequality easily holds in the quadrant $0 < x, y \leq \frac{1}{2}$. A graph of the function $\ln \frac{1}{x} - \widetilde{G}(x) = \ln \frac{1}{x} - (\frac{1}{x} - 1)^\beta \ln \frac{\gamma}{1-x}$ for $\frac{1}{2} \leq x \leq 1$, is given by the bold line in Fig. 3. We see that the function is nonnegative for every $\frac{1}{2} \leq x \leq 1$, and this could be easily proved rigorously. (The local minimum is attained at $x = \psi$ and its value is about 0.0015.) This takes care of the rectangle $0 < y \leq \frac{1}{2}$ and by symmetry also of the rectangle $0 < x \leq \frac{1}{2}$.

We are left with the quadrant $\frac{1}{2} \leq x, y \leq 1$. Note that $\widetilde{G}(x, 1) = \widetilde{G}(1, y) = 0$, so the inequality is verified for all the points on the boundary of the quadrant. The function $\widetilde{G}(x, y)$ is nondifferentiable on the hyperbola $xy = \frac{1}{2}$ but is differentible anywhere else inside the quadrant. A graph of the function

$$\begin{aligned}\widetilde{G}\left(x, \frac{1}{2x}\right) &= \widetilde{G}\left(\frac{1}{2}\right) - \widetilde{G}(x) - \widetilde{G}\left(\frac{1}{2x}\right) \\ &= \ln 2\gamma - \left(\frac{1}{x} - 1\right)^\beta \ln \frac{\gamma}{1-x} - (2x-1)^\beta \ln \frac{\gamma}{1-\frac{1}{2x}}\end{aligned}$$

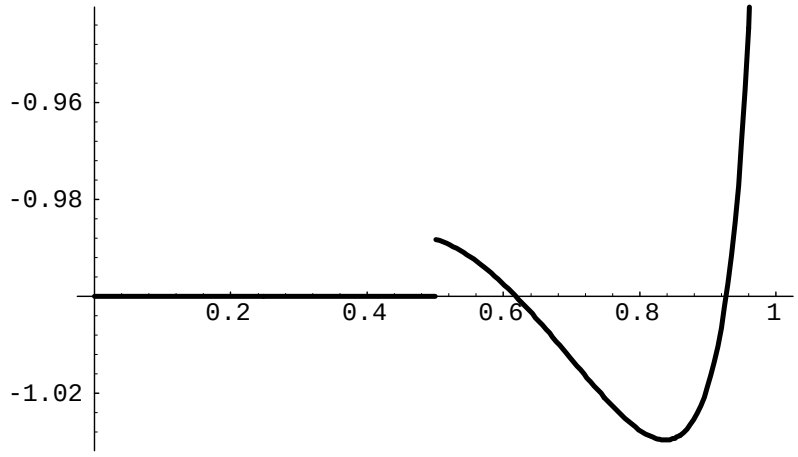


FIG. 4. *The function* $\widehat{G}(x)$. *No value* (*other than* -1) *is attained at three different* x *values.*

for $\frac{1}{2} \leq x \leq 1$ is given by the dotted line in Fig. 3. Again, it can be verified rigorously that it is nonnegative for every $\frac{1}{2} \leq x \leq 1$.

We now look for extremal points of the function $\widetilde{G}(x,y)$ inside the quadrant $\frac{1}{2} \leq x, y \leq 1$ that do not lie on the hyperbola $xy = \frac{1}{2}$. In these points, we must have

$$\frac{\partial \widetilde{G}(x,y)}{\partial x} = \frac{1}{x}\left[xy\widetilde{G}'(xy) - x\widetilde{G}'(x)\right] = 0,$$
$$\frac{\partial \widetilde{G}(x,y)}{\partial y} = \frac{1}{y}\left[xy\widetilde{G}'(xy) - y\widetilde{G}'(y)\right] = 0.$$

Define

$$\widehat{G}(x) = x\widetilde{G}'(x) = \begin{cases} -1 & \text{if } \ 0 < x < \frac{1}{2}, \\ -\left(\frac{\beta}{x}\ln\frac{\gamma}{1-x} - 1\right)\left(\frac{1}{x} - 1\right)^{\beta-1} & \text{if } \ \frac{1}{2} < x \leq 1. \end{cases}$$

If (x,y) is such an extremal point of $\widetilde{G}(x,y)$, then we must have

$$\widehat{G}(x) = \widehat{G}(y) = \widehat{G}(xy).$$

A graph of the function $\widehat{G}(x)$ is given in Fig. 4. It can be rigorously verified that the function $\widehat{G}(x)$ is unimodal for $\frac{1}{2} \leq x \leq 1$, so -1 is the only value attained by this function more than twice in the range $0 < x \leq 1$. The value -1 is attained at every $0 < x \leq \frac{1}{2}$, at $x = \psi = (\sqrt{5}-1)/2$, and at $x = \psi' \simeq 0.926812$.

Since $\psi\psi' \simeq 0.572801 > \frac{1}{2}$, we do not have $\widehat{G}(\psi\psi') = -1$. If $\widehat{G}(x) = \widehat{G}(y) = \widehat{G}(xy)$ for $\frac{1}{2} < x, y < 1$, then we must have $x = y$. All of the extremal points must therefore lie on the diagonal of the unit square. A graph of the function

$$\widetilde{G}(x,x) = \begin{cases} \ln\frac{\gamma}{x^2} - 2\left(\frac{1}{x} - 1\right)^{\beta}\ln\frac{\gamma}{1-x} & \text{if } \ \frac{1}{2} \leq x \leq \frac{\sqrt{2}}{2}, \\ \left(\frac{1}{x^2} - 1\right)^{\beta}\ln\frac{\gamma}{1-x^2} - 2\left(\frac{1}{x} - 1\right)^{\beta}\ln\frac{\gamma}{1-x} & \text{if } \ \frac{\sqrt{2}}{2} \leq x \leq 1 \end{cases}$$

is given by the solid line in Fig. 3. It can be checked that $x = y = \psi$ is indeed a local minimum and its value is 0. There is also a (global) maximum at $x = y \simeq 0.881135$. We conclude that the required inequality is therefore satisfied on the diagonal $x = y$ and consequently in the whole of the unit square. □

Appendix B. Proving the required properties of $L(x, y)$. Before proving Lemma 6.4, we establish some properties of the function $\ell(x, y)$. In what follows, we will always assume that x, y, x_1, y_1, x_2, and y_2 are in the interval $(0, 1)$, although we will not always write it explicitly.

LEMMA B.1.

1.	$\ell(x,y) \geq \ell(1-y,1-x)$	$\forall\ x \leq y \leq 1-x,$
2.	$\ell(x_1x_2, y_1y_2) \leq \ell(x_1,y_1) + \ell(x_2,y_2)$	$\forall\ x_1 \leq y_1,\ x_2 \leq y_2,$
3.	$\ell(1-y_1y_2, 1-x_1x_2) \leq \ell(1-y_1,1-x_1) + \ell(1-y_2,1-x_2)$	$\forall\ x_1 \leq y_1,\ x_2 \leq y_2.$

Relying on Lemma B.1, we can now prove Lemma 6.4.

Proof of Lemma 6.4. The function L automatically satisfies the conditions $L(x, y) = L(y, x) = L(1 - x, 1 - y)$. Therefore, we only have to prove condition 1 of Theorem 6.1. The symmetric properties of L allow us to assume that $x_1x_2 \leq y_1y_2$. (Otherwise, just change the roles of the x's and the y's.) We may also assume without loss of generality that $x_1 \leq y_1$. (If $x_1 > y_1$ and $x_2 > y_2$, then $x_1x_2 > y_1y_2$.)

As a consequence of first condition of Lemma B.1, we get that $L(x, y) = \max\{\ell(x, y), \ell(1 - y, 1 - x)\}$ for $x \leq y$. If $x_2 \leq y_2$, then

$$\begin{aligned} L(x_1x_2, y_1y_2) &= \max\{\ell(x_1x_2, y_1y_2), \ell(1-y_1y_2, 1-x_1x_2)\} \\ &\leq \max\{\ell(x_1,y_1) + \ell(x_2,y_2), \ell(1-y_1,1-x_1) + \ell(1-y_2,1-x_2)\} \\ &\leq \max\{\ell(x_1,y_1), \ell(1-y_1,1-x_1)\} + \max\{\ell(x_2,y_2) + \ell(1-y_2,1-x_2)\} \\ &= Ł(x_1,y_1) + Ł(x_2,y_2). \end{aligned}$$

In the passage from the first line to the second above, we used conditions 2 and 3 of Lemma B.1.

To finish the proof, note that if $x \leq x' \leq y' \leq y$ then $L(x', y') \leq L(x, y)$. This is easily seen since both $\ell(x, y)$ and $\ell(1 - y, 1 - x)$ are decreasing in x and inceasing in y. If $x_2 > y_2$, we have

$$\begin{aligned} L(x_1x_2, y_1y_2) &\leq L(x_1y_2, y_1y_2) \\ &\leq L(x_1,y_1) + L(y_2,y_2) = L(x_1,y_1) + 0 \\ &\leq L(x_1,y_1) + L(x_2,y_2). \end{aligned}$$

This completes the proof. □

We now turn to the proof of Lemma B.1.

Proof of Lemma B.1. 1. We have to show that $\ell(x, y) - \ell(1 - y, 1 - x) \geq 0$ for $0 < x \leq y \leq 1 - x$. If $y = x$ or $y = 1 - x$, then we have equality. Therefore, the claim will follow if we can show that $\ell(x, y) - \ell(1 - y, 1 - x)$ is decreasing in x. We therefore have to show that

$$\frac{\partial}{\partial x}[\ell(x,y) - \ell(1-y,1-x)] = -\frac{1}{x}\left(\ln\frac{\ln\frac{1}{x}}{\ln\frac{1}{y}} + 1\right) + \frac{1}{1-x}\cdot\frac{\ln\frac{1}{1-y}}{\ln\frac{1}{1-x}} \leq 0.$$

This claim is easily verified for $y = x$ (since $x \leq \frac{1}{2}$ and therefore $-\frac{1}{x} + \frac{1}{1-x} \leq 0$). It would therefore be enough to show that the above derivative is decreasing in y. We

therefore have to show that

$$\frac{\partial^2}{\partial y \partial x}[\ell(x,y) - \ell(1-y,1-x)] = -\frac{1}{xy}\cdot\frac{1}{\ln\frac{1}{y}} + \frac{1}{(1-x)(1-y)}\cdot\frac{1}{\ln\frac{1}{1-x}} \le 0$$

or, equivalently, that

$$\frac{1-x}{x}\ln\frac{1}{1-x} \ge \frac{y}{1-y}\ln\frac{1}{y}.$$

This is easily verified since the function $\frac{y}{1-y}\ln\frac{1}{y}$ is increasing in y (this follows from a further differentiation using the inequality $\ln\frac{1}{y} \ge 1-y$) and $y \le 1-x$.

2. A simple manipulation shows that

$$\begin{aligned}&\ell(x_1x_2,y_1y_2) - \ell(x_1,y_1) - \ell(x_2,y_2)\\ &\quad = -\ln\frac{1}{x_1}\cdot\left[\left(1+\frac{\ln\frac{1}{x_2}}{\ln\frac{1}{x_1}}\right)\ln\frac{1+\frac{\ln\frac{1}{y_2}}{\ln\frac{1}{y_1}}}{1+\frac{\ln\frac{1}{x_2}}{\ln\frac{1}{x_1}}} - \frac{\ln\frac{1}{x_2}}{\ln\frac{1}{x_1}}\cdot\ln\frac{\frac{\ln\frac{1}{y_2}}{\ln\frac{1}{y_1}}}{\frac{\ln\frac{1}{x_2}}{\ln\frac{1}{x_1}}}\right].\end{aligned}$$

We let

$$u = \frac{\ln\frac{1}{x_2}}{\ln\frac{1}{x_1}}, \qquad v = \frac{\ln\frac{1}{y_2}}{\ln\frac{1}{y_1}}.$$

To prove claim 2, we have to show that

$$f(u,v) = (1+u)\ln\frac{1+v}{1+u} - u\ln\frac{v}{u} \ge 0$$

for every $u,v>0$. This follows since the required inequality holds (with equality) if $u=v$, and it can be easily verified that $f(u,v)$ is increasing in u if $u \ge v$ and decreasing in u if $u \le v$.

3. We have to show that $\ell(1-y_1y_2,1-x_1x_2) - \ell(1-y_1,1-x_1) - \ell(1-y_2,1-x_2) \le 0$ for $0 < x_1 \le y_1 < 1$ and $0 < x_2 \le y_2 < 1$. We will prove this inequality in two stages:

$$\ell(1-y_1y_2,1-x_1x_2) - \ell(1-y_1,1-x_1) \le \ell(1-y_1y_2,1-y_1x_2)$$

and

$$\ell(1-y_1y_2,1-y_1x_2) \le \ell(1-y_2,1-x_2).$$

To obtain the first inequality, we show that $\ell(1-y_1y_2,1-x_1x_2) - \ell(1-y_1,1-x_1)$ is increasing in x_1. We have to show that

$$x_1\frac{\partial}{\partial x_1}[\ell(1-y_1y_2,1-x_1x_2) - \ell(1-y_1,1-x_1)] = -\frac{x_1x_2}{1-x_1x_2}\cdot\frac{\ln\frac{1}{1-y_1y_2}}{\ln\frac{1}{1-x_1x_2}} + \frac{x_1}{1-x_1}\cdot\frac{\ln\frac{1}{1-y_1}}{\ln\frac{1}{1-x_1}} \ge 0.$$

This inequality holds since

$$\ln\frac{1}{1-y_1y_2} \le \ln\frac{1}{1-y_1}, \qquad \frac{1-x_1x_2}{x_1x_2}\ln\frac{1}{1-x_1x_2} \ge \frac{1-x_1}{x_1}\ln\frac{1}{1-x_1}.$$

The second condition follows again from the fact that the function $\frac{1-x}{x}\ln\frac{1}{1-x}$ is decreasing in x.

To obtain the second inequality, we show that $\ell(1-y_1y_2, 1-y_1x_2)$ is increasing in y_1. We have to show that

$$y_1\frac{\partial}{\partial y_1}\ell(1-y_1y_2,1-y_1x_2) = \frac{y_1y_2}{1-y_1y_2}\left(\ln\frac{\ln\frac{1}{1-y_1y_2}}{\ln\frac{1}{1-y_1x_2}}+1\right) - \frac{y_1x_2}{1-y_1x_2}\cdot\frac{\ln\frac{1}{1-y_1y_2}}{\ln\frac{1}{1-y_1x_2}} \geq 0.$$

This can be done using similar methods. □

Appendix C. Proving Claims 7.4–7.8. To simplify the proofs of the claims involving $M(x,y)$, we first show how to bound this function using a slightly simpler expression. To this end, we define the function $\tilde{m}(x,y)$ by replacing the two appearances of the constant γ in the definition of $m(x,y)$ by the constant 1:

$$\tilde{m}(x,y) = \left[\left(\ln\frac{1}{x}\right)^{1/\alpha} - \left(\ln\frac{1}{y}\right)^{1/\alpha}\right]^{\alpha}.$$

On next lemma shows that $C\cdot\tilde{m}(x,y)$ can serve as a lower bound for $M(x,y)$.

LEMMA C.1. *$M(x,y) \geq C\cdot\tilde{m}(x,y)$ for every $0 < x \leq y < 1$.*

Proof. Consider the function $g(x) = x(\ln\frac{1}{x})^{1/\beta}$. It is easy to verify that $g(x) \geq G(x)$ for every $0 \leq x \leq 1$. As a consequence, we get that

$$M(x,y) = \frac{C}{\alpha^\alpha}\left|\int_x^y \frac{du}{G(u)}\right|^\alpha \geq \frac{C}{\alpha^\alpha}\left|\int_x^y \frac{du}{g(u)}\right|^\alpha = C\cdot\tilde{m}(x,y). \qquad \square$$

When y is bounded away from 1, the bound just given is tight up to a constant factor.

We now proceed with the proof of Claims 7.4–7.8.

Proof of Claim 7.4(a). Since all the functions and regions involved are symmetric with respect to the lines $y=x$ and $x+y=1$, it is enough to prove the claim for the region $(\mathcal{C}\setminus\mathcal{A})\cap\{x\leq y\leq 1-x\} = (\mathcal{C}'\setminus\mathcal{A}')\cap\{x\leq y\leq 1-x\}$. In this region, we have

$$M(x,y) \geq C\cdot\tilde{m}(x,y) = C\cdot\ln\tfrac{1}{x}\cdot\left[1-\left(\frac{\ln\frac{1}{y}}{\ln\frac{1}{x}}\right)^{1/\alpha}\right]^\alpha, \quad L(x,y) = \ell(x,y) = \ln\tfrac{1}{x}\cdot\ln\frac{\ln\frac{1}{x}}{\ln\frac{1}{y}}.$$

Let $u = \ln\frac{1}{x}/\ln\frac{1}{y}$ and note that in $(\mathcal{C}'\setminus\mathcal{A}')\cap\{x\leq y\leq 1-x\}$ we have $2\leq u\leq 6$. We therefore have

$$\frac{M(x,y)}{L(x,y)} \geq C\cdot\frac{\left[1-\left(\frac{1}{u}\right)^{1/\alpha}\right]^\alpha}{\ln u} \geq C\cdot\frac{\left[1-\left(\frac{1}{2}\right)^{1/\alpha}\right]^\alpha}{\ln 6} \gg 1. \qquad \square$$

Proof of Claim 7.4(b). Again, because of the symmetries involved, it is enough to prove the claim for $(x,y)\in\mathcal{A}''\setminus\mathcal{A}'\cap\{x,1-x\leq y\} = \{\sqrt{x}\leq y\leq x(2-x)\}$. Note that in this region $1-\psi\leq x<1$, where $\psi = (\sqrt{5}-1)/2 \simeq 0.62$. We now have

$$M(x,y) = M(1-y,1-x) \geq C\cdot\tilde{m}(1-y,1-x) = C\left[\left(\ln\tfrac{1}{1-y}\right)^{1/\alpha} - \left(\ln\tfrac{1}{1-x}\right)^{1/\alpha}\right]^\alpha.$$

Since both functions are increasing in y, we have

$$\min_{(x,y)\in\mathcal{A}''\setminus\mathcal{A}'}\frac{M(x,y)}{\ell(x,y)} \geq \min_{1-\psi\leq x<1}\frac{C\cdot\tilde{m}(1-\sqrt{x},1-x)}{\ell(x,x(2-x))}.$$

It can be checked that the minimum on the right is obtained at $x \approx 0.930677$ and its value is about $2450.5 \gg 1$. We will not give the (tedious) proof of this fact here. We will show instead that $\lim_{x\to 1} \tilde{m}(1-\sqrt{x}, 1-x)/\ell(x, x(2-x)) = +\infty$. This shows that the minimum appearing in the statement of the claim does exist and that its value is strictly positive. This would be enough to show that there exists a constant C for which the claim is valid. To that end, we note that as $x \to 1$, we have

$$\left(\left(\ln \tfrac{1}{1-\sqrt{x}}\right)^{1/\alpha} - \left(\ln \tfrac{1}{1-x}\right)^{1/\alpha}\right)^{\alpha} \approx \frac{1}{\left(\ln \frac{1}{1-x}\right)^{\alpha/\beta}},$$

$$\ln \tfrac{1}{x} \cdot \ln \frac{\ln \frac{1}{x}}{\ln \frac{1}{x(2-x)}} \approx (1-x) \ln \tfrac{1}{1-x},$$

where the sign $\approx$ here means that the quotient of the two sides tends to a positive constant as $x \to 1$. Therefore,

$$\frac{\tilde{m}(x, \sqrt{x})}{\ell(x, x(2-x))} \approx \frac{1}{(1-x)\left(\ln \frac{1}{1-x}\right)^{1+\alpha/\beta}} \to +\infty$$

as $x \to 1$, as promised. □

Proof of Claim 7.5(a). The proof is immediate from the definition of $\mathcal{A}'$. □

Proof of Claim 7.5(b). It is enough to prove the claim for the upper boundary of $\mathcal{A}''$, which is $(x, x(2-x))$ for $0 \le x \le 1$. To show that $(x^t, x^t(2-x)^t) \in \mathcal{A}''$, we have to show that $x^t(2-x)^t \le x^t(2-x^t)$ or, equivalently, that $x^t + (2-x)^t \le 2$ for every $0 \le x \le 1$ and $0 \le t \le 1$. To see this, note that equality holds for $t = 0$ and $t = 1$ and that $x^t + (2-x)^t$ is convex as a function of t since its second derivative is $(\ln x)^2 x^t + (\ln(2-x))^2 (2-x)^t \ge 0$. □

Proof of Claim 7.6(a). The proof is immediate since $\mathcal{A}'\mathcal{C}' \subseteq \mathcal{C}'\mathcal{C}' = \mathcal{C}'$. □

Proof of Claim 7.6(b). The upper boundary of $(\mathcal{A}'' \cap \{x+y \ge 1\})(\mathcal{B}'' \cap \{x+y \ge 1\})$ is all products of points on the upper boundary of $\mathcal{A}'' \cap \{x+y \ge 1\}$ and $\mathcal{B}'' \cap \{x+y \ge 1\}$. We therefore have to show that $(x_1, 1-(1-x_1)^2)(x_2, 1-(1-x_2)^4) \in \mathcal{C}''$, i.e., $[1-(1-x_1)^2][1-(1-x_2)^4] \le 1-(1-x_1x_2)^6$, whenever $x_1 \ge (1-x_1)^2$ and $x_2 \ge (1-x_2)^4$. This is easily verified. In fact, the claim remains true even if the exponent 6 is reduced to about 5.34. □

Proof of Claim 7.7(a). It is easily checked that

$$\frac{d}{dt}\ell({x_1}^t x_2, {y_1}^t y_2)\Big|_{t=0} = \ln \frac{1}{x_1}\left(\ln \frac{\ln \frac{1}{x_2}}{\ln \frac{1}{y_2}} + 1\right) - \ln \frac{1}{y_1} \cdot \frac{\ln \frac{1}{x_2}}{\ln \frac{1}{y_2}}$$

and that this expression is nonpositive iff

$$u = \frac{\ln \frac{1}{x_1}}{\ln \frac{1}{y_1}} \le \frac{\left(\frac{\ln \frac{1}{x_2}}{\ln \frac{1}{y_2}}\right)}{\ln\left(\frac{\ln \frac{1}{x_2}}{\ln \frac{1}{y_2}}\right) + 1} = \frac{v}{\ln v + 1},$$

where $v = \ln \frac{1}{x_2} / \ln \frac{1}{y_2}$. Since $(x_1, y_1) \in \mathcal{A}'$, we have $u \le 2$. Since $(x_2, y_2) \in \{x \le y\} \setminus \mathcal{C}'$, we have $v \ge 6$, and it is easy to verify that this implies $v/(\ln v + 1) \ge 2$. □

Proof of Claim 7.7(b). Simple manipulations yield

$$\frac{d}{dt}\ell(1-{y_1}^t y_2, 1-{x_1}^t x_2)\Big|_{t=0} = -\ln \frac{1}{y_1} \cdot (u-1)\left(\ln \frac{\ln u}{\ln v} + 1\right) + \ln \frac{1}{x_1} \cdot (v-1)\frac{\ln u}{\ln v},$$

where $v = \frac{1}{1-x_2}$ and $u = \frac{1}{1-y_2}$. Also let $w = (u-1)(\ln(\ln u/\ln v)+1)$. By the definition of $\mathcal{B}''$, we have $u \geq v^4$. Note that $x_2 \geq 0$ and $y_2 \geq \frac{1}{2}$ (since $(x_2, y_2) \in \{x+y \geq 1, x \leq y\}$) and therefore $v \geq 1$ and $u \geq 2$.

It can easily be checked that the conditions $u \geq 2, v^4$ and $v \geq 1$ imply that $(u-1)^{1/3}/\ln u > \frac{1}{2} \cdot \frac{v-1}{\ln v}$. (Note that for v large enough this is obvious. The constant $\frac{1}{2}$ was added to make the inequality valid for smaller values of v.) In particular,

$$(v-1)\frac{\ln u}{\ln v} < 2(u-1)^{1/3} \leq \frac{2}{(\ln 4+1)^{1/3}} \cdot w^{1/3} < 1.5 w^{1/3}$$

(since $w \geq (u-1)(\ln 4+1)$ and $2/(\ln 4+1)^{1/3} \approx 1.49666$) and therefore

$$\frac{d\ell}{dt}\Big|_{t=0} \leq -\ln\frac{1}{y_1} \cdot w + 1.5 \ln\frac{1}{x_1} \cdot w^{1/3}.$$

Note that $w \geq \ln 4 + 1$, so if

$$\frac{1.5 \ln \frac{1}{x_1}}{\ln \frac{1}{y_1}} \leq (\ln 4+1)^{2/3} \leq w^{2/3},$$

then $\frac{d\ell}{dt}|_{t=0}. \leq 0$. This proves the lower part of the claim since $(\ln 4+1)^{2/3}/1.5 \approx 1.19049 > 1.15$.

To get the upper part of the claim, we note that the expression $-\ln\frac{1}{y_1} \cdot w + 1.5 \ln\frac{1}{x_1} \cdot w^{1/3}$, considered as a function of w, is maximized when $w = (\ln\frac{1}{x_1}/2\ln\frac{1}{y_1})^{3/2}$. (This follows easily by differentiation.) Plugging this into the expression, we get that

$$\frac{d\ell}{dt}\Big|_{t=0} \leq \frac{\left(\ln\frac{1}{x_1}\right)^{3/2}}{\left(2\ln\frac{1}{y_1}\right)^{1/2}}.$$

Next, it is easily verified that if $(x_1, y_1) \in \mathcal{A}'' \cap \{x, 1-x \leq y\}$, then $\ln\frac{1}{y_1} > \frac{1}{2}(\ln\frac{1}{x_1})^2$. (In fact, the minimum of $\ln\frac{1}{y_1}/(\ln\frac{1}{x_1})^2$ over this region is attained at the point $(x_1, y_1) = (1-\psi, \psi)$, and its value is about 0.519.) We thus easily obtain

$$\frac{d\ell}{dt}\Big|_{t=0} \leq \frac{\left(\ln\frac{1}{x_1}\right)^{3/2}}{\left(2\ln\frac{1}{y_1}\right)^{1/2}} \leq \left(2\ln\frac{1}{y_1}\right)^{1/4}$$

as required. □

Proof of Claim 7.8. Since $M(x,y)$ is increasing in y, it is enough to prove that $M(x, x^{1/1.15}) \geq (\frac{2}{1.15}\ln\frac{1}{x})^{1/4}$ for every x such that $x + x^{1/1.15} \geq 1$, i.e., $x \geq 0.47579\ldots$. We will content ourselves by noting that $\lim_{x\to 1} M(x, x^{1/1.15})/(\ln\frac{1}{x})^{1/4} = +\infty$, which shows that the claim is valid for a sufficiently large choice of C. This limit can be obtained using essentially the same asymptotics as those used in the proof of Claim 7.4(b). Using some more uninspiring work it can be verified that the minimum of $\tilde{m}(1-x^{1/1.15}, 1-x)/(\frac{2}{1.15}\ln\frac{1}{x})^{1/4}$ is attained near $x = 0.999877$ and its value is about 1.84×10^{-6}. □

We note that the huge value of C was required only in the proof of Claim 7.8. For Claim 7.4, a much more modest value would have been enough. Also, no attempt to obtain the optimal value of C was made. We are sure that using a tighter yet even more tedious analysis, a much smaller value of C can be shown to suffice.

Acknowledgments. The authors would like to thank Noga Alon for many helpful discussions. The authors would also like to thank the two anonymous referees for their numerous comments.

REFERENCES

[1] L. Adleman, *Two theorems on random polynomial time*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1978, pp. 75–83.

[2] M. Ajtai and M. Ben-Or, *A theorem on probabilistic constant depth computations*, in Proc. 16th Annual ACM Symposium on Theory of Computing, ACM, New York, 1984, pp. 471–474.

[3] C. Bennett and J. Gill, *Relative to a random oracle A, $P^A \neq NP^A \neq co\text{-}NP^A$ with probability* 1, SIAM J. Comput., 10 (1981), pp. 96–113.

[4] R. Boppana, *Amplification of probabilistic Boolean formulas*, in Advances in Computer Research, Vol. 5: Randomness and Computation, JAI Press, Greenwich, CI, 1989, pp. 27–45.

[5] M. Dubiner and U. Zwick, *Amplification and percolation*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 258–267.

[6] M. Dubiner and U. Zwick, *Contact networks, amplification and percolation*, in preparation.

[7] M. Dubiner and U. Zwick, *How do read-once formulae shrink?*, Combin. Probab. Comput., 3 (1994), pp. 455–469.

[8] Q. Gu and A. Maruoka, *Amplification of bounded depth monotone read-once Boolean formulas*, SIAM J. Comput., 20 (1991), pp. 41–55.

[9] A. Gupta and S. Mahajan, *Using amplification to compute majority with majority*, manuscript, 1992.

[10] J. Håstad, A. Razborov, and A. Yao, *On the shrinkage exponent for read-once formulae*, Theoret. Comput. Sci., 141 (1995), pp. 269–282.

[11] V. M. Khrapchenko, *Complexity of the realization of a linear function in the class of π-circuits*, Math. Notes Acad. Sci. USSR, 9 (1971), pp. 21–23.

[12] V. M. Khrapchenko, *A method of determining lower bounds for the complexity of π-schemes*, Math. Notes Acad. Sci. USSR, 10 (1971), pp. 474–479.

[13] O. Lupanov, *On computing symmetric functions of the propositional calculus by switching networks*, Problemy Kibernet., 15 (1965), pp. 85–100 (in Russian).

[14] E. Moore and C. Shannon, *Reliable circuits using less reliable relays*, J. Franklin Inst., 262 (1956), pp. 191–208 and 281–297.

[15] M. Paterson, N. Pippenger, and U. Zwick, *Faster circuits and shorter formulae for multiple addition, multiplication and symmetric Boolean functions*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 642–650.

[16] M. Paterson, N. Pippenger, and U. Zwick, *Optimal carry save networks*, in Boolean Function Complexity, M. Paterson, ed., London Mathematical Society Lecture Note Series, Vol. 169, Cambridge University Press, Cambridge, UK, 1992, pp. 174–201.

[17] M. Paterson and U. Zwick, *Shrinkage of de Morgan formulae under restriction*, Random Structures Algorithms, 4 (1993), pp. 135–150.

[18] J. Radhakrishnan and K. Subrahmanyam, *Directed monotone contact networks for threshold functions*, Inform. Process. Lett., 50 (1994), pp. 199–203.

[19] L. Valiant, *Short monotone formulae for the majority function*, J. Algorithms, 5 (1984), pp. 363–366.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 39–58, February 1997

003

MINIMAL ASCENDING AND DESCENDING TREE AUTOMATA*

MAURICE NIVAT† AND ANDREAS PODELSKI‡

Abstract. We propose a generalization of the notion "deterministic" to "l-r-deterministic" for descending tree automata (also called root-to-frontier). The corresponding subclass of recognizable tree languages is characterized by a structural property that we name "homogeneous." Given a descending tree automaton recognizing a homogeneous tree language, it can be left-to-right (l-r) determinized and then minimized. The obtained minimal l-r-deterministic tree automaton is characterized algebraically. We exhibit a formal correspondence between the two evaluation modes on trees (ascending and descending) and the two on words (right-to-left and left-to-right). This is possible by embedding trees into the free monoid of pointed trees. We obtain a unified view of the theories of minimization of deterministic ascending and l-r-deterministic descending tree automata.

Key words. tree automata, minimization, Nerode congruence

AMS subject classifications. 68Q68, 20M35

PII. S0097539789164078

1. Introduction and synopsis.

Automata on finite trees come up in various areas of computer science (compiler generation [13, 16, 25], type theory [19, 30, 28], constraint solving [3, 14, 15, 24, 31], rewriting [6], etc.) as a generic decision tool for problems over finite trees. Historically, their first application was a decision problem over strings, namely for monadic second-order logic formulas interpreted on the finite tree as a finite subset of the free monoid [7, 33].

The theory of tree automata and recognizable sets of trees has been introduced as a (seemingly) natural extension from the string case. The extension consists of going from unary to n-ary algebras, in the algebraic view of trees, or from one to multiple successors, in the logical view of the tree as a model. Recent works on structural properties of sets of finite trees [34, 17, 18, 32, 27], however, have exhibited some difficulties in carrying this extension further. The essence of the difference between strings and trees has not been revealed yet.

One important difference comes from the well-known fact that one cannot determinize every descending tree automaton.[1] In the string case, the direction of evaluation is usually assumed to be left-to-right, but this does not affect the family of recognizable languages. For the extension from the word to the tree case, where a word fga is considered a tree $f(g(a))$, it is the right-to-left evaluation which corresponds to the ascending[2] tree automaton. An automaton of this kind can always be transformed into a deterministic one which recognizes the same tree language. A finite deterministic ascending tree automaton is an algebraic notion (the quotient of the

*Received by the editors April 3, 1989; accepted for publication (in revised form) April 13, 1995. This research was supported in part by the CNRS in the PROCOPE program in which the authors participated in cooperation with L. Priese, University of Koblenz, Koblenz, Germany.
http://www.siam.org/journals/sicomp/26-1/16407.html

†LITP, Université Paris 7, 2 Place Jussieu, F-75251 Paris cedex 05, France (maurice.nivat@litp.ibp.fr).

‡Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany (podelski@mpi-sb.mpg.de).

[1] In our graphical representation of a tree, its root is on the top. Hence *descending* or *top-down* correspond to the more precise, but somewhat lengthy, identifier *root-to-frontier* which is used in [12].

[2] And *ascending* or *bottom-up* correspond to *frontier-to-root*.

tree algebra by a congruence with finitely many classes), and we can apply algebraic concepts in order to implement minimization.

Descending tree automata have, to our knowledge, not yet been investigated using algebraic concepts and methods. They are important, however, since they correspond naturally to tree grammars and rewriting systems. Certain type systems, for example, employ a class of tree grammars which correspond to deterministic descending tree automata [14, 36, 19]. Also, this lack of algebraic concepts [9, 26] is one of the reasons why the extension of the decision problem of [7, 33] to the infinite tree is so involved [10, 21] (obviously, the ascending evaluation is not possible here).

The minimization of the *deterministic* descending tree automata is done in [11] with a nonalgebraic state-reduction method. The class of corresponding tree languages is characterized as a proper subclass of the recognizable ones by the structural property called path closed in [35] and tuple distributive in [19]. This class is investigated in a logic framework in [34].

Given the lack of a suitable algebra framework for descending tree automata, we propose a monoid framework. Namely, we consider tree automata as automata on strings, i.e., elements of a suitable free monoid. Then ascending and descending correspond exactly to right-to-left and left-to-right, respectively.

In section 2, we introduce this monoid. Its elements are called pointed trees. It is freely generated by (infinitely many) basic pointed trees. For every set of trees L one can construct a set $\widehat{L}$ of pointed trees. One principal idea in this paper is to apply the formal framework of free monoids and the one-to-one correspondence between the sets L and $\widehat{L}$ on descending tree automata.

Pointed trees are related to special trees in [20, 34] or to terms over the free Σ algebra with exactly one occurrence of one variable. A free monoid structure, however, has not been exhibited. Also, the construction of the set $\widehat{L}$ from L is new.

In section 3, the representation of the minimization of deterministic finite *ascending* tree automata is novel in that the Nerode equivalence is defined on the free monoid of pointed trees. This allows us to extend the word case in the very direct way to the tree case. In fact, the only extension necessary is the one from a finitely to an infinitely generated free monoid.

By constructing the response function $\hat{\lambda}$ from the transition function λ, we assign essentially a word automaton to an ascending tree automaton, since the arguments of $\hat{\lambda}$ are a state and an element of the free monoid. The evaluation of that word automaton is a right-to-left one.

We recall that the standard way of defining the Nerode equivalence consists of using algebra congruences. There, the extension from the word to the tree case is the one from unary to binary function symbols for the signature of the algebra. This implies that the Nerode equivalence is viewed as a congruence on an (in the word case, unary) algebra. In contrast, we view the Nerode equivalence as a semicongruence (or right congruence) on the free monoid, in the word as well as the tree case.

It is a well-known result (attributed to Hall in [2]) that, in the algebra framework, translations can be used for defining the greatest congruence $\sim_L$ saturating a given subset L of the algebra. This has been exploited for tree automata in [12]. We have pushed the result one step further by defining a relation $\sim_{\hat{L}}$ on translations themselves and by exhibiting a free monoid structure on the set of translations in the free algebra

(which correspond to pointed trees). To summarize, section 3 uses the concept of *left-invariant* monoid semicongruences in order to define the minimal deterministic ascending tree automaton.

In section 4, we construct the response function $\hat{\lambda}$ from the transition function λ of a *descending* tree automaton. By this means, we assign it a word automaton with left-to-right evaluation. Using the response function, the characterization of the acceptance of a tree by a descending tree automaton becomes now concise and natural.

In section 5, we use the response function $\hat{\lambda}$ to introduce a new version of determinism for descending tree automata, defining it by the condition $card(\hat{\lambda}(\ldots)) = 1$. Generally, in automata theory it does not matter whether one requires the transition function λ or the response function $\hat{\lambda}$ to be deterministic: the two conditions are equivalent. Here, however, we obtain the notion of l-r determinism which is strictly more general than the standard notion of determinism for descending tree automata. That notion is based directly on the transition function, being defined by $card(\lambda(\ldots)) = 1$.

The class of tree languages which are recognized by l-r-deterministic finite descending tree automata strictly includes the class of tree languages which are recognized by deterministic ones but is itself included in the class of all recognizable tree languages. It can be characterized by a structural property that we will introduce. We call it homogeneous.

A theory of minimization of descending tree automata can seemingly not be based on the concept of algebra congruences. Given the presentation of minimization in section 3, we will use the fact that l-r-deterministic descending tree automata can also be embedded in the monoid framework. Namely, since they read the "words" of the free monoid $\Sigma^{(\#)}$ deterministically in left-to-right fashion, they induce a *right-invariant* monoid semicongruence on $\Sigma^{(\#)}$. It comes as no surprise now that we will define the Nerode equivalence relation that yields the states of the minimal descending tree automaton of a homogeneous tree language in the same way as is done for word languages [9]: we just have to apply this method on the language $\widehat{L}$ of words of $\Sigma^{(\#)}$ that corresponds to a tree language.

In section 6, we show the existence of a minimal l-r-deterministic finite descending tree automaton $\mathcal{A}(L)$ for every homogeneous recognizable tree language L. This automaton is minimal in the algebraic sense (using automata morphisms) and thus also in the number of states. It is reduced and unique up to isomorphism. Its states are the equivalence classes of the new *descending Nerode equivalence* of L which we introduce.

These results applied on path-closed tree languages yield (minimal) deterministic descending tree automata. For an arbitrary tree language L, we still can define $\mathcal{A}(L)$, which characterizes through its finiteness the recognizability of L.

For showing this last result, we just have to connect the two theories of minimal ascending and descending tree automata of a tree language L (not necessarily homogeneous and not necessarily recognizable). If and only if L is recognizable, then the language $\widehat{L}$ is a recognizable subset of the free monoid $\Sigma^{(\#)}$ and so is its reverse (the set of mirror images of its elements). Hence the minimal automaton recognizing $\widehat{L}$ with right-to-left evaluation is finite and so is the one with left-to-right evaluation. The first one can be transformed into an ascending tree automaton which recognizes L. The second one can be transformed into a descending tree automaton which recognizes L if L is homogeneous. In both cases, one obtains hereby the minimal automaton in that class. In general, the descending tree automaton obtained by the transformation recognizes the smallest homogeneous tree language containing L.

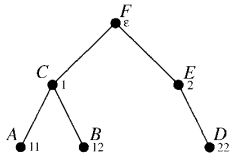


FIG. 1. *The tree* t_1.

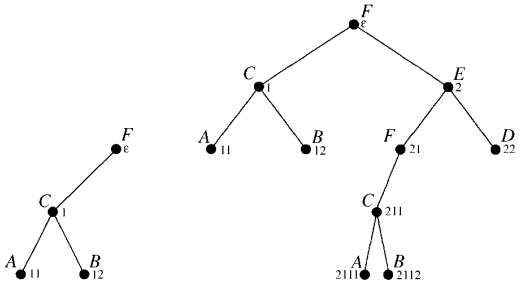


FIG. 2. *The trees* t_2 *and* $t_3 = t_1 + 21t_2$.

2. Tree monoids. For ease of notation, we only consider the case of binary trees on an unranked alphabet Σ.[3] This is the most standard case. One can give encodings of n-ary into binary trees as in [5, 8]. Also, the results of this article are straightforwardly extended to the general case. A (binary) *tree domain* is a set D of finite strings over 1 and 2 (which form the free monoid generated by $\{1, 2\}$, hence $D \subseteq \{1, 2\}^*$) such that for every element $fg \in D$ also $f \in D$. A subset of a free monoid with this closure property is called *left factorial* (cf. [1]).

A (finite, binary-ordered, labeled) *tree* on the alphabet Σ is the pair $t = (\mathrm{Supp}(t), \widehat{t}\,)$ formed by its support, which is a finite tree domain, $\mathrm{Supp}(t) \subset_{\text{finite}} \{1, 2\}^*$, and its labeling function, which goes from its support into the alphabet, $\widehat{t} : \mathrm{Supp}(t) \to \Sigma$.

We call $\Sigma^{\#}$ the set of trees on Σ. The elements of $\mathrm{Supp}(t)$ are the nodes of the tree t. Hence a node is identified with the string which corresponds to the path leading to it, where 1 and 2 stand for a step to the right and to the left, respectively. The tree t_1 given by $\mathrm{Supp}(t_1) = \{\varepsilon, 1, 11, 12, 2, 22\}$ and $\widehat{t_1} = \{(\varepsilon, F), (1, C), (11, A), (12, B), (2, E), (22, D)\}$ is presented in Figure 1.

We call the *border* of t the set $\mathcal{B}(t) = \{fi \mid f \in \mathrm{Supp}(t), i \in \{1, 2\},\ fi \notin \mathrm{Supp}(t)\}$ of all nodes "just outside the tree." Thus we might consider the border nodes as the unlabeled leaves of the tree. The border of the tree depicted in Figure 1 is the set $\mathcal{B}(t_1) = \{111, 112, 121, 122, 21, 221, 222\}$. The height of a tree t is the maximal length of a string in $\mathcal{B}(t)$.

The *empty tree*, denoted Ω, is the tree whose support is empty, $\mathrm{Supp}(\Omega) = \emptyset$. Its border is conveniently defined as $\mathcal{B}(\Omega) = \{\varepsilon\}$. A *punctual tree* is a tree of the form $t = (\{\varepsilon\}, \widehat{t}\,)$ consisting of exactly one node. If this node is labeled by $a \in \Sigma$ (i.e., $\widehat{t}(\varepsilon) = a$), then t is simply denoted a. By this identification, $\Sigma \subset \Sigma^{\#}$. The border of a punctual tree a is $\mathcal{B}(a) = \{1, 2\}$.

Given two trees t, $t' \in \Sigma^{\#}$ and a border node f of the tree t, $f \in \mathcal{B}(t)$, the *sum* of t and t' at f is the result of attaching t' to t at f [22]. Formally, it is the tree

[3] From now on, we assume the alphabet Σ given fixed. Hence we may simply use trees, automata, etc., for trees on Σ, Σ-automata, etc.

$s = t + ft'$, where

$$\mathrm{Supp}(s) = \mathrm{Supp}(t) \cup \{ff' | f' \in \mathrm{Supp}(t')\},$$
$$\widehat{s}(g) = \begin{cases} \widehat{t}(g) & \text{if } g \in \mathrm{Supp}(t), \\ \widehat{t'}(f') & \text{if } g = ff', f' \in \mathrm{Supp}(t'). \end{cases}$$

For example, if t_2 is the tree to the left in Figure 2, then $t_3 = t_1 + 21t_2$ is the tree to the right. Clearly, every tree $t \in \Sigma^{\#}$ can be represented as the sum of punctual trees $a \in \Sigma \subset \Sigma^{\#}$. This motivates the use of the symbol # for the set of trees, in reminiscence of the star operator $*$ for the set of strings. Our formalism reflects a geometric view of trees and their composition, as opposed to the standard one of substitution of terms.

We consider a tree t together with a fixed node f from its border. We shall call the pair (t, f) a *pointed tree* and f its pointed border node [23, 24]. The set of all pointed trees is denoted by $\Sigma^{(\#)}$.

If we attach to the tree t with pointed border node f the tree t' with pointed border node f', then the path of t leading to f is continued with the path of t' which leads to f'. The resulting path leads to ff', which is in the border of $t + ft'$. It thus seems natural to take as the result of this composition the tree $t + ft'$ with the pointed border node ff'. We thus obtain the *tree monoid*

$$\Sigma^{(\#)} = \{(t, f) \,|\, t \in \Sigma^{\#}, f \in \mathcal{B}(t)\},$$

where the composition of the pointed trees (t, f) and $t', f')$ is given by

$$(t, f)(t', f') = (t + ft', ff').$$

Clearly, the composition is associative and has (Ω, ε) as its neutral element. This operation can also be defined through the wreath product of two suitable monoids [29].

The following set of *basic* pointed trees will be of particular interest in the following.

$$\Gamma = \{(t, f) \in \Sigma^{(\#)} \,|\, f = 1 \ \text{ or } \ f = 2\}.$$

Every element of $(t, f) \in \Gamma$ is either of the form $(a + 2t', 1)$ or of the form $(a + 1t', 2)$ for some $a \in \Sigma$ and $t' \in \Sigma^{\#}$. In this notation, we use again that $\Sigma \subset \Sigma^{\#}$.

The set Γ consists of all pointed trees, different from the empty tree, which can no longer be decomposed into pointed trees other than themselves and the empty tree. This implies that $\Sigma^{(\#)}$ is not finitely generated. The tree monoid $\Sigma^{(\#)}$ is, however, a *free monoid*. Every pointed tree has a unique representation as a product of elements of Γ.

Given a set of trees $L \subseteq \Sigma^{\#}$, we define the set $\widehat{L} \subseteq \Sigma^{(\#)}$ of all its trees with a pointed border node

$$\widehat{L} = \{(t, f) \,|\, t \in L, f \in \mathcal{B}(t)\}.$$

Clearly, this yields a one-to-one mapping from sets of trees to sets of pointed trees.

3. Ascending tree automata. A [*finite*] (*nondeterministic*) *ascending tree automaton* $A = (Q, \lambda, q_0, Q_{\text{fin}})$ is given by its [finite] set of states Q, the initial state $q_0 \in Q$, the set of final states $Q_{\text{fin}} \subseteq Q$, and its (nondeterministic) transition function

$$\lambda : Q \times Q \times \Sigma \to \mathcal{P}(Q).$$

A *computation* of A on a tree $t \in \Sigma^{\#}$ starts by associating all border nodes (the "unlabeled leaves") of t with the initial state. It proceeds by associating a node $f \in \text{Supp}(t)$ with a state $q \in \lambda(q_1, q_2, a)$ if the node f is labeled with $a \in \Sigma$ and its left and right descendants $f1$ and $f2$ are already associated with states q_1 to the left and q_2 to the right (here $f1$ and $f2$ might be border nodes). A difficulty for the intuitive understanding of tree automata might be that this description of the evaluation of a tree does not yield an explicit algorithm. In fact, one can imagine different evaluation orders; of course, a state can be associated with a node only after this has been done for its two direct descendants, and so on.

Formally, a computation on a tree t is a tree S which has the same nodes as t plus its border nodes and is labeled, not over Σ but over Q; i.e., $S \in Q^{\#}$ and $\text{Supp}(S) = \text{Supp}(t) \cup \mathcal{B}(t)$ and

$$\begin{aligned} \forall f \in \mathcal{B}(t) : &\quad \widehat{S}(f) = q_0, \\ \forall f \in \text{Supp}(t) : &\quad \widehat{S}(f) \in \lambda(\widehat{S}(f1), \widehat{S}(f2), \widehat{t}(f)), \end{aligned} \tag{1}$$

A tree t is accepted by A if the result of the computation of A on t, the state that it associates with the root of t, is a final state. A tree language $L \subseteq \Sigma^{\#}$ is recognized by A, $L = \mathcal{L}(A)$ if L is the set of all trees t which are accepted by A. It is called *recognizable* if there exists a *finite* tree automaton that recognizes it.

We now consider the result of a computation of A on t which starts at some given border node $f \in \mathcal{B}(t)$ with a given state $q \in Q$ but at all other border nodes of t with the initial state. Formally, we define the *response function* $\hat{\lambda} : \Sigma^{(\#)} \times Q \to \mathcal{P}(Q)$, where $\hat{\lambda}(t, f, q)$ is the set of all states $\widehat{S}(\varepsilon)$, where S is a computation of A on (t, f) starting with q; that is $\text{Supp}(S) = \text{Supp}(t) \cup \mathcal{B}(t)$ and

$$\begin{aligned} &\widehat{S}(f) = q, \\ \forall g \in \mathcal{B}(t) - \{f\} : &\quad \widehat{S}(g) = q_0, \\ \forall g \in \text{Supp}(t) : &\quad \widehat{S}(g) \in \lambda(\widehat{S}(g1), \widehat{S}(g2), \widehat{t}(g)). \end{aligned}$$

Using this function we get a simple characterization of the fact that $L \subseteq \Sigma^{\#}$ is recognized by A, namely via the correspondence $L \leftrightarrow \widehat{L}$.

$$L = L(A) \quad \text{iff } \widehat{L} = \{(t, f) \mid \widehat{\lambda}(t, f, q_0) \in Q_{\text{fin}}\}.$$

We remark that $\widehat{\lambda}(t, f, q_0) = \widehat{\lambda}(t, f', q_0)$ for all f, $f' \in \mathcal{B}(t)$. This invariance will not hold for the response function of descending tree automata which we will define in the next section.

If we set $\widehat{\lambda}(t, f, q_0) = \tilde{\lambda}(t)$, we obtain a function $\tilde{\lambda} : \Sigma^{\#} \to Q$ which characterizes the set of recognized trees as $L(A) = \tilde{\lambda}^{-1}(Q_{\text{fin}})$. Both $\widehat{\lambda}$ and $\tilde{\lambda}$ can also be defined recursively.

We define the ascending tree automaton A to be *deterministic* if its response function $\widehat{\lambda}$ is deterministic, i.e., a function into the set of states Q. (We view Q as

a subset of $\mathcal{P}(Q)$, identifying a singleton with its element.) This coincides with the standard condition that λ be deterministic.

We recall that an equivalence relation $\sim$, here on the tree monoid $\Sigma^{(\#)}$, is a *left-invariant semicongruence* if $(t_1, f_1) \sim (t_2, f_2)$ implies $(t, f)(t_1, f_1) \sim (t, f)(t_2, f_2)$ for all $(t, f) \in \Sigma^{(\#)}$. Also, $\sim$ *saturates* the subset $\widehat{L} \subseteq \Sigma^{(\#)}$ of the tree monoid if $\widehat{L}$ is the union of some of its equivalence classes, i.e., $(t_1, f_1) \sim (t_2, f_2)$ implies $(t_1, f_1) \in \widehat{L}$ iff $(t_2, f_2) \in \widehat{L}$. Finally, a relation $\sim_1$ is *coarser* than another one $\sim_2$ if $x \sim_2 y$ implies $x \sim_1 y$; i.e., the relation $\sim_1$ as a set is included in $\sim_2$.

DEFINITION 1. *The Nerode equivalence of a tree language $L \subseteq \Sigma^{\#}$, denoted $\sim_L$, is the coarsest left-invariant semicongruence on the tree monoid $\Sigma^{(\#)}$ which saturates $\widehat{L}$.*

Another way to characterize $\sim_L$ is to define that $(t_1, f_1) \sim_L (t_2, f_2)$ iff

$$\forall (t, f) \in \Sigma^{(\#)} : (t, f)(t_1, f_1) \in \widehat{L} \quad \text{iff } (t, f)(t_2, f_2) \in \widehat{L}.$$

We say that two pointed trees (t, f) and (t', f') are *border equivalent* iff $t = t'$. We observe that border equivalence is coarser than the Nerode equivalence of any tree language, i.e., always $(t, f) \sim_L (t, f')$. This will not hold for the descending Nerode equivalence that we will define in section 5.

THEOREM 1. *The tree language $L \subseteq \Sigma^{\#}$ is recognizable iff its Nerode equivalence $\sim_L$ on the tree monoid has a finite index, which means that the set*

$$\Sigma^{(\#)}_{/\sim_L} = \{[(t, f)]_{\sim_L} \mid (t, f) \in \Sigma^{(\#)}\}$$

is finite. This set is the set of states of the minimal deterministic finite ascending tree automaton of L (which exists uniquely for every recognizable tree language).

Proof. Given that $card(\Sigma^{(\#)}_{/\sim_L})$ is finite, we can define a deterministic finite ascending tree automaton $A^L = (Q^L, \lambda^L, q_0^L, Q_{\text{fin}}^L)$ by $Q^L = \Sigma^{(\#)}_{/\sim_L}$, $q_0^L = [(\Omega, \varepsilon)]_{\sim_L}$, $Q_{\text{fin}}^L = \{[(t, f)]_{\sim_L} \mid (t, f) \in \widehat{L}\}$, and $\lambda^L([(t_1, f_1)]_{\sim_L}, [(t_2, f_2)]_{\sim_L}, a) = [(a + 1t_1 + 2t_2, 1f_1)]_{\sim_L}$.

Using the property mentioned above, one proves that the definition of λ^L is well founded and that $\widehat{\lambda^L}(t, f, q_0) = [(t, f)]_{\sim_L}$. This shows one direction, since a tree t is accepted by $\mathcal{A}^L$ if $\widehat{\lambda^L}(t, f, q_0) \in Q_{\text{fin}}^L$.

Conversely, consider a deterministic finite ascending tree automaton $A = (Q, \lambda, q_0, Q_{\text{fin}})$ which recognizes the tree language $L \subseteq \Sigma^{\#}$. We can attach to A the relation $\sim_A$ on $\Sigma^{(\#)}$ defined by $(t_1, f_1) \sim_A (t_2, f_2)$ iff $\widehat{\lambda}(t_1, f_1, q_0) = \widehat{\lambda}(t_2, f_2, q_0)$.

The following property proves that this equivalence relation is left invariant. It says that λ respects the composition of the tree monoid:

$$\widehat{\lambda}(t, f, \widehat{\lambda}(t', f', q)) = \widehat{\lambda}((t, f)(t', f'), q).$$

This equation implies that $\widehat{\lambda}((t, f)(t_1, f_1), q_0) = \widehat{\lambda}((t, f)(t_2, f_2), q_0)$. The left invariance of $\sim_A$ implies that $\sim_L$ is coarser than $\sim_A$, which means that $card(\Sigma^{(\#)}_{/\sim_L}) \leq card(\Sigma^{(\#)}_{/\sim_A})$. However, $card(\Sigma^{(\#)}_{/\sim_A}) \leq card(Q)$, and this again implies that the Nerode equivalence $\sim_L$ has finite index and that $\mathcal{A}^L$ has the minimal number of states among all deterministic finite ascending tree automata recognizing L. □

A stronger notion of minimality than the one based on the number of states is the following algebraic one. An automaton A is called *minimal* in a given class of automata iff every equivalent automaton A' in the class can be mapped onto A by

a corresponding automata morphism. Here a morphism π from $A = (Q, \lambda, q_0, Q_{\text{fin}})$ onto $A' = (Q', \lambda', q'_0, Q'_{\text{fin}})$ is a surjective mapping from Q onto Q' which satisfies that $\pi(\lambda(q_1, q_2, a)) = \lambda'(\pi(q_1), \pi(q_2), a)$, $\pi(q_0) = q'_0$, and $\pi(Q_{\text{fin}}) = Q'_{\text{fin}}$. It is not difficult to show that $\mathcal{A}^L$ is minimal in the algebraic sense.[4]

COROLLARY 1. *A tree language L is recognizable iff $\widehat{L}$ is saturated by a left-invariant equivalence relation $\sim$ on the tree monoid $\Sigma^{(\#)}$ which has finite index.* □

4. Descending tree automata. A [*finite*] (*nondeterministic*) *descending tree automaton* $A = (Q, \lambda, q_0, Q_{\text{fin}})$ is given by its [finite] set of states Q, the initial state q_0, and the subset Q_{fin} of Q of final states and the (nondeterministic) transition function

$$\lambda : Q \times \Sigma \to \mathcal{P}(Q \times Q).$$

We will carry over the notions from the case of ascending to the case of descending tree automata and sometimes use the same notation.

A *computation* of A on a tree t now starts at its root by associating the initial state with this node. At each node with, say, label, $A \in \Sigma$ and already associated state $q \in Q$, the computation proceeds by choosing nondeterministically a pair of successor states $(q_1, q_2) \in \lambda(q, a)$ and associating q_1 and q_2 with the left respective right son ("descendant") of this node.

Formally, we define a computation on t to be a tree $S \in Q^{\#}$ with support $\text{Supp}(S) = \text{Supp}(t) \cup \mathcal{B}(t)$ which satisfies the following conditions (instead of (1) for the ascending tree automaton).

$$\widehat{S}(\varepsilon) = q_0,$$
$$\forall f \in \text{Supp}(t): \quad (\widehat{S}(f1), \widehat{S}(f_2)) \in \lambda(\widehat{S}(f), \widehat{t}(f)).$$

The computation of A on t is called *successful* if it associates all border nodes with a final state, i.e., $\widehat{S}(g) \in Q_{\text{fin}}$ for all $g \in \mathcal{B}(t)$. A tree t is accepted by A if there exists a successful computation of A on t. The tree language L recognized by A, $L - \mathcal{L}(A)$ is the set of all accepted trees.

It is not evident how one can define a function that describes the evaluation of a tree by a descending tree automaton. We can, however, describe the evaluation of a pointed tree by defining the *response function* $\widehat{\lambda} : Q \times \Sigma^{(\#)} \to \mathcal{P}(Q)$ of a descending tree automaton as follows. (Note that we changed the order of the arguments of the response function of the ascending tree automaton.)

$$\begin{aligned}\widehat{\lambda}(q, t, f) = \{\widehat{S}(f) \mid \quad & S \text{ is a computation on } t,\\ & \widehat{S}(\varepsilon) = q,\\ & \forall g \in \mathcal{B}(t) - \{f\} : \widehat{S}(g) \in Q_{\text{fin}}\}\end{aligned}$$

That is, $q' \in \widehat{\lambda}(q, t, f)$ if q' is the state at the border node f of t obtained by a computation of A on t which starts at the root of t with the state q and which ends at all border nodes other than f with a final state. We also say that q' is the result of a computation on the pointed tree (t, f) starting in q.

We define the set of all pointed trees such that the response function assigns them a final state:

$$\widehat{L}_q = \{(t, f) \mid \widehat{\lambda}(q, t, f) \cap Q_{\text{fin}} \neq \emptyset\}.$$

[4]When considering the algebraic notion of minimality of automata, one excludes automata with nonaccessible states (a state is accessible if there exists a computation in which it occurs).

We can characterize the tree language recognized by A by $\widehat{L_{q_0}}$ as follows:

$$L = \mathcal{L}(A) \quad \text{iff } \widehat{L} = \{(t,f) \mid \widehat{\lambda}(q_0,t,f) \cap Q_{\text{fin}} \neq \emptyset\}.$$

If we define $L_q = \{t \in \Sigma^{\#} \mid (t,f) \in \widehat{L_q}\}$ (where it does not matter whether we require $(t,f) \in \widehat{L_q}$ for at least one or for all $f \in \mathcal{B}(t)$), then we obtain that $\mathcal{L}(A) = L_{q_0}$.

The following property expresses the "associativity of computation" for descending tree automata.

$$\widehat{\lambda}(\widehat{\lambda}(q,t,f),s,g) = \widehat{\lambda}(q,(t,f)(s,g)) \tag{2}$$

For this equation we assume the canonical extension of the response function from states to sets of states. Its proof proceeds by induction on the structure of (t,f), using the recursive definition of $\widehat{\lambda}$.

We use the response function in order to define some standard automata properties for a descending tree automaton A in a concise way. A state $q \in Q$ is called *accessible* if there exists a pointed tree (t,f) such that $\widehat{\lambda}(q_0,t,f) \ni q$, i.e., an "elsewhere successful" computation on t can reach this state at the border node f. For the concept of minimality of section 6, we need to assume that the states of the automata considered are all accessible. The state q is called *coaccessible* if there exists a pointed tree (t,f) such that $\widehat{\lambda}(q,t,f) \cap Q_{\text{fin}} \neq \emptyset$; that is, there exists a successful computation on t starting in this state. The automaton A is called *trim* if all states are accessible and coaccessible. Of course, every finite descending tree automaton can be effectively transformed into an equivalent trim one. Two states q, $q' \in Q$ are called *separable* if there exists a pointed tree (t,f) such that $\widehat{\lambda}(q,t,f) \cap Q_{\text{fin}} = \emptyset$ and $\widehat{\lambda}(q',t,f) \cap Q_{\text{fin}} \neq \emptyset$, or vice versa. The automaton A is called *reduced* if all states are pairwise separable.

Finally, we call two states $q, q' \in Q$ *disjoint* if L_q and $L_{q'}$ are disjoint sets, $L_q \cap L_{q'} = \emptyset$.

The function $\widehat{\lambda}$ can also be defined recursively on the set of pointed trees $\Sigma^{(\#)}$ by

$$\begin{aligned}
\widehat{\lambda}(q,\Omega,\varepsilon) &= \{q\},\\
\widehat{\lambda}(q,a+1t_1+2t_2,1f_1) &= \cup\{\widehat{\lambda}(q_1,t_1,f_1) \mid \exists q_2 \in Q : t_2 \in L_{q_2}, (q_1,q_2) \in \lambda(q,a)\},\\
\widehat{\lambda}(q,a+1t_1+2t_2,2f_2) &= \cup\{\widehat{\lambda}(q_2,t_2,f_2) \mid \exists q_1 \in Q : t_1 \in L_{q_1}, (q_1,q_2) \in \lambda(q,a)\}.
\end{aligned}$$

Observe that the condition $t_2 \in L_{q_2}$, which is $\widehat{\lambda}(q_2,t_2,f_2) \cap Q_{\text{fin}} \neq \emptyset$ for some $f_2 \in \mathcal{B}(t_2)$, as well as the condition $t_1 \in L_{q_1}$ are checked by recursion since t_1 and t_2 are smaller trees than $a+1t_1+2t_2$. We see that the recursive definition of $\widehat{\lambda}$ suggests a depth-first transversal of the tree t to be evaluated: for each two direct subtrees t_1 and t_2 of a subtree at some node f of t, the subtree t_1 is traversed once a computation on t_2 is successful.

5. Introducing l-r-determinism. We now define the counterpart of the notion of ascending deterministic tree automata.

DEFINITION 2. *A finite descending tree automaton A is* l-r-deterministic *if the response function $\widehat{\lambda}$ of A is deterministic; this means that, applied on any state and any pointed tree, its value is a singleton.*

Formally, the defining condition means that $card(\widehat{\lambda}(q,t,f)) = 1$ holds for all $q \in Q$ and all $(t,f) \in \Sigma^{(\#)}$. This is equivalent to the condition that $card(\widehat{\lambda}(q_0,t,f)) = 1$ holds for all $(t,f) \in \Sigma^{(\#)}$ and to the condition that $card(\widehat{\lambda}(q,t,f)) = 1$ holds for all $q \in Q$ and all $(t,f) \in \Gamma$.

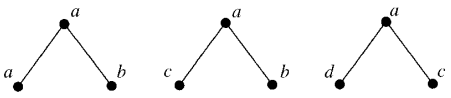


FIG. 3. *The trees of* L_0.

One can express l-r-determinism also without using the response function, namely by the following condition.[5] Whenever there is a nondeterministic transition with two successor state pairs, then the left successor states can be different only if the right successor states are disjoint, and vice versa. Formally, this condition says that if (q_1, q_2), $(q_1', q_2') \in \lambda(q, a)$, then

$$q_1 \neq q_1' \quad \text{implies} \quad L_{q_2} \cap L_{q_2'} = \emptyset,$$
$$q_2 \neq q_2' \quad \text{implies} \quad L_{q_1} \cap L_{q_1'} = \emptyset.$$

In particular, if the two left successor states q_1 and q_1' are different, then they are disjoint; if they are equal, then the corresponding right successor states q_2 and q_2' are also equal.

For the evaluation of an l-r-deterministic descending tree automaton, this means at a node f of a tree t several transitions might still be possible; the successor state to the right side, however, is unique for all the transitions that can be continued successfully on the left side (and vice versa).

That is, whichever of the transitions $(q_1^1, q_2^1), \ldots, (q_1^k, q_2^k)$ is chosen, where the subtree t' of t starting at $f2$ is recognized by the right successor state (i.e., t' lies in $L_{q_2^1}, \ldots, L_{q_2^k}$), then the left successor state is the same ($q_1^1 = \cdots = q_1^k$).

In order to compare the two notions of determinism, we consider the tree language $L_0 = \{a + 1a + 2b, a + 1c + 2b, a + 1d + 2c\}$ over $\Sigma = \{a, b, c, d\}$ depicted in Figure 3. Clearly, L_0 *cannot* be recognized by a deterministic descending tree automaton. If (q_1, q_2) were the only successor state pair in $\lambda(q_0, a)$, then $a, c \in L_{q_1}$ and $b, c \in L_{q_2}$ and hence also $a + 1c + 2c$ would be recognized. The same reasoning can be made for the tree language $L_0' = \{a+1a+2b, a+1d+2c\}$. One can give simple l-r-deterministic descending tree automata recognizing L_0 and L_0'. We will derive them in a way which will serve later as an example for the l-r-determinization algorithm (from the proof of Theorem 3) and illustrate its intrinsic difficulty.

The descending tree automaton $A_0 = (\{q_0, \ldots, q_6, q_{\text{fin}}\}, q_0, \lambda, \{q_{\text{fin}}\})$ recognizes L_0 if we set $\lambda(q_0, a) = \{(q_1, q_2), (q_3, q_4), (q_5, q_6)\}$ and $\lambda(q_1, a) = \lambda(q_2, b) = \lambda(q_3, c) = \lambda(q_4, b) = \lambda(q_5, d) = \lambda(q_6, c) = \{(q_{\text{fin}}, q_{\text{fin}})\}$.

We now set $P_1 = \{q_1, q_3\}$, $P_2 = \{q_2, q_4\}$, $P_3 = \{q_5\}$, and $P_4 = \{q_6\}$ and define the descending tree automaton $A_0^d = (\mathcal{P}(Q), \{q_0\}, \lambda^d, Q_{\text{fin}}^d)$, where $\lambda^d(\{q_0\}, a) = \{(P_1, P_2), (P_3, P_4)\}$ and $\lambda^d(P_i, x) = \{(q_{\text{fin}}, q_{\text{fin}})\}$ if $x \in L_q$ for some $q \in P_i$ and $Q_{\text{fin}}^d = \{P \in \mathcal{P}(Q) | P \cap \{q_{\text{fin}}\} \neq \emptyset\}$.

It is easy to see that A^d is l-r-deterministic and recognizes L_0. For example, if the subtree to the right side is the punctual tree b, then the successor state to the other side is determined to be P_1. Thus $\widehat{\lambda}(\{q_0\}, a + 2b, 1)$ contains only the one element P_1.

[5] In a strict sense, this condition is not equivalent to the one in Definition 2 since it allows the value of $\hat{\lambda}$ to be a singleton or the empty set. The two conditions coincide if we identify the empty set with the singleton containing a "trap state." This identification is natural when one considers minimal deterministic automata (which have either a partial transition function or a unique trap state).

The following lemma describes the set of successor state pairs (q_1, q_2) of a state q at a node with some label a. For every pair, there exists a tree $s = a+1s_1+2s_2$ which is accepted by the automaton starting in the state q, and the left successor state q_1 is determined by the right subtree of any such tree (and vice versa, the right successor state q_2 is determined by the left subtree of any such tree).

LEMMA 1. *Let $A = (Q, \lambda, q_0, Q_{\mathrm{fin}})$ be a trim l-r-deterministic descending tree automaton. Then*

$$\lambda(q,a) = \{(q_1,q_2) \mid \exists s_1, s_2 \in \Sigma^{\#} : \quad a + 1s_1 + 2s_2 \in L_q, \\ q_1 = \widehat{\lambda}(q, a + 2s_2, 1), \quad q_2 = \widehat{\lambda}(q, a + 1s_1, 2)\}.$$

Proof. We call K the set on the right-hand side. We will show $\lambda(q,a) \subseteq K$ first. Let $(q_1, q_2) \in \lambda(q,a)$. Since A is assumed trim, q_1 and q_2 are coaccessible. Thus there exist $s_1, s_2 \in \Sigma^{\#}$ such that

$$\forall g_1 \in \mathcal{B}(s_1) : \quad \widehat{\lambda}(q_1, s_1, g_1) \in Q_{\mathrm{fin}}, \\ \forall g_2 \in \mathcal{B}(s_2) : \quad \widehat{\lambda}(q_2, s_2, g_2) \in Q_{\mathrm{fin}}.$$

If we choose $s = a + 1s_1 + 2s_2$, then for $g \in \mathcal{B}(s)$, say, $g = 1g_1$, equation (2) implies that $\widehat{\lambda}(q, s, g) = \widehat{\lambda}(\widehat{\lambda}(q, a+2s_2, 1), s_1, g_1)$ and thus $\lambda(q, s, g) \in Q_{\mathrm{fin}}$. Together with the definition of $\widehat{\lambda}$ applied to $\widehat{\lambda}(q, a + 2s_2, 1)$, $\widehat{\lambda}(q, a + 1s_1, 2)$ this gives $(q_1, q_2) \in K$.

For the other inclusion, $\lambda(q,a) \supseteq K$, let (q_1, q_2) be an element in K and the tree $s = a+1s_1+2s_2$ satisfy the condition in the definition of K. Then $q_1 = \widehat{\lambda}(q, a+2s_2, 1)$ implies that there exists $q_2' \in Q$ such that $(q_1, q_2') \in \lambda(q,a)$ and $\widehat{\lambda}(q_2', s_2, f) \in Q_{\mathrm{fin}}$ for all $f \in \mathcal{B}(s_2)$. Symmetrically, $q_2 = \widehat{\lambda}(q, a + 1s_1, 2)$ implies that there exists $q_1' \in Q$ such that $(q_2, q_1') \in \lambda(q,a)$ and $\widehat{\lambda}(q_1', s_1, f) \in Q_{\mathrm{fin}}$ for all $f \in \mathcal{B}(s_1)$.

Finally, we use equation (2) and the fact that $\widehat{\lambda}(q, s, g) \in Q_{\mathrm{fin}}$ holds for all $g \in \mathcal{B}(s)$ to infer the following. For all $g_1 \in \mathcal{B}(s_1)$,

$$\begin{aligned} \widehat{\lambda}(q_1, s_1, g_1) &= \widehat{\lambda}(\widehat{\lambda}(q, a + 2s_2, 1), s_1, g_1) \\ &= \widehat{\lambda}(q, a + 1s_1 + 2s_2, 1g_1) \in Q_{\mathrm{fin}} \end{aligned}$$

since $g_1 \in \mathcal{B}(s_1)$ means that $1g_1 \in \mathcal{B}(s)$.

Thus $s_1 \in L_{q_1}$ and hence $q_2' \in \widehat{\lambda}(q, a + 1s_1, 2)$. The l-r-determinism then yields $q_2' = q_2$. Symmetrically, $q_1' = q_1$ and thus $(q_1, q_2) \in \lambda(q,a)$. □

The following corollary expresses the connection between the left and the right successor states in a pair (q_1, q_2). Namely, the right successor state recognizes the right subtree that, in turn, determines the left successor state, and vice versa.

COROLLARY 2. *Let $A = (Q, \lambda, q_0, Q_{\mathrm{fin}})$ be a trim l-r-deterministic descending tree automaton. Then*

$$\lambda(q,a) = \{(q_1,q_2) \mid \exists s_1 \in L_{q_1} : q_2 = \widehat{\lambda}(q, a + 1s_1, 2), \\ \exists s_2 \in L_{q_2} : q_1 = \widehat{\lambda}(q, a + 2s_2, 1)\}.$$

Proof. We use the notation of the previous proof and call K' the set on the right-hand side of the above equation. We will show that $K = K'$. The direction $\subseteq$ says that $a + 1s_1 + 2s_2 \in L_q$ implies $q_1 \in L_{q_1}$ and $q_2 \in L_{q_2}$, but this is clear: if $a + 1s_1 + 2s_2 \in L_q$, then there must exist $(q_1', q_2') \in \lambda(q,a)$ such that $s_1 \in L_{q_1'}$ and

$s_2 \in L_{q_2'}$. Hence $q_1' = \widehat{\lambda}(q, a + 2s_2, 1)$ and $q_2' = \widehat{\lambda}(q, a + 1s_1, 2)$ and thus $q_1' = q_1$ and $q_2' = q_2$.

Now if $q_2 = \widehat{\lambda}(q, a + 1s_1, 2)$, then there exists q_1' such that $s_1 \in L_{q_1'}$ and $(q_1', q_2) \in \lambda(q, a)$. Since $q_2 \in L_{q_2}, a + 1s_1 + 2s_2 \in L_q$. This shows the direction $\supseteq$. □

We now introduce the structural property which, as we will show, corresponds to the new notion of determinism.

DEFINITION 3. *A tree language $L \subseteq \Sigma^{\#}$ is called homogeneous if its membership relation is "subtree distributive" in the following manner. For all subtrees $s_1, s_2, t_1, t_2 \in \Sigma^{\#}$, all trees $t \in \Sigma^{\#}$ with border node $f \in \mathcal{B}(t)$, and all labels $a \in \Sigma$, the conjunction of*

$$\begin{aligned} t + f(a + 1t_1 + 2t_2) &\in L, \\ t + f(a + 1s_1 + 2t_2) &\in L, \\ t + f(a + 1t_1 + 2s_2) &\in L \end{aligned}$$

implies $t + f(a + 1s_1 + 2s_2) \in L$.

Thus in a homogeneous tree language L, if one has the three pairs (t_1, t_2), (s_1, t_2), and (t_1, s_2) for the left and right subtrees at the same node f of a tree in L, then the fourth pair (s_1, s_2) also yields a tree in L.

Remark 1. A tree language $L \subseteq \Sigma^{\#}$ is homogeneous iff for all sets of subtrees L_1, L_2, K_1, and K_2 such that the conjunction of

$$\begin{aligned} t + f(a + 1L_1 + 2L_2) &\subseteq L, \\ t + f(a + 1K_1 + 2K_2) &\subseteq L, \\ (L_1 \cap K_1 \neq \emptyset \quad &\text{or} \quad L_2 \cap K_2 \neq \emptyset) \end{aligned}$$

implies $t + f(a + 1(L_1 \cup K_1) + 2(L_2 \cup K_2)) \subseteq L$.

Proof. Let L be homogeneous and L_1, L_2, K_1, and K_2 be as in the statement with, say, $u_1 \in L_1 \cap K_1 \neq \emptyset$. Given $t_1 \in L_1$, $t_2 \in L_2$, $s_1 \in K_1$, and $s_2 \in K_2$, we will show that $t + f(a + 1s_1 + 2t_2) \in L$ and $t + f(a + 1t_1 + 2s_2) \in L$. The first follows from

$$\begin{aligned} t + f(a + 1s_1 + 2s_2) &\in L, \\ t + f(a + 1u_1 + 2s_2) &\in L, \\ t + f(a + 1u_1 + 2t_2) &\in L. \end{aligned}$$

The second membership statement follows from

$$\begin{aligned} t + f(a + 1t_1 + 2t_2) &\in L, \\ t + f(a + 1u_1 + 2t_2) &\in L, \\ t + f(a + 1u_1 + 2s_2) &\in L. \end{aligned}$$

In order to show that the condition in the remark implies that L is homogeneous, we take trees s_1, s_2, t_1, and t_2 as in the formulation of the definition and set $K_1 = \{s_1\}$, $K_2 = \{t_2\}$, $L_1 = \{t_1\}$, and $L_2 = \{s_2, t_2\}$. □

The class of homogeneous languages contains properly the class of languages which are called *path closed* in [12]. A path is a tree where each node has at most one descendant. A path of a tree t is a tree p with the same labeling on its support $\mathrm{Supp}(p) \subseteq \mathrm{Supp}(t)$. A tree is an element of a path-closed language L already if each

of its paths is a path of some tree of L. This condition characterizes path closedness; one can show that it is equivalent to the following:

$$t + f(a + 1s_1 + 2s_2) \in L,$$
$$t + f(a + 1t_1 + 2t_2) \in L$$

implies $t + f(a + 1s_1 + 2t_2) \in L$, or, equivalently (cf. Remark 1),

$$t + f(a + 1L_1 + 2L_2) \in L,$$
$$t + f(a + 1K_1 + 2K_2) \in L$$

implies $t + f(a + 1(L_1 \cup K_1) + 2(L_2 \cup K_2)) \in L$.

Just as the path-closed tree languages correspond to those recognizable by deterministic descending tree automata [12], the homogeneous tree languages correspond to those recognizable to l-r-deterministic descending tree automata.

THEOREM 2. *A [recognizable] tree language L is homogeneous iff it can be recognized by a [finite] l-r-deterministic descending tree automaton A.*

Proof. Let L be recognized by the trim l-r-deterministic descending tree automaton $A = (Q, \lambda, q_0, Q_{\text{fin}})$. Let $t, s_1, s_2, t_1, t_2 \in \Sigma^{\#}$, $a \in \Sigma$, and $f \in \mathcal{B}(t)$ be such that

$$t + f(a + 1t_1 + 2t_2) \in L, t + f(a + 1s_1 + 2t_2) \in L \quad \text{and} \quad t + f(a + 1t_1 + 2s_2) \in L.$$

Let $q = \widehat{\lambda}(q_0, t, f)$ be the result of a computation on the pointed tree (t, f), and let $q_1 = \widehat{\lambda}(q, a + 2s_2, 1)$ and $q_2 = \widehat{\lambda}(q, a + 1s_1, 2)$ be the results of a computation on the basic pointed trees $(a + 2s_2, 1)$ and $(a + 1s_1, 2)$ starting in q. Corollary 2 yields that $(q_1, q_2) \in \lambda(q, a)$. The definition of l-r-determinism now implies that $s_1, t_1 \in L_{q_1}$ and $s_2, t_2 \in L_{q_2}$. Thus $a + 1s_1 + 2s_2 \in L_1$, and hence $t + f(a + 1s_1 + s_2) \in \mathcal{L}(A)$.

This shows the homogeneity of L and the "if" direction. The "only if" direction will follow from Theorem 3 below (but also from Theorem 4 in the next section). □

THEOREM 3. *A finite descending tree automaton A recognizing a homogeneous tree language can be effectively transformed into an equivalent l-r-deterministic descending tree automaton A^d.*

The naïve extension of the powered construction would define

$$\lambda^d(P, a) = \{(\widehat{\lambda}(P, a + 2t_2, 1), \widehat{\lambda}(P, a + 1t_1, 2)) \mid a + 1t_1 + 2t_2 \in L_P\}.$$

However, the example of the tree automaton A_0 recognizing the tree language L_0 given above shows that this does not always yield an l-r-deterministic descending tree automaton. We will give a construction which yields A_0^d as in the example. The idea is to augment the successor states P_1 and P_2 in the pairs above with all successor states q_1 and q_2 of A such that $L_{q_1} \subseteq L_{P_1}$ and $L_{q_2} \subseteq L_{P_2}$.

Proof. We denote by λ^1 and λ^2 the projections of λ on the first and second component, respectively. As usual, we use the same symbol for a function and its canonical extension to subsets of its domain. Thus $\lambda(P, a) \cup \{\lambda(q, a) \mid q \in P\}$ for $P \subseteq Q$. Furthermore, let $L_P = \cup\{L_q \mid q \in P\}$.

Given $A = (Q, \lambda, q_0, Q_{\text{fin}})$ recognizing the homogeneous tree language L, we will construct the descending tree automaton $A^d = (Q^d, \lambda^d, q_0^d, Q_{\text{fin}}^d)$. For $P \subseteq Q$ and $a \in \Sigma$, we set

$$\lambda^d(P, a) = \{(\{q_1 \in \lambda^1(q, a) \mid a + 1L_{q_1} + 2t_2 \subseteq L_P\},$$
$$\{q_2 \in \lambda^2(q, a) \mid a + 1t_1 + 2L_{q_2} \subseteq L_P\}) \mid a + 1t_1 + 2t_2 \in L_P\}.$$

We define $q_0^d = \{q_0\}$, $Q^d \subseteq \mathcal{P}(Q)$ as the set of all accessible states in $\mathcal{P}(Q)$ and $Q^d_{\text{fin}} = \{P \in Q^d \mid P \cap Q_{\text{fin}} \neq \emptyset\}$.

Since we will need the homogeneity of L_P, we will first show that for every $P \in Q^d$ there exists a pointed tree $(t, f) \in \Sigma^{(\#)}$ such that $L_P = \{s \mid t + fs \in L\}$. Since we consider accessible states only, we can proceed by induction on $\Sigma^{(\#)}$. Clearly, q_0^d corresponds to $(t, f) = (\Omega, \varepsilon)$. Now if $P_1 \in \lambda^d(P, a + 2t_2, 1)$, where by induction hypothesis P corresponds to (t, f), we will show $L_{P_1} = \{s \mid (t + fa + f2t_2) + f1s \in L\}$. By the definition of λ^d, there exists a tree t_1 such that $t + f(a + 1t_1 + 2t_2) \in L$ and $t + f(a + 1L_{P_1} + 2t_2) \subseteq L$. However, for every tree t_1' such that $t + f(a + 1t_1' + 2t_2) \in L$, there exists a state q_1' such that $t_1' \in L_{q_1'}$ and $a + 1L_{q_1'} + 2t_2 \subseteq L_P$. Thus $q_1' \in P_1$ and the statement follows.

The fact that A^d recognizes $L = \mathcal{L}(A)$ is a special case (for $P = \{q_0\}$) of the equivalence

$$t \in L_P^d \quad \text{iff } t \in L_P, \tag{3}$$

which we will prove by induction over $\Sigma^{\#}$. The base step $t = \Omega$ is clear by the definition of Q^d_{fin}. Assuming the equivalence for the trees s_1 and s_2, we will show it for $s = a + 1s_1 + 2s_2$. If $s \in L_P^d$, then there exist states P_1 and P_2 of A^d such that

$$s_1 \in L^d_{P_1}, \qquad s_2 \in L^d_{P_2}, \qquad (P_1, P_2) \in \lambda^d(P, a).$$

Applying the induction hypothesis and the definition of λ^d yields $s_1 \in L_{P_1}$, $s_2 \in L_{P_2}$ and there exist trees t_1 and t_2 such that

$$\begin{aligned} a + 1L_{P_1} + 2t_2 &\subseteq L_P, \\ a + 1t_1 + 2L_{P_2} &\subseteq L_P, \\ a + 1t_1 + 2t_2 &\in L_P. \end{aligned}$$

The homogeneity of L_P yields directly that $s \in L_P$. Vice versa, if $s = a + 1s_1 + 2s_2 \in L_P$, say $s \in L_q$, where $q \in P$, then there exist states q_1 and q_2 of A such that $s_1 \in L_{q_1}$, $s_2 \in L_{q_2}$, and $(q_1, q_2) \in \lambda(q, a)$. Hence there exist sets P_1 and P_2 of states of A which include q_1 and q_2 (hence $s_1 \in L_{P_1}$ and $s_2 \in L_{P_2}$) such that $(P_1, P_2) \in \lambda^d(q, a)$. Applying the induction hypothesis yields $s_1 \in L^d_{P_1}$ and $s_2 \in L^d_{P_2}$ and thus $s \in L_P^d$. In order to show that A^d is l-r-deterministic, we only show the left determinism; the proof of the right determinism is symmetrical. Assuming two successor state pairs (P_1, P_2) and (P_1', P_2') in $\lambda^d(P, a)$ with $L^d_{P_2} \cap L^d_{P_2'} \neq \emptyset$, we will show $P_1 = P_1'$.

The assumption together with (3) yields

$$\begin{aligned} a + 1L_{P_1} + 2L_{P_2} &\subseteq L_P, \\ a + 1L_{P_1'} + 2L_{P_2'} &\subseteq L_P, \\ L_{P_2} \cap L_{P_2'} &\neq \emptyset. \end{aligned}$$

The homogeneity of L_P and Remark 1 imply that $a+1(L_{P_1} \cup L_{P_1'})+2(L_{P_2} \cup L_{P_2'}) \subseteq L_P$, but this, together with the definition of λ^d, yields $P_1 = P_1'$.

If L is recognizable and A is chosen finite, then A^d is also finite. The determinization described above can then be done effectively by restricting all occurring sets of trees to sets of trees of height $h \leq 2^{|Q|}$. This is not a real restriction since the above statements are true iff they are true when quantified over the trees of height $h \leq 2^{|Q|}$. □

If $\mathcal{L}(A)$ is path closed, then the above construction yields a deterministic descending tree automaton. If $\mathcal{L}(A)$ is not homogeneous, then A^d recognizes just a homogeneous superset of $\mathcal{L}(A)$.

6. Introducing l-r-minimization. The following definition is the analogue of Definition 1.

DEFINITION 4. *The* descending Nerode equivalence relation $\approx_L$ *of a tree language* $L \subseteq \Sigma^{\#}$ *is the coarsest right invariant equivalence relation on the tree monoid* $\Sigma^{(\#)}$ *which saturates* $\widehat{L}$.

That is, for all $(t_1, f_1), (t_2, f_2) \in \Sigma^{(\#)}$ if $(t_1, f_1) \approx_L (t_2, f_2)$, then

$$(t_1, f_1)(t, f) \approx_L (t_2, f_2)(t, f) \quad \forall (t, f) \in \Sigma^{(\#)}$$

and

$$(t_1, f_1) \in \widehat{L} \quad \text{iff } (t_2, f_2) \in \widehat{L}.$$

Furthermore, if $\approx$ is another equivalence relation with these two properties, then $\approx_L$ is coarser than $\approx$, i.e., $(t_1, f_1) \approx (t_2, f_2)$ always implies $(t_1, f_1) \approx_L (t_2, f_2)$.

We can define $\approx_L$ equivalently by the following condition. For all $(t_1, f_1), (t_2, f_2) \in \Sigma^{(\#)}$, $(t_1, f_1) \approx_L (t_2, f_2)$ iff

$$\forall (s, g) \in \Sigma^{(\#)} : \quad ((t_1, f_1)(s, g) \in \widehat{L} \quad \text{iff } (t_2, f_2)(s, g) \in \widehat{L}). \tag{4}$$

If we define $(t, f)^{-1}\widehat{L} = \{(s, g) \in \Sigma^{(\#)} \mid (t, f)(s, g) \in \widehat{L}\}$, then (4) is the same as $(t_1, f_1)^{-1}\widehat{L} = (t_2, f_2)^{-1}\widehat{L}$. If we set

$$(t, f)^{-1}L = \{s \in \Sigma^{\#} \mid t + fs \in L\},$$

then (4) becomes, more simply,

$$(t_1, f_1) \approx_L (t_2, f_2) \quad \text{iff } (t_1, f_1)^{-1}L = (t_2, f_2)^{-1}L. \tag{5}$$

Given the descending Nerode equivalence of a tree language L, we define the descending tree automaton $\mathcal{A}(L) = (Q^L, \lambda^L, q_0^L, Q_{fin}^L)$ where

$$\begin{aligned} Q^L &= \{(s, g)^{-1}L \mid (s, g) \in \Sigma^{(\#)}\}, \\ q_0^L &= L, \\ Q_{\text{fin}}^L &= \{K \in Q^L \mid \Omega \in K\}, \quad \text{and} \\ \lambda^L(K, a) &= \{((a + 2s_2, 1)^{-1}K, (a + 1s_1, 2)^{-1}K) \mid a + 1s_1 + 2s_2 \in K\}. \end{aligned}$$

Note that $q_0^L \in Q^L$ since $L = (\Omega, \varepsilon)^{-1}L$.

LEMMA 2. *The descending tree automaton* $\mathcal{A}(L)$ *of a homogeneous tree language* L *is l-r-deterministic and recognizes* L.

Proof. We will first prove, by structural induction on (t, f), the validity of the equation

$$\widehat{\lambda^L}((s, g)^{-1}L, t, f) = (t, f)^{-1}((s, g)^{-1}L). \tag{6}$$

The base step $(t, f) = (\Omega, \varepsilon)$ follows from the definition of $\widehat{\lambda}$, since generally $\widehat{\lambda}(q, \Omega, \varepsilon) = q$ for any state q and any transition function λ. For the induction step, we suppose (6)

for (t_1, f_1) and (t_2, f_2) and show that for $(t, f) = (a+1t_1+t_2, 1f_1)$, if $q \in \widehat{\lambda^L}((s, g)^{-1}L, t, f)$, then $q = ((s, g)(t, f))^{-1}L$.

By (2), there exists a state $q_1 \in \widehat{\lambda^L}((s, g)^{-1}L, a+2t_2, 1)$ such that $q \in \widehat{\lambda^L}(q_1, t_1, f_1)$. By the definition of $\widehat{\lambda^L}$, there exists a state $q_2 \in Q^L$ such that

$$(q_1, q_2) \in \lambda^L((s, g)^{-1}L, a), \tag{7}$$

$$\widehat{\lambda^L}(q_2, t_2, f_2) \cap Q^L_{\mathrm{fin}} \neq \emptyset \quad \text{for } f_2 \in \mathcal{B}(t_2). \tag{8}$$

The definition of λ^L and (7) imply the existence of $s_1, s_2 \in \Sigma^{\#}$ such that

$$a + 1s_1 + 2s_2 \in (s, g)^{-1}L, \tag{9}$$

$$q_1 = (a + 2s_2, 1)^{-1}(s, g)^{-1}L, \tag{10}$$

$$q_2 = (a + 1s_1, 2)^{-1}(s, g)^{-1}L. \tag{11}$$

We can write (9) as

$$s + g(a + 1s_1 + 2s_2) \in L. \tag{12}$$

If we put the induction hypothesis on (t_2, f_2) and (11) together, we obtain that $\widehat{\lambda^L}(q_2, t_2, f_2) = ((s, g)(a + 1s_1, 2)(t_2, f_2))^{-1}L$. According to (8), $\Omega \in \widehat{\lambda^L}(q_2, t_2, f_2)$ or

$$s + g(a + 1s_1 + 2t_2) \in L. \tag{13}$$

We will use the two facts above and the homogeneity of L in order to show that the state q is of the form $q = ((s, g)(t, f))^{-1}L$. Since $q \in \widehat{\lambda^L}(q_1, t_1, f_1)$ and q_1 is of the form given by (10), the induction hypothesis applied to (t_1, f_1) yields

$$q = ((s, g)(a + 2s_2, 1)(t_1, f_1))^{-1}L. \tag{14}$$

In order to show the conjectured equality, which is $q = ((s, g)(a + 2t_2, 1)(t_1, f_1))^{-1}L$, let $\bar{t} \in q$; that is,

$$s + g(a + 1(t_1 + f_1\bar{t}) + 2s_2) \in L. \tag{15}$$

The homogeneity of L together with (12), (13), and (15) now implies

$$s + g(a + 1(t_1 + f_1\bar{t}) + 2t_2) \in L, \tag{16}$$

which is the same as $\bar{t} \in ((s, g)(a + 2t_2, 1)(t_1, f_1))^{-1}L$. Vice versa, the homogeneity of L applied to (12), (13), and (16) implies (15), which proves $\bar{t} \in q$ and hence the statement on the form of q.

Thus the only element of $\widehat{\lambda^L}((s, g)^{-1}L, t, f)$ is $q = ((s, g)(t, f))^{-1}L$, which proves the left determinism and, by symmetry, the l-r determinism. The equivalences

$$\begin{aligned}(t, f) \in \widehat{\mathcal{L}}(\mathcal{A}(L)) \quad &\text{iff } (\Omega, \varepsilon) \in \widehat{\lambda^L}(L, t, f),\\ &\text{iff } (\Omega, \varepsilon) \in (t, f)^{-1}L,\\ &\text{iff } (t, f) \in \widehat{L}\end{aligned}$$

show that $L = \mathcal{L}(\mathcal{A}(L))$ and complete the proof of Lemma 2. □

The l-r-deterministic descending tree automaton $\mathcal{A}(L)$ is reduced. Clearly, for a state $M = (t, f)^{-1}L \in Q^L$, $\widehat{L}_M = \{(s, g) \in \Sigma^{(\#)} \mid \widehat{\lambda}(M, s, g) \in Q^L_{\text{fin}}\}$ and hence $L_M = M$. Therefore, the two equivalent states are equal.

We recall that an automaton A' is called *minimal* (in the algebraic sense) in a given class of automata iff every equivalent automaton A in the class can be mapped onto A' by an automaton morphism. A morphism π from $A = (Q, \lambda, q_0, Q_{\text{fin}})$ to $A' = (Q', \lambda', q'_0, Q'_{\text{fin}})$ is a mapping $\pi : Q \to Q'$ (which is canonically extended to $\pi : \mathcal{P}(Q \times Q) \to \mathcal{P}(Q' \times Q')$) such that $\pi(q_0) = q'_0$, $\pi(Q_{\text{fin}}) \subseteq Q'_{\text{fin}}$ and $\pi(\lambda(q, a)) = \lambda'(\pi(q), a)$. This condition implies, of course, the minimality of the number of states. As already stated in footnote 4, one excludes automata with nonaccessible states when considering the algebraic notion of minimality of automata.

THEOREM 4. *For every homogeneous tree language L there exists a minimal l-r-deterministic descending tree automaton recognizing L. It is given by $\mathcal{A}(L)$. This is the unique (up to isomorphism) reduced l-r-deterministic descending tree automaton recognizing L. It is finite iff L is recognizable.*

Proof. We assume a l-r-deterministic descending tree automaton $A = (Q, \lambda, q_0, Q_{\text{fin}})$ whose states are all accessible and which recognizes L. The morphism π from A on $\mathcal{A}(L)$ is given by the mapping $\pi : Q \to Q^L$, $\pi(q) = L_q$.

First we show that π is indeed a function into Q^L. Let $q \in Q$ be a state of A and $(t, f) \in \Sigma^{\#}$ be a pointed tree such that $\widehat{\lambda}(q_0, t, f) = q$.

By equation (2), the state $\widehat{\lambda}(q, s, g) = \widehat{\lambda}(\widehat{\lambda}(q_0, t, f), s, g)$ which is equal to $\widehat{\lambda}(q_0, (t, f)(s, g))$. This is a final state of A iff $(t, f)(s, g) \in \widehat{L}$. Thus $L_q = (t, f)^{-1}L$ and $\pi(q)$ is indeed a state of $\mathcal{A}(L)$.

Clearly, π is surjective. Given $(t, f)^{-1}L \in Q^L$, we have $(t, f)^{-1}L = \pi(q)$ where $q = \widehat{\lambda}(q_0, t, f)$, since $L_q = (t, f)^{-1}L$.

If $(t, f)^{-1}L \neq \emptyset \in Q^L$, then $\lambda^L(q^L_0, t, f) \neq \emptyset$, and setting $q = \widehat{\lambda}(q_0, t, f)$, we have $L_q = (t, f)^{-1}L = \pi(q)$ and thus π is surjective. (The empty set $\emptyset \in Q^L$ is the image of any not accessible or not coaccessible state of A. If A is assumed to be trim, we define Q^L without the element $\emptyset$.)

In order to show that $(q_1, q_2) \in \lambda(q, a)$ iff $(\pi(q_1), \pi(q_2)) \in \widehat{\lambda}(\pi(q), a)$, it is sufficient to show that $\pi(\widehat{\lambda}(q, t, f)) = \widehat{\lambda^L}(\pi(q), t, f)$, thanks to Corollary 2, but this is trivially clear from the l-r determinism of A.

By Lemma 2, $\mathcal{A}(L)$ is l-r deterministic and recognizes L. If A is reduced, then π is bijective, which proves the second statement of Theorem 4 if L is recognizable, then there exists, by Theorem 2, a l-r-deterministic finite descending tree automaton recognizing L. By *reducing* A, that is, by identifying unseparable states, $\mathcal{A}^L$ can be obtained effectively and is, of course, finite if A is finite. □

If the tree language L is path closed, then $\mathcal{A}(L)$ is not only l-r deterministic but also deterministic. Of course, every deterministic descending tree automaton recognizing L is also l-r deterministic. Together, this implies the following corollary whose first part has been shown in [12].

COROLLARY 3. *For every path-closed tree language L there exists a unique (up to isomorphism) reduced deterministic descending tree automaton recognizing L. This minimal deterministic descending tree automaton recognizing L is given by $\mathcal{A}(L)$ with, as a set of states, the left residuals of L (or the descending Nerode equivalence classes).*

Thus we have obtained the main result of [12, section II.11], with elegant methods

and sharpened it: $\mathcal{A}(L)$ is minimal not only in the class of all deterministic, but also in the larger class of l-r-deterministic descending tree automata. Moreover, we are able to describe $\mathcal{A}(L)$ algebraically solely by means of the tree language L.

We will now investigate the meaning of $\mathcal{A}(L)$ is the general case, where L is not necessarily a homogeneous tree language. We note that the intersection of any family of homogeneous sets is again homogeneous. Hence there exists a smallest homogeneous set containing L. We call it the *homogeneous closure* of L, written $\mathcal{HC}(L)$. Also in the general case, we call $\mathcal{A}(L)$ the minimal l-r-deterministic descending tree automaton of L; this is justified by the following characterization.

THEOREM 5. *For any tree language L, the minimal l-r-deterministic descending tree automaton $\mathcal{A}(L)$ is finite iff L is recognizable. It recognizes the homogeneous closure of L.*

Hence $\mathcal{A}(L)$ recognizes always an approximation of L (i.e., a superset).

Proof. The class $[(t,f)]_{\sim_L}$ of the ascending Nerode congruence $\sim_L$ is characterized exactly by (1), the set of all sets $(s,g)^{-1}L$ which contain (t,f), and (2), the set of all those sets which do not contain (t,f). Hence there are finitely many congruence classes iff there are finitely many sets $(s,g)^{-1}L$. According to property (5) from section 6, this holds iff there are finitely many descending Nerode congruence classes $[(t,f)] \approx_L$.

Next, we show that every $t \in \Sigma^{\#}$ which is recognized by $\mathcal{A}(L)$ starting in the state $(s,g)^{-1}L$ is an element of the homogeneous closure of $(s,g)^{-1}L$. We proceed by induction over t. If $t = \Omega$, the statement is true by definition of $\mathcal{A}(L)$. If $t = a + 1t_1 + 2t_2$, where we assume the statement for t_1 and t_2 to hold by induction hypothesis, then there exist trees s_1 and s_2 such that $a+1s_1+2s_2 \in (s,g)^{-1}L$, and t_1 and t_2 are recognized by $\mathcal{A}(L)$ starting in the corresponding left and right successor states, which are, respectively, $(a + 2s_2, 1)(s,g)^{-1}L$ and $(a + 1s_2, 2)(s,g)^{-1}L$. By induction hypothesis, $a + 1t_1 + 2s_2$ and $a + 1s_1 + 2t_2$ are in the homogeneous closure of the set $(s,g)^{-1}L$. This yields the statement for $t = a + 1t_1 + 2t_2$.

By definition, $\mathcal{A}(L)$ recognizes a homogeneous tree language which includes every tree of L. Hence the homogeneous closure $\mathcal{HC}(L)$ is a subset of $\mathcal{L}(\mathcal{A}(L))$, and, together with the above, the two sets are equal. □

COROLLARY 4. *A tree language L is recognizable iff the descending Nerode equivalence of L has a finite index iff there exists a frontier invariant equivalence relation on $\Sigma^{(\#)}$ which saturates $\widehat{L}$ and has a finite index.* □

7. Conclusion. The two existing notions of descending tree automata being either too restrictive or not amenable to algebraic tools, we have introduced a third, intermediate family. The investigation in this work has shown that this family of l-r-deterministic descending tree automata is a natural algebraic notion on which the usual automata theoretic methods apply. Our characterization of the corresponding family of homogeneous tree languages offers the possibility to approximate a recognizable set of trees by a homogeneous superset and to represent it by an l-r-deterministic descending tree automaton. As an immediate practical application, a type system like the one of [19], for example (for others, cf. [28]), can be made more powerful by incorporating this approximation.

Also with respect to the application on type systems, it would be useful to characterize the corresponding family of regular systems of equations. It seems less interesting, and also less feasible, to try the same for regular expressions. Another branch of further work would consist of exploring the algorithmic aspects. Here one can exploit

the relationship between l-r-deterministic descending tree automata and automata on strings that we have exhibited.

Acknowledgments. We thank A. Arnold and B. Courcelle for fruitful discussions and the anonymous referees for their useful detailed comments.

REFERENCES

[1] D. Beauquier and M. Nivat, *About rational subsets of algebras of infinite words*, in Automata, Languages and Programming, Lecture Notes in Comput. Sci. 194, W. Brauer, ed., Springer-Verlag, Berlin, 1985, pp. 33–42.
[2] M. P. Cohn, *Universal Algebra*, Harper and Row, New York, 1965.
[3] H. Comon and C. Delor, *Equational formulae with membership constraints*, Rapport de Recherche 649, LRI, Université de Paris Sud, Orsay, France, 1991; Inform. and Comput., 112 (1994), pp. 167–216.
[4] S. S. Cosmodakis, H. Gaifman, P. C. Kanellakis, and M. Y. Vardi, *Decidable optimization problems for database logic programs*, in Proc. 20th Annual ACM Symposium on Theory on Computing, ACM, New York, 1988, pp. 477–490.
[5] B. Courcelle, *On recognizable sets and tree automata*, in Resolution of Equations in Algebraic Structures, Vol. I, H. Aït-Kaci and M. Nivat, eds., Academic Press, Boston, 1989, pp. 93–125.
[6] N. Dershowitz and J.-P. Jouannaud, *Rewrite systems*, in Handbook of Theoretical Computer Science, Vol. B, J. van Leeuwen, ed., North–Holland, Amsterdam, 1990, pp. 243–309.
[7] J. Doner, *Tree acceptors and some of their applications*, J. Comput. System Sci., 4 (1970), pp. 406–451.
[8] P. J. Downey, R. Sethi, and R. E. Tarjan, *Variations on the common subexpression problem*, J. Assoc. Comput. Mach., 25 (1980), pp. 758–771.
[9] S. Eilenberg, *Automata, Languages and Machines*, Vol. B, Academic Press, New York, 1976.
[10] E. A. Emerson and C. S. Jutla, *The complexity of tree automata and logics of programs*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 328–337.
[11] F. Gécseg and M. Steinby, *Minimal ascending tree automata*, Acta Cybernet, 4 (1984), pp. 37–44.
[12] F. Gécseg and M. Steinby, *Tree Automata*, Akademiai Kiado, Budapest, 1984.
[13] R. Giegerich and K. Schmal, *Code selection techniques*: *Pattern matching, tree parsing, and inversion of derivors*, in Proc. 1988 European Symposium on Programming, Lecture Notes in Comput. Sci. 300, Springer-Verlag, Heidelberg, 1988, pp. 245–268.
[14] J. Heintze, *Set based program analysis*, Ph.D. thesis, School of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, 1992.
[15] N. Heintze and J. Jaffar, *A finite representation theorem for approximating logic programs*, in Proc. 17th ACM Conference on Principles of Programming Languages, ACM, New York, 1990, pp. 197–209.
[16] C. Hemerie and J. P. Katoen, *Bottom-up tree acceptors*, Sci. Comput. Programming, 13 (1990), pp. 51–72.
[17] U. Heuter, *First-order properties of finite trees, star-free expressions and aperiodicity*, in Proc. 5th Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Comput. Sci. 294, Springer-Verlag, Berlin, New York, Heidelberg, 1988, pp. 136–149.
[18] U. Heuter, *Definite tree languages*, Bull. European Assoc. Theoret. Comput. Sci., 35 (1988), pp. 137–142.
[19] P. Mishra, *Towards a theory of types in prolog*, in Proc. 1st IEEE Symposium on Logic Programming, IEEE Computer Society Press, Los Alamitos, CA, 1984, pp. 456–461.
[20] D. E. Muller, *Notes on the theory of automata*, manuscript, University of Illinois at Urbana–Champaign, Champaign, IL, 1987.
[21] D. E. Muller and P. E. Schupp, *Alternating automata on infinite trees*, Theoret. Comput. Sci., 54 (1987), pp. 267–276.
[22] M. Nivat, *Binary tree codes*, in Tree Automata and Languages, M. Nivat and A. Podelski, eds., North–Holland, Amsterdam, 1992, pp. 1–20.
[23] M. Nivat and A. Podelski, *Tree monoids and recognizable sets of trees*, in Resolution of Equations in Algebraic Structures, Vol. I., H. Aït-Kaci and M. Nivat, eds., Academic Press, Boston, 1989.

[24] M. Nivat and A. Podelski, *Another variation on the common subexpression problem*, Discrete Math., 114 (1993), pp. 379–401.
[25] A. E. Pelegri-Llopart, *Rewrite systems, pattern matching, and code generation*, Ph.D. thesis, Technical Report UCB/CSD 88/423, University of California at Berkeley, Berkeley, CA, 1988.
[26] D. Perrin, *Finite automata*, in Handbook of Theoretical Computer Science, Vol. B, J. van Leeuwen, ed., North–Holland, Amsterdam, 1990, pp. 243–309.
[27] P. Péladeu, and A. Podelski, *On reverse and general definite tree languages*, in Proc. International Conference on Automata, Languages and Programming (ICALP), Lecture Notes in Comput. Sci. 623, W. Kuich, ed., Springer-Verlag, Berlin, New York, Heidelberg, 1992, pp. 150–161.
[28] F. Pennig, *Types in Logic Programming*, MIT Press, Cambridge, MA, 1992.
[29] A. Podelski, *Monoïdes d'arbres et automates d'arbres*, thèse de doctorat, Université de Paris VII, Paris, 1989.
[30] J. C. Reynolds, *Automatic computation of data set definition*, Inform. Process., 68 (1969), pp. 456–461.
[31] H. Seidl, *Deciding equivalence of finite tree automata*, SIAM J. Comput., 19 (1990), pp. 424–437.
[32] M. Steinby, *A theory of tree language varieties*, in Tree Automata and Languages, M. Nivat and A. Podelski, eds., North–Holland, Amsterdam, 1992, pp. 57–82.
[33] J. W. Thatcher and J. B. Wright, *Generalized finite automata theory with an application to a decision problem of second-order logic*, Math. Systems Theory, 2 (1967), pp. 57–81.
[34] W. Thomas, *Logical aspects in the study of tree languages*, in 9th Colloquium on Trees in Algebra and in Programming, B. Courcelle, ed., Cambridge University Press, Cambridge, UK, 1984, pp. 31–49.
[35] J. Viragh, *Deterministic ascending tree automata* I, Acta Cybernet., 5 (1981), pp. 33–42.
[36] E. Yardeni and E. Shapiro, *A type system for logic programs*, in Concurrent Prolog, Vol. 2, E. Shapiro, ed., MIT Press, Cambridge, MA, 1987, pp. 211–244.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 59–78, February 1997

004

THRESHOLD COMPUTATION AND CRYPTOGRAPHIC SECURITY*

YENJO HAN†, LANE A. HEMASPAANDRA‡, AND THOMAS THIERAUF§

Abstract. Threshold machines are Turing machines whose acceptance is determined by what portion of the machine's computation paths are accepting paths. Probabilistic machines are Turing machines whose acceptance is determined by the probability weight of the machine's accepting computation paths. In 1975, Simon proved that for unbounded-error polynomial-time machines these two notions yield the same class, PP. Perhaps because Simon's result seemed to collapse the threshold and probabilistic modes of computation, the relationship between threshold and probabilistic computing for the case of bounded error has remained unexplored.

In this paper, we compare the bounded-error probabilistic class BPP with the analogous threshold class, $\mathrm{BPP}_{\mathrm{path}}$, and, more generally, we study the structural properties of $\mathrm{BPP}_{\mathrm{path}}$. We prove that $\mathrm{BPP}_{\mathrm{path}}$ contains both $\mathrm{NP}^{\mathrm{BPP}}$ and $\mathrm{P}^{\mathrm{NP}[\log]}$ and that $\mathrm{BPP}_{\mathrm{path}}$ is contained in $\mathrm{P}^{\Sigma_2^p[\log]}$, $\mathrm{BPP}^{\mathrm{NP}}$, and PP. We conclude that, unless the polynomial hierarchy collapses, bounded-error threshold computation is strictly more powerful than bounded-error probabilistic computation.

We also consider the natural notion of secure access to a database: an adversary who watches the queries should gain no information about the input other than perhaps its length. We show for both BPP and $\mathrm{BPP}_{\mathrm{path}}$ that if there is *any* database for which this formalization of security differs from the security given by oblivious database access, then $\mathrm{P} \neq \mathrm{PSPACE}$. It follows that if any set lacking small circuits can be securely accepted, then $\mathrm{P} \neq \mathrm{PSPACE}$.

Key words. complexity theory, cryptography, probabilistic computation, threshold computation

AMS subject classifications. 68Q15, 94A60

PII. S0097539792240467

1. Introduction.

In 1975, Simon [27] defined threshold machines. A threshold machine is a nondeterministic Turing machine that accepts a given input if more than half of all computation paths on that input are accepting paths. Gill [13] defined the class PP as the class of sets for which there exists a probabilistic polynomial-time Turing machine that accepts exactly the members of the set with probability greater than $\frac{1}{2}$. Simon [27] showed that the class of sets accepted by polynomial-time threshold machines characterizes the unbounded-error probabilistic class PP.

In this paper, we extend the notion of threshold computation to bounded-error probabilistic classes, and we study the degree to which threshold and probabilistic database ("oracle") computations hide information from observers.

In particular, we introduce $\mathrm{BPP}_{\mathrm{path}}$ and $\mathrm{R}_{\mathrm{path}}$ as the threshold analogues of BPP and R [13]. We give evidence that, unlike the case for PP, these threshold classes are different from their probabilistic counterparts. Section 3 studies the

* Received by the editors November 18, 1992; accepted for publication (in revised form) April 13, 1995.

http://www.siam.org/journals/sicomp/26-2/24046.html

† Microtec Research Inc., 2350 Mission College Boulevard, Santa Clara, CA 95054 (yenjo@mri.com). The research of this author was supported in part by NSF grant CCR-9322513. This research was performed while this author was at the University of Rochester.

‡ Department of Computer Science, University of Rochester, Rochester, NY 14627 (lane@cs.rochester.edu). The research of this author was supported in part by NSF grants CCR-8957604, CCR-9322513, INT-9116781/JSPS-ENG-207, and INT-9513368/DAAD-315-PRO-fo-ab.

§ Abteilung Theoretische Informatik, Universität Ulm, Oberer Eselsberg, 89069 Ulm, Germany (thierauf@informatik.uni-ulm.de). The research of this author was supported in part by NSF grants CCR-9057486 and CCR-9322513 and by DFG Postdoctoral Stipend Th 472/1-1. Part of this research was performed while this author was visiting the University of Rochester and Princeton University.

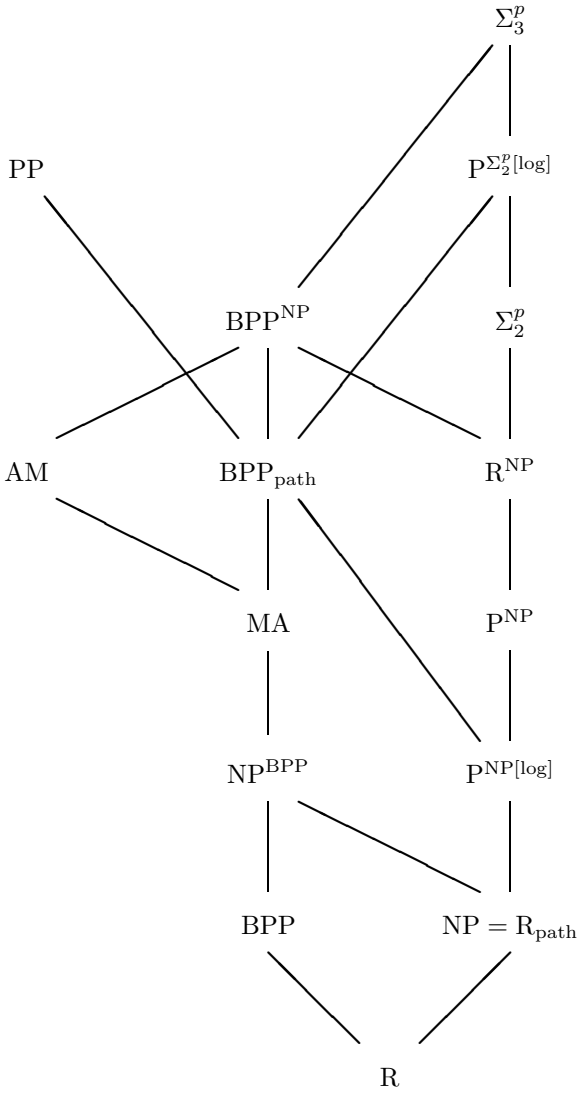


FIG. 1. *Inclusion relations between* $\mathrm{BPP}_{\mathrm{path}}$ *and other complexity classes. Not shown are the inclusions* $\mathrm{BPP} \subseteq \mathrm{R}^{\mathrm{NP}}$ *and* $\mathrm{AM} \subseteq \mathrm{P}^{\Sigma_2^p[\log]}$ *(in fact,* $\mathrm{AM} \subseteq \Pi_2^p$*). As an open problem, we ask whether* $\mathrm{BPP}_{\mathrm{path}}$ *is contained in* Σ_2^p *or even* R^{NP}*. There is an oracle relative to which* $\mathrm{BPP}_{\mathrm{path}}$ *is not in* P^{NP}*.*

properties of the class $\mathrm{BPP}_{\mathrm{path}}$ and its relationship to other complexity classes. For example, we show—in contrast to the BPP case—that $\mathrm{BPP}_{\mathrm{path}}$ is self-low (i.e., ${\mathrm{BPP}_{\mathrm{path}}}^{\mathrm{BPP}_{\mathrm{path}}} = \mathrm{BPP}_{\mathrm{path}}$) only if the polynomial hierarchy collapses. We also show that BPP is low for $\mathrm{BPP}_{\mathrm{path}}$, that there is a relativized world in which $\mathrm{BPP}_{\mathrm{path}}$ does not contain the smallest reasonable counting class, and that $\mathrm{BPP}_{\mathrm{path}}$ has many closure properties. Figure 1 gives an overview of the inclusion relations that we establish between $\mathrm{BPP}_{\mathrm{path}}$ and other complexity classes; in particular, note that though contained in the polynomial hierarchy, $\mathrm{BPP}_{\mathrm{path}}$ contains NP and coNP.

Section 4 studies, for threshold and probabilistic computations that have Turing (that is, adaptive) access to a database, the degree to which the input can be hidden from an observer. In particular, we consider the least restrictive possible notion ensuring that a powerful observer should gain no information about the input other than its length [5]. For the cases of unbounded-error probabilistic and threshold computation, we note that this optimal degree of security can be achieved in all cases. For the cases of bounded-error probabilistic and threshold computations, we prove the following result: if there exists any database D to which secure access yields more power than oblivious access (a notion in which the querying machine—until finished querying—is wholly denied access to the input other than the length of the input [9]), then P $\neq$ PSPACE.

2. Definitions and discussion. Throughout this paper, we use the alphabet $\Sigma = \{0, 1\}$. For a string $x \in \Sigma^*$, $|x|$ denotes the length of x. For a set $A \subseteq \Sigma^*$, $A(x)$ denotes the characteristic function of A, $A^{=n}$ denotes $\{\, y \mid y \in A \text{ and } |y| = n \,\}$, $A^{\leq n}$ denotes $\{\, y \mid y \in A \text{ and } |y| \leq n \,\}$, and $\|A\|$ denotes the cardinality of A. The complement of A is $\overline{A} = \Sigma^* - A$, and for a class $\mathcal{C}$ of sets, $\mathrm{co}\,\mathcal{C} = \{\, \overline{A} \mid A \in \mathcal{C} \,\}$.

Let $(\cdot, \cdot)_b : \Sigma^* \times \Sigma^* \to \Sigma^*$ be a polynomial-time computable, polynomial-time invertible, one-to-one, onto function. For any string z, let $z+1$ denote the string that lexicographically follows z, and for any string $z \neq \epsilon$, let $z-1$ denote the string that lexicographically precedes z. Let k_s be the lexicographically kth string in Σ^*. We define our (multi-arity, onto) pairing function by $(x_1, x_2, \ldots, x_k)$ equals (a) $(\epsilon, \epsilon)_b$ when $k = 0$, (b) $(\epsilon, x_1 + 1)_b$ when $k = 1$, and (c) $(k_s, (x_1, (x_2, (\cdots (x_{k-1}, x_k)_b \cdots)_b)_b)_b)_b$ when $k \geq 2$.

P (NP) denotes the class of languages that are accepted by polynomial-time deterministic (nondeterministic) Turing machines. For nondeterministic Turing machines, we assume without loss of generality that the nondeterministic branching degree is at most two. M is polynomial-normalized (henceforward denoted *normalized*) if there is a polynomial p such that on every input x the machine M makes exactly $p(|x|)$ nondeterministic moves on each computation path. FP is the class of polynomial-time computable functions. One can define relativized classes such as P^{NP} (respectively, $\mathrm{P}^{\mathrm{NP[log]}}$) by employing P machines having some NP oracle that can be asked polynomially (respectively, logarithmically) many queries, i.e., so-called oracle machines [3]. This is called a *Turing reduction* (to NP). If the queries are made nonadaptively (i.e., in parallel), we call this a *truth-table reduction* (see Ladner, Lynch, and Selman [22]). By $\mathrm{P}_{\mathrm{tt}}^{\mathrm{NP}}$ we denote the class of sets that are truth-table reducible to NP—but, in fact, $\mathrm{P}_{\mathrm{tt}}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{NP[log]}}$ [15].

The *polynomial hierarchy* [25], [29] is defined as follows:

$$\begin{aligned} \Sigma_1^p &= \mathrm{NP}, \\ \Sigma_{k+1}^p &= \mathrm{NP}^{\Sigma_k^p} \quad (\text{for } k \in \{1, 2, 3, \ldots\}), \quad \text{and} \\ \mathrm{PH} &= \bigcup_{k \geq 1} \Sigma_k^p. \end{aligned}$$

P/poly [19] denotes the class of sets having small circuits.

For a nondeterministic polynomial-time Turing machine M, let $acc_M(x)$ ($rej_M(x)$) denote the number of accepting (rejecting) paths of M on input x and let $total_M(x)$ denote the total number of paths of M on input x. #P is the class of functions f such that for some nondeterministic polynomial-time Turing machine M it holds that $(\forall x)\,[f(x) = acc_M(x)]$.

2.1. Probabilistic and threshold computation. A *probabilistic polynomial-time Turing machine* [13] is a nondeterministic polynomial-time Turing machine M such that M chooses with equal probability each of the nondeterministic choices at each choice point. $\Pr[M(x) = 1]$ denotes the probability weight of those paths on which M accepts x and $\Pr[M(x) = 0]$ denotes the probability weight of those paths on which M rejects x.

We now define some complexity classes in terms of probabilistic polynomial-time Turing machines.

DEFINITION 2.1 (probabilistic classes).

1. PP [13] *is the class of all sets L such that there exists a probabilistic polynomial-time Turing machine M such that for all $x \in \Sigma^*$ it holds that* $\Pr[M(x) = L(x)] > \frac{1}{2}$.

2. BPP [13] *is the class of all sets L such that there exist a probabilistic polynomial-time Turing machine M and an $\epsilon > 0$ such that for all $x \in \Sigma^*$ it holds that* $\Pr[M(x) = L(x)] > \frac{1}{2} + \epsilon$.

3. R [13] *is the class of all sets L such that there exists a probabilistic polynomial-time Turing machine M such that for all $x \in \Sigma^*$ it holds that*

$$x \in L \Longrightarrow \Pr[M(x) = 1] > \frac{1}{2} \quad \textit{and}$$
$$x \notin L \Longrightarrow \Pr[M(x) = 0] = 1.$$

By definition, we clearly have $\mathrm{R} \subseteq \mathrm{BPP} \subseteq \mathrm{PP}$ [13].

The class PP can also be characterized as the class of sets L such that there exist a nondeterministic polynomial-time Turing machine M and a function $f \in \mathrm{FP}$ such that for all $x \in \Sigma^*$ it holds that $x \in L \iff acc_M(x) \geq f(x)$.

By looking at the portion of accepting paths rather than the probability weight of the accepting paths, we now introduce the threshold analogues of the above probabilistic classes. Let $\#[M(x) = 1]$ denote $acc_M(x)$ and let $\#[M(x) = 0]$ denote $rej_M(x)$.

DEFINITION 2.2 (threshold classes).

1. $\mathrm{PP}_{\mathrm{path}}$ [27] *is the class of all sets L such that there exists a nondeterministic polynomial-time Turing machine M such that for all $x \in \Sigma^*$ it holds that* $\#[M(x) = L(x)] > \frac{1}{2}\, total_M(x)$.

2. $\mathrm{BPP}_{\mathrm{path}}$ *is the class of all sets L such that there exist a nondeterministic polynomial-time Turing machine M and an $\epsilon > 0$ such that for all $x \in \Sigma^*$ it holds that* $\#[M(x) = L(x)] > (\frac{1}{2} + \epsilon)\, total_M(x)$.

3. $\mathrm{R}_{\mathrm{path}}$ *is the class of all sets L such that there exists a nondeterministic polynomial-time Turing machine M such that for all $x \in \Sigma^*$ it holds that*

$$x \in L \Longrightarrow acc_M(x) > \frac{1}{2}\, total_M(x) \quad \textit{and}$$
$$x \notin L \Longrightarrow rej_M(x) = total_M(x).$$

It is easy to see that $\mathrm{R}_{\mathrm{path}} \subseteq \mathrm{BPP}_{\mathrm{path}} \subseteq \mathrm{PP}_{\mathrm{path}}$. For all threshold classes in this paper, as a notational convenience we will place oracles above the word "path" (e.g., $\mathrm{BPP}_{\mathrm{path}}^{\mathrm{BPP}}$ denotes $(\mathrm{BPP}_{\mathrm{path}})^{\mathrm{BPP}}$).

It is known that R, BPP, and PP sets can be accepted via normalized probabilistic polynomial-time Turing machines: just extend each computation path of a given machine up to a fixed polynomial length and, on each new path, accept if the path that was extended accepted and reject otherwise. The modified machine has the same

acceptance probability as the original one. Observe that for normalized machines, the probabilistic interpretation of the machine accepts the same set as the threshold interpretation of the machine. Thus each of the probabilistic classes is contained in the corresponding threshold class, i.e., $\mathrm{PP} \subseteq \mathrm{PP}_{\mathrm{path}}$, $\mathrm{BPP} \subseteq \mathrm{BPP}_{\mathrm{path}}$, and $\mathrm{R} \subseteq \mathrm{R}_{\mathrm{path}}$.

In fact, Simon [27] has already shown that $\mathrm{PP}_{\mathrm{path}}$ is not a bigger class than PP. For completeness, we give a proof here.

THEOREM 2.3 ([27]). $\mathrm{PP}_{\mathrm{path}} = \mathrm{PP}$.

Proof. It suffices to show that $\mathrm{PP}_{\mathrm{path}} \subseteq \mathrm{PP}$. Let $L \in \mathrm{PP}_{\mathrm{path}}$ via $\mathrm{PP}_{\mathrm{path}}$ machine M with polynomial q bounding M's runtime. Consider the machine M' that on input x extends each path y of M by appending a full binary subtree of depth $q(|x|) - |y|$. Furthermore, on the leftmost path of this appended subtree, M' branches into two accepting (rejecting) paths if M accepted (rejected) on the path y. On each remaining path of the subtree, M' branches into one accepting and one rejecting path.

M' on input x has $2^{q(|x|)+1}$ paths and of these $2^{q(|x|)} + acc_M(x) - rej_M(x)$ are accepting paths. This shows that $L \in \mathrm{PP}$ via M'. □

Interestingly, this equivalence between probabilistic and threshold classes cannot hold for R and BPP unless the polynomial hierarchy collapses to its second level. This follows from the fact that NP is contained in $\mathrm{R}_{\mathrm{path}}$ and thus is also contained in $\mathrm{BPP}_{\mathrm{path}}$.

PROPOSITION 2.4. $\mathrm{R}_{\mathrm{path}} = \mathrm{NP}$.

Proof. $\mathrm{R}_{\mathrm{path}} \subseteq \mathrm{NP}$ is immediate from the definition. For the reverse inclusion, let M be a nondeterministic Turing machine and let polynomial p bound the runtime of M.

Consider the machine M' that on input x first simulates M on input x, and if the simulation ends in an accepting path y, then M' appends $2^{p(|x|)+1}$ accepting paths to y and otherwise M' rejects.

Now more than half of all paths of M' on input x are accepting if $x \in L(M)$, and M' has no accepting paths otherwise. This shows that $L(M) \in \mathrm{R}_{\mathrm{path}}$. □

COROLLARY 2.5. $\mathrm{NP} \subseteq \mathrm{BPP}_{\mathrm{path}}$.

It follows that if $\mathrm{BPP}_{\mathrm{path}}$ is equal to BPP, then $\mathrm{BPP}_{\mathrm{path}}$, and hence NP, has small circuits, which in turn, by the result of Karp, Lipton, and Sipser (see [19]), implies that the polynomial hierarchy collapses. Thus we cannot expect $\mathrm{BPP}_{\mathrm{path}}$ to have normalized machines. For different, contemporaneous work related to normalized versus nonnormalized computation, see Hertrampf et al. [16] and Jenner, McKenzie, and Thérien [17].

We have now seen that there are some crucial differences between BPP and its threshold analogue, $\mathrm{BPP}_{\mathrm{path}}$. We will study $\mathrm{BPP}_{\mathrm{path}}$ in more detail in section 3 and, especially, we will strengthen Corollary 2.5.

2.2. Secure computation. In this subsection and in section 4, we study notions of secure adaptive access to databases in the presence of a powerful spying observer. We give below what we feel are the most natural definitions. In these definitions, we obtain security by requiring that an observer (seeing a path drawn uniformly from all the machine's paths) should learn nothing about the input string other than perhaps its length. For threshold computation, this notion is new. For probabilistic computation, the appendix proves that this definition is equivalent to the notion of "one-oracle instance-hiding schemes that leak at most the length of their inputs" [5]. The original motivation for such classes, as explained, for example, by Beaver and Feigenbaum [5], is, very roughly, to study whether weak devices can solve hard problems by asking some powerful device questions in such a way that no observer can tell which problem

was actually solved by the weak device. Since NP $\subseteq$ $\mathrm{BPP}_{\mathrm{path}}$, $\mathrm{BPP}_{\mathrm{path}}$ is clearly not a computationally weak class. It nonetheless makes sense to consider the same interactive model in the case that applies here: studying whether a relatively powerful class ($\mathrm{BPP}_{\mathrm{path}}$) can use a (potentially powerful) information source while shielding information on the problem being solved even from extremely powerful observers.

DEFINITION 2.6 (secure threshold computation). *For any set D, a set A is said to be in* secure$\mathrm{BPP}^D_{\mathrm{path}}$ *(that is, is said to be "securely accepted by a bounded-error threshold polynomial-time machine via access to database D") if there is a nondeterministic polynomial-time Turing machine N such that the following hold:*

1. *[$A \in \mathrm{BPP}^D_{\mathrm{path}}$ via machine N.] There exists an $\epsilon > 0$ such that for all $x \in \Sigma^*$ it holds that $\#[N^D(x) = A(x)] > (\frac{1}{2} + \epsilon)\, total_{N^D}(x)$ (see part 2 of Definition 2.2).*

2. *[The queries of N^D reveal no information to an observer other than perhaps the length of the input.] For every $k \in \{0, 1, 2, \ldots\}$, every vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_k)$, $v_1, v_2, \ldots, v_k \in \Sigma^*$, and every pair of strings $x \in \Sigma^*$ and $y \in \Sigma^*$ such that $|x| = |y|$, it holds that*

$$\frac{path\text{-}occurences_{N^D(x)}(\boldsymbol{v})}{total_{N^D}(x)} = \frac{path\text{-}occurences_{N^D(y)}(\boldsymbol{v})}{total_{N^D}(y)},$$

where $path\text{-}occurences_{N^D(z)}(\boldsymbol{v}) = ||\{p \mid p$ is a path of $N^D(z)$ on which $\boldsymbol{v}$ is the sequence of queries asked to the oracle (in the order asked, possibly with duplications if the same query is asked more than once)[1]$\}||$.

Similarly, for the probabilistic class BPP, we have the following definition of secure access.

DEFINITION 2.7 (secure probabilistic computation). *For any set D, a set A is said to be in* secureBPP^D *(that is, is said to be "securely accepted by a bounded-error probabilistic polynomial-time machine via access to database D") if there is a probabilistic polynomial-time Turing machine N such that the following hold:*

1. *[$A \in \mathrm{BPP}^D$ via machine N.] There exists an $\epsilon > 0$ such that for all $x \in \Sigma^*$ it holds that $\Pr[N^D(x) = A(x)] > \frac{1}{2} + \epsilon$ (see part 2 of Definition 2.1).*

2. *[The queries of N^D reveal no information to an observer other than perhaps the length of the input.] For every $k \in \{0, 1, 2, \ldots\}$, every vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_k)$, $v_1, v_2, \ldots, v_k \in \Sigma^*$, and every pair of strings $x \in \Sigma^*$ and $y \in \Sigma^*$ such that $|x| = |y|$, it holds that*

$$\Pr[\text{the query vector of } N^D(x) \text{ is } \boldsymbol{v}] = \Pr[\text{the query vector of } N^D(y) \text{ is } \boldsymbol{v}].$$

Oblivious self-reducibility was discussed in [9], and we now define complexity classes capturing the notion of oblivious access.

DEFINITION 2.8 (oblivious probabilistic and threshold classes). *For any set D, a set A is said to be in* oblivious$\mathrm{BPP}^D_{\mathrm{path}}$ *(respectively,* obliviousBPP^D*) if there is a nondeterministic (respectively, probabilistic) polynomial-time Turing machine N such that the following hold:*

1. *[$A \in \mathrm{BPP}^D_{\mathrm{path}}$ (respectively, $A \in \mathrm{BPP}^D$) via machine N.] There exists an $\epsilon > 0$ such that for all $x \in \Sigma^*$ it holds that $\#[N^D(x) = A(x)] > (\frac{1}{2} + \epsilon)\, total_{N^D}(x)$ (respectively, $\Pr[N^D(x) = A(x)] > \frac{1}{2} + \epsilon$).*

2. *N is an oblivious machine in the sense that on an input z it initially is given access to a "preinput" tape on which $0^{|z|}$ is written. N then performs its adaptive queries to D. Then after making all queries to D, machine N is given access to z.*

[1] Henceforth, we will refer to this as a *query vector*.

We clearly have that, for every D,

$$\mathrm{BPP}^D_{\mathrm{path}} \supseteq \mathrm{secureBPP}^D_{\mathrm{path}} \supseteq \mathrm{obliviousBPP}^D_{\mathrm{path}} \quad \text{and}$$
$$\mathrm{BPP}^D \supseteq \mathrm{secureBPP}^D \supseteq \mathrm{obliviousBPP}^D.$$

Are these inclusions proper? In other words, does using security against observers as the *definition* of secure computation ($\mathrm{secureBPP}^D_{\mathrm{path}}$, $\mathrm{secureBPP}^D$) yield a more flexible notion of security than does blinding the machine to its input ($\mathrm{obliviousBPP}^D_{\mathrm{path}}$, $\mathrm{obliviousBPP}^D$)? Formally, is $\mathrm{obliviousBPP}^D_{\mathrm{path}} \neq \mathrm{secureBPP}^D_{\mathrm{path}}$ or $\mathrm{obliviousBPP}^D \neq \mathrm{secureBPP}^D$? Our intuition says that both inequalities hold. However, section 4 shows that establishing that "yes" is the answer implies that P $\neq$ PSPACE (and even implies the stronger result that BPP $\neq$ PP). Since it is commonly believed that P $\neq$ PSPACE, this does not provide evidence that equality holds; rather, it merely suggests that witnessing a separation will be hard with current techniques. We note that results (such as Theorem 4.1 and Corollary 4.2) that connect the existence of an oracle separation to the existence of a real-world separation (see, e.g., the survey [7]) usually occur in cases in which the oracle is tremendously restricted (e.g., to the class of tally sets or the class of sparse sets [4], [24]); in contrast, section 4 provides such a relativization result that applies without restriction of the database D.

Note that we could also define classes partially-secure-$\mathrm{BPP}^D_{\mathrm{path}}$ and partially-secure-BPP^D based on the notion (see, e.g., [9] and the papers cited therein) that an observer watching *one* query should get no information other than perhaps about the length (clearly, for all D, $\mathrm{BPP}^D_{\mathrm{path}} \supseteq$ partially-secure-$\mathrm{BPP}^D_{\mathrm{path}} \supseteq \mathrm{secureBPP}^D_{\mathrm{path}}$ and $\mathrm{BPP}^D \supseteq$ partially-secure-$\mathrm{BPP}^D \supseteq \mathrm{secureBPP}^D$), and, more generally, one could study a variety of classes between BPP^D and $\mathrm{secureBPP}^D$ (or between $\mathrm{BPP}^D_{\mathrm{path}}$ and $\mathrm{secureBPP}^D_{\mathrm{path}}$) based upon security against observers using various strengths of query access. (For example, one could require security against observers who could see two queries, or against observers who could make $\mathcal{O}(\log n)$ adaptive queries into the query vector, or so on.) However, we restrict our attention to what we feel are the most natural security classes: $\mathrm{secureBPP}^D$ and $\mathrm{secureBPP}^D_{\mathrm{path}}$.

There is no point in defining security classes for unbounded-error computation since it is easy to see that, for every D, $\mathrm{PP}^D = \mathrm{securePP}^D = \mathrm{obliviousPP}^D = \mathrm{PP}^D_{\mathrm{path}} = \mathrm{securePP}^D_{\mathrm{path}} = \mathrm{obliviousPP}^D_{\mathrm{path}}$.

Finally, we note that all sets that are accepted by an oblivious machine relative to some database D have small circuits. Let $\mathrm{obliviousBPP}^\star$ denote $\bigcup_{D \in 2^{\Sigma^*}} \mathrm{obliviousBPP}^D$. We have the following result.

PROPOSITION 2.9. $\mathrm{obliviousBPP}^\star = \mathrm{P/poly}$.

COROLLARY 2.10. $(\exists L)\,[L \notin \mathrm{obliviousBPP}^L]$.

Though for most common classes $\mathcal{C}$ it holds that $(\forall L)\,[L \in \mathcal{C}^L]$, Corollary 2.10 should not be surprising; it is natural that weak machines, when accepting a hard set via a hard database, may leak some information to an observer. Interestingly, a similar result holds for secure computation. Namely, Abadi, Feigenbaum, and Kilian [1] have shown that $\mathrm{secureBPP}^D \subseteq \mathrm{NP/poly} \cap \mathrm{coNP/poly}$ for any database D. Thus for any set D, no NP-hard set is in $\mathrm{secureBPP}^D$ unless the polynomial hierarchy collapses.

3. $\mathbf{BPP_{path}}$. We have already argued that BPP and $\mathrm{BPP}_{\mathrm{path}}$ differ unless the polynomial hierarchy collapses. These classes nonetheless share certain properties. For example, as is also the case for BPP [36], $\mathrm{BPP}_{\mathrm{path}}$ has a strong amplification property.

THEOREM 3.1. *Let L be in $\mathrm{BPP}_{\mathrm{path}}$. For each polynomial q, there is a nondeterministic polynomial-time Turing machine M such that for all $x \in \Sigma^*$ it holds that*

$$\#[M(x) = L(x)] > \left(1 - 2^{-q(|x|)}\right) \mathit{total}_M(x).$$

The proof is analogous to the corresponding proof for BPP.

BPP is closed under Turing reductions [20], [34]. However, no relativizable proof can establish the closure of $\mathrm{BPP}_{\mathrm{path}}$ under Turing reductions. In particular, Beigel [6] constructed an oracle relative to which P^{NP} is not contained in PP. Since $\mathrm{BPP}_{\mathrm{path}}$ is clearly contained in PP (and the proof relativizes), it follows that, relative to the same oracle, P^{NP} is not contained in $\mathrm{BPP}_{\mathrm{path}}$, and hence, since $\mathrm{NP} \subseteq \mathrm{BPP}_{\mathrm{path}}$, $\mathrm{BPP}_{\mathrm{path}}$ is not closed under Turing reductions relative to this oracle. That is, there exists an A such that $\mathrm{BPP}^A_{\mathrm{path}} \neq \mathrm{P}^{\mathrm{BPP}^A_{\mathrm{path}}}$.

For $\mathrm{BPP}_{\mathrm{path}}$, we can prove closure under truth-table reductions.

THEOREM 3.2. $\mathrm{BPP}_{\mathrm{path}}$ *is closed under polynomial-time truth-table reductions.*

Proof. Let $A \leq^p_{tt} B$ for $B \in \mathrm{BPP}_{\mathrm{path}}$, i.e., there exists a polynomial-time Turing machine M such that $L = L(M^B)$ and, for each input x of length n, machine M makes at most $q(n)$ queries (nonadaptively) to B. Without loss of generality, we may assume that all queries have the same length $l(n)$, $l(n) \geq n$, and that $q(n)$ is a nondecreasing function.

Let N be a $\mathrm{BPP}_{\mathrm{path}}$ machine for B such that on input y,

$$\#[N(y) = B(y)] > \left(1 - \frac{1}{3q(|y|)}\right) \mathit{total}_N(y).$$

Consider the machine M' that on input x, $|x| = n$, computes $y_1, \ldots, y_k$, the truth-table queries of M on input x, where $k \leq q(n)$, and for each query y_i, machine M' guesses a path of N on input y_i and takes the output of this path as the answer to query y_i, for $i = 1, \ldots, k$. Using these answers instead of the oracle B, M' simulates M on input x and outputs the result.

M' has $\mathit{total}_{M'}(x) = \prod_{i=1}^k \mathit{total}_N(y_i)$ paths. At least on those paths on which all the answers to the oracle queries are correct, M' decides correctly whether x is in A, i.e., we have

$$\begin{aligned}
\#[M'(x) = A(x)] &\geq \prod_{i=1}^k \#[N(y_i) = B(y_i)] \\
&\geq \prod_{i=1}^k \left(1 - \frac{1}{3q(l(n))}\right) \mathit{total}_N(y_i) \quad \text{by assumption} \\
&\geq \left(1 - \frac{1}{3q(n)}\right)^k \prod_{i=1}^k \mathit{total}_N(y_i) \quad \text{since } l(n) \geq n \\
&\geq \left(1 - \frac{k}{3q(n)}\right) \prod_{i=1}^k \mathit{total}_N(y_i) \\
&\geq \frac{2}{3} \mathit{total}_{M'}(x).
\end{aligned}$$

This shows that $A \in \mathrm{BPP}_{\mathrm{path}}$. □

COROLLARY 3.3. $\mathrm{BPP}_{\mathrm{path}}$ *is closed under complementation, intersection, and union.*

Since NP is contained in $\mathrm{BPP}_{\mathrm{path}}$, it follows that the closure of NP under truth-table reductions is contained in $\mathrm{BPP}_{\mathrm{path}}$.

COROLLARY 3.4. $\mathrm{P}^{\mathrm{NP[log]}} \subseteq \mathrm{BPP}_{\mathrm{path}}$.

It is known that BPP is low for PP [21] and for itself [20], [34], i.e., $\mathrm{PP}^{\mathrm{BPP}} = \mathrm{PP}$ and $\mathrm{BPP}^{\mathrm{BPP}} = \mathrm{BPP}$. We show in the next theorem that BPP is also low for $\mathrm{BPP}_{\mathrm{path}}$. Observe that relative to Beigel's previously mentioned oracle making P^{NP} not contained in PP, we must also have that NP, and hence $\mathrm{BPP}_{\mathrm{path}}$, cannot be low for PP. That is, there exists an A such that $\mathrm{PP}^{\mathrm{BPP}^{A}_{\mathrm{path}}} \neq \mathrm{PP}^{A}$. Furthermore, by an easy induction, we have that if $\mathrm{BPP}_{\mathrm{path}}$ is low for itself then the polynomial hierarchy, PH, is contained in $\mathrm{BPP}_{\mathrm{path}}$. However, as we will see in Theorem 3.11 below, $\mathrm{BPP}_{\mathrm{path}}$ is contained in some level of the polynomial hierarchy. Thus $\mathrm{BPP}_{\mathrm{path}}$ is not low for $\mathrm{BPP}_{\mathrm{path}}$ unless the polynomial hierarchy collapses.

THEOREM 3.5. $\mathrm{BPP}^{\mathrm{BPP}}_{\mathrm{path}} = \mathrm{BPP}_{\mathrm{path}}$.

Proof. Let $L \in \mathrm{BPP}^{\mathrm{BPP}}_{\mathrm{path}}$ via a machine M and a set $A \in \mathrm{BPP}$ such that polynomial p bounds the runtime of M^A and for all $x \in \Sigma^*$ it holds that $\#[M^A(x) = L(x)] > \frac{7}{8}\ total_{M^A}(x)$.

Let $B = \{\, (0^n, w_1, a_1, \ldots, w_k, a_k) \mid k \leq p(n) \text{ and } (\forall i : 1 \leq i \leq k)\, [|w_i| \leq p(n) \text{ and } A(w_i) = a_i] \,\}$. Since BPP is closed under truth-table reductions [20], [34], $B \in \mathrm{BPP}$. Hence there exist a probabilistic polynomial-time Turing machine M_B and a polynomial q such that for any input $z = (0^n, w_1, a_1, \ldots, w_k, a_k)$, M_B's error probability is bounded by $2^{-(p(n)+4)}$ and M_B's computation tree is a full binary tree with $total_{M_B}(z) = 2^{q(n)}$.

Consider the machine M' that on input x, $|x| = n$, performs the following steps.

1. M' simulates M^A on input x. Whenever M queries the oracle, M' nondeterministically guesses the answer. Let $(w_1, a_1), \ldots, (w_k, a_k)$ be the sequence of queried strings and guessed answers along a computation path y.

2. To verify the guessed answers, M' simulates M_B on input $(0^n, w_1, a_1, \ldots, w_k, a_k)$.

3. M' amplifies the output of M on path y from the first step if the guessed answers there are certified in the second step. More precisely, M' now appends $2^{p(n)+4}$ accepting (rejecting) paths if path y was accepting (rejecting) and the simulation in the second step ended in an accepting path of M_B. Otherwise, M' rejects.

After the first two steps, M' has at most $2^{p(n)}\, 2^{q(n)}$ computation paths. In the last step, M' amplifies all paths (a) in which the guessed oracle answers are correct and that are certified by M_B in the second step, i.e., at most $total_{M^A}(x)\, 2^{q(n)}$ paths, and (b) all paths in which the guessed oracle answers are false but are wrongly certified by M_B, i.e., at most $2^{p(n)}\, 2^{-(p(n)+4)}\, 2^{q(n)}$ paths. Thus we have

$$total_{M'}(x) \leq 2^{p(n)}\, 2^{q(n)} \ + \ 2^{p(n)+4} \left(total_{M^A}(x)\, 2^{q(n)} \ + \ 2^{p(n)}\, 2^{-(p(n)+4)}\, 2^{q(n)} \right).$$

The paths on which M' decides correctly include at least those paths that correspond to correct paths of M in the first step and are subsequently certified in the second step. Since these paths are amplified in the last step, we have

$$\#[M'(x) = L(x)] \geq \left(\frac{7}{8}\ total_{M^A}(x) \right) \left((1 - 2^{-(p(n)+4)})\, 2^{q(n)} \right)\ 2^{p(n)+4}.$$

Now it is not hard to see that $\#[M'(x) = L(x)] > \frac{2}{3}\ total_{M'}(x)$. Thus $L \in \mathrm{BPP}_{\mathrm{path}}$. ∎

If we define a function class $\mathrm{FBPP}_{\mathrm{path}}$ in the natural manner (see the analogous class FBPP of Ko [20]), then it is not hard to see that the same proof technique also establishes that $\mathrm{FBPP}_{\mathrm{path}}^{\mathrm{BPP}} = \mathrm{FBPP}_{\mathrm{path}}$.

COROLLARY 3.6. $\mathrm{NP}^{\mathrm{BPP}} \subseteq \mathrm{BPP}_{\mathrm{path}}$.

Indeed, we even have $\mathrm{P}^{\mathrm{NP}^{\mathrm{BPP}}[\log]\oplus\mathrm{BPP}} \subseteq \mathrm{BPP}_{\mathrm{path}}$.

Babai [2] introduced the Arthur–Merlin classes MA and AM. It is known that $\mathrm{NP}^{\mathrm{BPP}} \subseteq \mathrm{MA} \subseteq \mathrm{AM} \subseteq \mathrm{BPP}^{\mathrm{NP}}$ [2], [35]. It is not known whether any of the inclusions is strict or not, though various relevant oracle separations are known (e.g., Fenner et al. [11] have constructed an oracle world in which $\mathrm{NP}^{\mathrm{BPP}}$ and MA differ). Below, we strengthen Corollary 3.6 to show that even MA is contained in $\mathrm{BPP}_{\mathrm{path}}$. This improves the result of Vereshchagin [33] that $\mathrm{MA} \subseteq \mathrm{PP}$.

THEOREM 3.7. $\mathrm{MA} \subseteq \mathrm{BPP}_{\mathrm{path}}$.

Proof. Let $L \in \mathrm{MA}$. By standard amplification techniques, there exist a polynomial-time predicate Q and polynomials p and q such that for all $x \in \Sigma^*$,

$$x \in L \Longrightarrow (\exists y \in \Sigma^{p(|x|)})\ [\Pr[Q(x,y,z)] > 1 - 2^{-(p(|x|)+4)}],$$
$$x \notin L \Longrightarrow (\forall y \in \Sigma^{p(|x|)})\ [\Pr[Q(x,y,z)] < 2^{-(p(|x|)+4)}],$$

where the probability is taken uniformly over all $z \in \Sigma^{q(|x|)}$.

Consider the machine M that on input x guesses $y \in \Sigma^{p(|x|)}$ and $z \in \Sigma^{q(|x|)}$, and if $Q(x,y,z)$ is false, M rejects; otherwise, M produces $2^{p(|x|)+2}$ accepting paths.

It is not difficult to see that $\#[M(x) = L(x)] > \frac{2}{3}\ total_M(x)$. Thus $L \in \mathrm{BPP}_{\mathrm{path}}$. □

It is an open question whether AM is contained in $\mathrm{BPP}_{\mathrm{path}}$. Vereshchagin [33] constructed an oracle A such that relative to A the class AM is not a subset of PP, i.e., $\mathrm{AM}^A \not\subseteq \mathrm{PP}^A$. Thus AM is not a subset of $\mathrm{BPP}_{\mathrm{path}}$ relative to A. On the other hand, $\mathrm{BPP}_{\mathrm{path}}$ is not a subset of AM unless the polynomial hierarchy collapses. This follows from the result of Boppana, Håstad, and Zachos [8] that if $\mathrm{coNP} \subseteq \mathrm{AM}$, then the polynomial hierarchy collapses to its second level. Since $\mathrm{coNP} \subseteq \mathrm{BPP}_{\mathrm{path}}$, we get the same consequence from the assumption that $\mathrm{BPP}_{\mathrm{path}}$ is contained in AM.

Sipser and Gács [28] (see also [23]) showed that $\mathrm{BPP} \subseteq \mathrm{R}^{\mathrm{NP}}$. It is an open question whether the same inclusion holds for $\mathrm{BPP}_{\mathrm{path}}$. However, we show that $\mathrm{BPP}_{\mathrm{path}} \subseteq \mathrm{BPP}^{\mathrm{NP}}$. As a first step, we show that a $\mathrm{BPP}_{\mathrm{path}}$ set can be decided by a deterministic polynomial-time Turing machine making logarithmically many queries to a Σ_2^p oracle, and hence $\mathrm{BPP}_{\mathrm{path}}$ is in the polynomial hierarchy. A randomized version of this algorithm can decide a $\mathrm{BPP}_{\mathrm{path}}$ set with an NP oracle. The proof applies Sipser's Coding Lemma for universal hashing [28].

We mention that we could get a shorter proof by applying the results of Stockmeyer [30] to approximate #P functions and those of Jerrum, Valiant, and Vazirani [18], who showed a probabilistic version of Stockmeyer's theorem. However, we prefer to give a self-contained proof here, thereby encouraging the reader to see whether he or she can improve our result, for example, by getting a one-sided error probabilistic algorithm (in part 2 of Theorem 3.11). Since there is an oracle relative to which BPP is not contained in P^{NP} [30], one cannot obtain a deterministic algorithm with relativizable techniques.

DEFINITION 3.8 ([28]). *Let $X \subseteq \Sigma^m$ and let $H_1, \ldots, H_k : \Sigma^m \to \Sigma^k$ be a collection of linear functions given as $k \times m$ 0–1 matrices. The predicates Separate and Hash are defined as follows.*

1. *Separate*$_X(H_1, \ldots, H_k) \iff (\forall\, y \in X)\,(\exists\, i : 1 \le i \le k)\,(\forall\, z \in X : y \neq$

$z)\,[H_i(y) \neq H_i(z)]$, *where* $H_i(y)$ *means multiplication of the* $k \times m$ *matrix* H_i *with the* m *vector* y, *yielding a* k *vector, with the arithmetic done in GF*[2].

2. $\mathit{Hash}_X(k) \iff (\exists H_1, \ldots, H_k \in \Sigma^{km})\,[\mathit{Separate}_X(H_1, \ldots, H_k)]$.

The intuition about predicate *Hash* is that the size of the range of the hash functions (which is determined by k) has to be sufficiently large with respect to the size of X for a collection $H_1, \ldots, H_k$ that separates X to exist.

LEMMA 3.9 ([28]). *Let* $X \subseteq \Sigma^m$ *and let* $k = \lfloor \log \|X\| \rfloor + 2$. *For a random collection of functions* $H_1, \ldots, H_k : \ \Sigma^m \to \Sigma^k$,

$$\Pr[\mathit{Separate}_X(H_1, \ldots, H_k)] \ \geq \ \frac{7}{8}.$$

As a consequence of this lemma, we get a lower bound for the size of a set X. The upper bound follows by the pigeonhole principle (see [30]).

COROLLARY 3.10 ([28]). *If* $X \subseteq \Sigma^m$ *and* k_X *is the smallest* k *such that* $\mathit{Hash}_X(k)$ *is true, then* $2^{k_X - 3} \leq \|X\| \leq k_X\, 2^{k_X}$.

THEOREM 3.11.

1. $\mathrm{BPP}_{\mathrm{path}} \subseteq \mathrm{P}^{\Sigma_2^p[\log]}$.
2. $\mathrm{BPP}_{\mathrm{path}} \subseteq \mathrm{BPP}^{\mathrm{NP}}$.

Proof. Let $L \in \mathrm{BPP}_{\mathrm{path}}$. There exist a nondeterministic Turing machine M and a polynomial p that bounds the runtime of M such that for all $x \in \Sigma^*$ it holds that $\#[M(x) = L(x)] > (1 - 2^{-|x|})\ \mathit{total}_M(x)$.

Sipser's proof that BPP $\subseteq \Sigma_2^p$ uses the fact that $\mathit{total}_M(x)$ is known a priori. However, here we have only an upper bound.

Fix $x \in \Sigma^*$; let n denote $|x|$. Define

$$A = \{y\ 0^{p(n)-|y|} \mid y \text{ is an accepting computation of } M \text{ on input } x\} \quad \text{and}$$
$$R = \{y\ 0^{p(n)-|y|} \mid y \text{ is a rejecting computation of } M \text{ on input } x\}.$$

Clearly, $\|A\| = \mathit{acc}_M(x)$ and $\|R\| = \mathit{rej}_M(x)$.

Observe that *Separate* is a coNP predicate in x and the hash functions $H_1, \ldots, H_k$ when applied to A or R, and *Hash* is a Σ_2^p predicate in x and k.

Let k_A (k_R) denote the minimal k such that $\mathit{Hash}_A(k)$ ($\mathit{Hash}_R(k)$) is true. k_A and k_R can be computed by a binary search making at most $\log p(n)$ many queries to Hash_A and $\mathit{Hash}_R(k)$, respectively. From Corollary 3.10, it follows that $2^{k_A - 3} \leq \mathit{acc}_M(x) \leq k_A\, 2^{k_A}$ and that $2^{k_R - 3} \leq \mathit{rej}_M(x) \leq k_R\, 2^{k_R}$. Now it is not difficult to see that, for all but finitely many x, we have $x \in L \iff k_R < k_A$. This proves $L \in \mathrm{P}^{\Sigma_2^p[\log]}$.

Next, we show that $L \in \mathrm{BPP}^{\mathrm{NP}}$. Consider the following probabilistic procedure, which tries to approximate k_A and k_R by randomly generating a collection of functions $H_1, \ldots, H_k$ and directly asking the oracle $\mathit{Separate}_X$ about $(H_1, \ldots, H_k)$, for a given set X and increasing k.

```
APPROXIMATE(x, X)
k ← 0
repeat
    k ← k + 1
    randomly choose H_1, ..., H_k
until Separate_X(H_1, ..., H_k) or k = p(n)
return k.
```

The following main algorithm decides whether x is in L, and is correct with high probability.

MAIN(x)
$k_a \leftarrow$ APPROXIMATE(x, A)
$k_r \leftarrow$ APPROXIMATE(x, R)
if $k_a > k_r$ **then** accept
else reject.

By the definition of k_A, we always have $k_A \leq k_a$. Note that by the upper bound of Corollary 3.10 and since $k_A \leq p(n)$, it follows that $\log(\|A\|/p(n)) \leq k_A$. From Lemma 3.9, it follows that $k_a \leq \lfloor \log \|A\| \rfloor + 2$ holds with probability at least $\frac{7}{8}$. Since the same bounds hold for k_r, we have that with probability at least $\frac{3}{4}$ it holds that both (a) $\log \frac{acc_M(x)}{p(n)} \leq k_a \leq \log acc_M(x)+2$ and (b) $\log \frac{rej_M(x)}{p(n)} \leq k_r \leq \log rej_M(x)+2$. This implies that for all but finitely many x it holds that $x \in L \iff k_a > k_r$, with probability at least $\frac{3}{4}$. Thus $L \in \mathrm{BPP}^{\mathrm{NP}}$. □

As already mentioned just before Theorem 3.5, $\mathrm{BPP}_{\mathrm{path}}$ cannot be low for itself unless the polynomial hierarchy collapses to $\mathrm{BPP}_{\mathrm{path}}$. From Theorem 3.11, we thus have the following claim (see also the discussion just before Theorem 3.5).

COROLLARY 3.12. *If* $\mathrm{BPP}_{\mathrm{path}}^{\mathrm{BPP}_{\mathrm{path}}} = \mathrm{BPP}_{\mathrm{path}}$, *then* $\mathrm{PH} = \mathrm{P}^{\Sigma_2^p[\log]} = \mathrm{BPP}_{\mathrm{path}}$.

Zachos [35] has shown that NP $\subseteq$ BPP implies PH = BPP. Since this result relativizes (i.e., for all A, $\mathrm{NP}^A \subseteq \mathrm{BPP}^A$ implies $\mathrm{PH}^A = \mathrm{BPP}^A$), we obtain the following corollary from Theorem 3.11.

COROLLARY 3.13. $\Sigma_2^p \subseteq \mathrm{BPP}_{\mathrm{path}} \Longrightarrow \mathrm{PH} = \mathrm{BPP}^{\mathrm{NP}}$.

Toda [31] and Toda and Ogiwara [32] showed that $\mathrm{PH} \subseteq \mathrm{BPP}^{\mathcal{C}}$ for any class $\mathcal{C}$ among $\{\mathrm{PP}, \mathrm{C_{=}P}, \oplus\mathrm{P}\}$. As a consequence, none of these classes can be contained in the polynomial hierarchy unless the polynomial hierarchy collapses. Thus none of these classes can be contained in $\mathrm{BPP}_{\mathrm{path}}$ unless the polynomial hierarchy collapses.

Ogiwara and Hemachandra [26] and Fenner, Fortnow, and Kurtz [10] independently defined the counting class SPP as follows.

DEFINITION 3.14 ([26], [10]). SPP *is the class of all sets L such that there exist a nondeterministic polynomial-time Turing machine M and an* FP *function f such that for all $x \in \Sigma^*$ it holds that*

$$x \in L \Longrightarrow acc_M(x) = f(x) + 1 \quad \text{and}$$
$$x \notin L \Longrightarrow acc_M(x) = f(x).$$

Fenner, Fortnow, and Kurtz [10] argue that SPP is in some sense the smallest class that is definable in terms of the number of accepting and rejecting computations. In particular, SPP is low for PP, $\mathrm{C_{=}P}$, and $\oplus$P [10]. Though it is an open question whether SPP is contained in $\mathrm{BPP}_{\mathrm{path}}$, there is an oracle relative to which this is not the case.[2]

THEOREM 3.15. *There is an oracle A such that* $\mathrm{SPP}^A \not\subseteq \mathrm{BPP}_{\mathrm{path}}^A$.

Proof. Let $M_1, M_2, \ldots$ be an enumeration of nondeterministic polynomial-time Turing machines and let $p_1, p_2, \ldots$ be an enumeration of polynomials such that polynomial p_i bounds the runtime of machine M_i. Without loss of generality, we assume $p_i(n) = n^i + i$. Let $s(i)$, $i = 1, 2, \ldots$, be a sequence of integers defined by $s(1) = 5$ and, for $i > 1$, $s(i+1) = 2^{s(i)}$.

[2] Very recently, Fortnow [12] has improved our result by constructing an oracle relative to which SPP is not contained in the polynomial hierarchy.

We define the test language

$$L(A) = \{1^n \mid (\exists j)\,[n = s(j) \text{ and } \|A^{=n}\| = 2^{n-1}]\}.$$

Below, we will construct a set A such that for every $i \geq 1$, $\|A^{=s(i)}\|$ is either $2^{s(i)-1}$ or $2^{s(i)-1} - 1$. For such an A, we have $L(A) \in \mathrm{SPP}^A$. Furthermore, we will construct A such that, for each $i \geq 1$, at least one of the following requirements holds.

(R1) M_i^A is not a $\mathrm{BPP}^A_{\mathrm{path}}$ machine. That is, there exists an $x \in \Sigma^*$ such that

$$\frac{1}{4}\, total_{M_i^A}(x) \leq acc_{M_i^A}(x) \leq \frac{3}{4}\, total_{M_i^A}(x).$$

(R2) There exists an $n \geq 1$ such that $M_i^A(1^n)$ accepts if and only if $1^n \notin L(A)$.

It follows from Theorem 3.1 that the existence of such an oracle establishes the theorem.

We construct the set A in stages. In stage i, we diagonalize against machine M_i. Initially, $i = 1$ and $A_1 = \emptyset$.

Stage i. Let $n = s(i)$. We will add only strings of length n to A_i. Since $p_j(s(j)) < n$ for all $j < i$, this will not effect the construction done in earlier stages.

Define

$$\mathcal{A} = \{A_i \cup Z \mid Z \subseteq \Sigma^n \text{ and } \|Z\| = 2^{n-1}\} \quad \text{and}$$
$$\mathcal{B} = \{A_i \cup Z \mid Z \subseteq \Sigma^n \text{ and } \|Z\| = 2^{n-1} - 1\}.$$

If there is a set $X \in \mathcal{A} \cup \mathcal{B}$ such that X fulfills requirement (R1), i.e., M_i^X is not a $\mathrm{BPP}^X_{\mathrm{path}}$ machine, then define $A_{i+1} = X$ and go to the next stage. Otherwise, we show that there is a set in $\mathcal{A} \cup \mathcal{B}$ such that requirement (R2) is fulfilled.

Let X be a set such that the number of paths of M_i^X on input 1^n is maximal for all $X \in \mathcal{A} \cup \mathcal{B}$. That is, we have

$$(\star) \qquad (\forall\, Y \in \mathcal{A} \cup \mathcal{B})\ [total_{M_i^Y}(1^n) \leq total_{M_i^X}(1^n)].$$

Suppose $X \in \mathcal{A}$. If $1^n \notin L(M_i^X)$, then we are done since $1^n \in L(X)$. Thus suppose that $1^n \in L(M_i^X)$. For $w \in X \cap \Sigma^n$, define $X_w = X - \{w\}$. By definition, $1^n \notin L(X_w)$. We claim that there exists a $w \in X \cap \Sigma^n$ such that $1^n \in L(M_i^{X_w})$. For such a w, define $A_{i+1} = X_w$. Then requirement (R2) is fulfilled.

To prove our claim, assume that, for all $w \in X \cap \Sigma^n$, it holds that $1^n \notin L(M_i^{X_w})$. By taking w out of X, at least $acc_{M_i^X}(1^n) - acc_{M_i^{X_w}}(1^n)$ accepting paths of M either change to rejecting paths or disappear, and hence w must have been queried on those paths. Since

$$\begin{aligned} acc_{M_i^X}(1^n) - acc_{M_i^{X_w}}(1^n) &\geq \frac{3}{4}\, total_{M_i^X}(1^n) - \frac{1}{4}\, total_{M_i^{X_w}}(1^n) \\ &\geq \frac{1}{2}\, total_{M_i^X}(1^n) \quad \text{by } (\star), \end{aligned}$$

each $w \in X \cap \Sigma^n$ is queried by M_i^X on input 1^n on at least half of all paths. Thus M_i^X asks at least $2^{n-1}\,\frac{1}{2}\, total_{M_i^X}(1^n) = 2^{n-2}\, total_{M_i^X}(1^n)$ queries to its oracle. On the other hand, M_i^X cannot ask more than $p_i(n)\, total_{M_i^X}(1^n)$ queries to its oracle. Since $p_i(n) < 2^{n-2}$, this yields a contradiction.

The case $X \in \mathcal{B}$ is symmetric. Here one has to define X_w by *adding* a string $w \in \Sigma^n - X$ to X, and then, in case $1^n \in L(M_i^{X_w})$ for all $w \in \Sigma^n - X$, argue regarding the number of rejecting instead of accepting paths of M_i. □

4. If secure and oblivious computation differ, then P $\neq$ PSPACE. We show, for both threshold and probabilistic computation, that secure computation is more powerful than oblivious computation only if BPP $\neq$ PP (which would resolve in the affirmative the important question of whether polynomial time differs from polynomial space).

THEOREM 4.1. *If there is a database D such that* $\text{secureBPP}^D_{\text{path}} \neq \text{obliviousBPP}^D_{\text{path}}$, *then* BPP $\neq$ PP.

Proof. Assume BPP = PP. Note that this implies that BPP = $\text{P}^{\#\text{P}}$ (since $\text{P}^{\text{PP}} = \text{P}^{\#\text{P}}$ [4] and BPP = P^{BPP}). Let D be a database and let L be a language such that $L \in \text{secureBPP}^D_{\text{path}}$. We will show that $L \in \text{obliviousBPP}^D_{\text{path}}$, thereby proving the theorem.

Let N be the machine of Definition 2.6 certifying that $L \in \text{secureBPP}^D_{\text{path}}$. We may assume without loss of generality (since it is easy to see that $\text{secureBPP}^D_{\text{path}}$ machines can be amplified in the standard way and still remain secure) that the ϵ of Definition 2.6 satisfies $\epsilon > \frac{1}{4}$. Also, let $p(n)$ be a polynomial, of the form $n^i + i$ for some integer $i \geq 1$, such that for all sets L the runtime of N^L is at most $p(n)$.

Very informally summarized, in the following, a secure computation of N is decomposed (query vector by query vector) to allow an oblivious BPP_{path} machine to mimic N's computation. This will be possible because our assumption gives #P-like computational power to our oblivious BPP_{path} machine.

We will now define an oblivious machine Q such that Q^D certifies that $L \in \text{obliviousBPP}^D_{\text{path}}$. Let x, $|x| = n$, be the input for N^D. The computation of Q^D has essentially two stages. In the first stage, as long as the oblivious machine Q^D asks oracle queries, it only has 0^n available as input. What it does is the following: Q^D simulates N^D on input 0^n. At the end of each path, Q^D has defined a query vector, say $\boldsymbol{v}$. By the definition of secure computation, the proportion of occurrences of $\boldsymbol{v}$ is the same in $N^D(0^n)$ and $N^D(x)$, that is,

$$\frac{\textit{path-occurences}_{N^D(0^n)}(\boldsymbol{v})}{\textit{total}_{N^D}(0^n)} = \frac{\textit{path-occurences}_{N^D(x)}(\boldsymbol{v})}{\textit{total}_{N^D}(x)}. \tag{1}$$

In the second stage, Q gets access to its input x (and thus cannot ask anymore oracle queries). Let $\alpha_{N^D(x)}(\boldsymbol{v})$ denote the number of accepting paths of $N^D(x)$ that have query vector $\boldsymbol{v}$. Roughly speaking, at each path with query vector $\boldsymbol{v}$ found in the first stage, Q will append a full binary tree having approximately a portion of $\alpha_{N^D(x)}(\boldsymbol{v})/\textit{path-occurences}_{N^D(x)}(\boldsymbol{v})$ accepting paths. Therefore, Q^D will have approximately the same overall acceptance behavior as N^D.

More formally, we partition the unit interval into 2^q intervals of equal length, for some appropriately chosen q, and take the largest $k/2^q$, $k \in \{0, \ldots, 2^q - 1\}$, that is still less than $\alpha_{N^D(x)}(\boldsymbol{v})/\textit{path-occurences}_{N^D(x)}(\boldsymbol{v})$ as an approximation for it. This is done as follows. For a query vector $\boldsymbol{v}$ let $V = \{ v \mid v \in D \text{ and } v \text{ is a component of } \boldsymbol{v} \}$. Now, Q guesses k of length q and tests whether $(x, \boldsymbol{v}, V, k) \in A$, where A is defined as follows. For $y \in \Sigma^*$, a vector $\boldsymbol{w}$ of at most $p(|y|)$ strings each of length at most $p(|y|)$, a set of strings W each occurring as a component of vector $\boldsymbol{w}$, and a string j of length q, interpreted as a binary number between 0 and $2^q - 1$,

$$(y, \boldsymbol{w}, W, j) \in A \iff j \leq 2^q \frac{\alpha_{N^W(y)}(\boldsymbol{w})}{\textit{path-occurences}_{N^W(y)}(\boldsymbol{w})} - 1.$$

Clearly, $A \in \text{P}^{\#\text{P}}$, and thus A is in BPP by assumption. Hence there exist a probabilistic machine M_A and a polynomial h such that M_A accepts A with error probability

bounded by 2^{-q}, and furthermore, for any input $(y, \boldsymbol{w}, W, j)$, the computation tree of M_A is a full binary tree with $2^{h(|y|)}$ paths.

In order to test whether $(x, \boldsymbol{v}, V, k)$ is in A, Q simulates M_A on input $(x, \boldsymbol{v}, V, k)$. Q accepts x if and only if the simulation ends in an accepting state of M_A. This completes the definition of Q.

We will argue that the machine Q has the desired properties. By the definition of Q, it is clearly an oblivious machine. Furthermore, for any given input x, let $\boldsymbol{v}$ be a query vector that actually occurs in the run of $N^D(x)$. From equation (1), we get that the portion of paths in the tree of Q^D that have query vector $\boldsymbol{v}$ is identical to the portion in the tree of $N^D(x)$ that have query vector $\boldsymbol{v}$. We now argue that those paths in $Q^D(x)$ having query vector $\boldsymbol{v}$ have almost the same portion accepting as do those paths in $N^D(x)$. Since $\boldsymbol{v}$ was an arbitrary occurring query vector, it will follow that $Q^D(x)$ has appropriate behavior.

By our construction, we can bound $\alpha_{Q^D(x)}(\boldsymbol{v})$, the number of accepting paths of Q^D that have query vector $\boldsymbol{v}$ as follows. Let V be the associated answer set for $\boldsymbol{v}$. Note that $\alpha_{N^V(x)}(\boldsymbol{v}) = \alpha_{N^D(x)}(\boldsymbol{v})$ and $\mathit{path\text{-}occurences}_{N^V(x)}(\boldsymbol{v}) = \mathit{path\text{-}occurences}_{N^D(x)}(\boldsymbol{v})$. Hence we have $(x, \boldsymbol{v}, V, k) \in A$ if and only if $0 \leq k \leq \lfloor 2^q \left(\alpha_{N^D(x)}(\boldsymbol{v}) / \mathit{path\text{-}occurences}_{N^D(x)}(\boldsymbol{v})\right) \rfloor - 1$. Since M_A has error probability at most 2^{-q}, we get the following lower bound for $\alpha_{Q^D(x)}(\boldsymbol{v})$:

$$\mathit{path\text{-}occurences}_{Q^D(x)}(\boldsymbol{v}) \frac{\left\lfloor 2^q \frac{\alpha_{N^D(x)}(\boldsymbol{v})}{\mathit{path\text{-}occurences}_{N^D(x)}(\boldsymbol{v})} \right\rfloor}{2^q} (1 - 2^{-q}) \leq \alpha_{Q^D(x)}(\boldsymbol{v}).$$

For an upper bound, we have to count the small number of extra accepting paths caused by the error probability of M_A:

$$\alpha_{Q^D(x)}(\boldsymbol{v}) \leq \mathit{path\text{-}occurences}_{Q^D(x)}(\boldsymbol{v}) \frac{2^q \frac{\alpha_{N^D(x)}(\boldsymbol{v})}{\mathit{path\text{-}occurences}_{N^D(x)}(\boldsymbol{v})} + 1}{2^q}.$$

With these bounds on $\alpha_{Q^D(x)}(\boldsymbol{v})$, it is now easy to bound the error of Q^D for query vector $\boldsymbol{v}$. Namely, let

$$\mathit{error}(\boldsymbol{v}) = \left| \frac{\alpha_{Q^D(x)}(\boldsymbol{v})}{\mathit{path\text{-}occurences}_{Q^D(x)}(\boldsymbol{v})} - \frac{\alpha_{N^D(x)}(\boldsymbol{v})}{\mathit{path\text{-}occurences}_{N^D(x)}(\boldsymbol{v})} \right|;$$

then we get from the above bounds on $\alpha_{Q^D(x)}(\boldsymbol{v})$ that $\mathit{error}(\boldsymbol{v}) \leq 2^{-q+1}$. Since this holds for each occurring query vector $\boldsymbol{v}$, it certainly holds that 2^{-q+1} bounds the overall error portion: the difference between the portion of accepting paths of $N^D(x)$ and the portion of accepting paths of $Q^D(x)$ is at most 2^{-q+1}. Now define $q = 4$. Since N^D had an ϵ (of Definition 2.6) of at least $\frac{1}{4}$, and since we have $\frac{1}{4} - \frac{1}{8} = \frac{1}{8}$, we may conclude that Q^D is an oblivious machine accepting the same language as N^D and having ϵ (of Definition 2.8) equal to $\frac{1}{8}$. □

The proof of Theorem 4.1 can easily be modified to show the corresponding result for probabilistic classes.

COROLLARY 4.2. *If there is a database D such that* secureBPP$^D \neq$ obliviousBPP^D, *then* BPP $\neq$ PP.

Recall that sets in obliviousBPP^D have small circuits. Thus the existence of a set in secureBPP^D not having a small circuit would separate obliviousBPP^D from secureBPP^D.

COROLLARY 4.3. *If there is a database D such that* $\text{secureBPP}^D \not\subseteq \text{P/poly}$*, then* $\text{BPP} \neq \text{PP}$.

Since $\text{P} \subseteq \text{BPP} \subseteq \text{PP} \subseteq \text{PSPACE}$, we immediately have the result promised in the section title.

COROLLARY 4.4. *If there is a database D such that* $\text{secureBPP}^D_{\text{path}} \neq \text{obliviousBPP}^D_{\text{path}}$*, then* $\text{P} \neq \text{PSPACE}$.

5. Open problems. There are several open problems regarding BPP_{path}. Is BPP_{path} contained in Σ^p_2 or even in R^{NP}? It seems that the proof technique of Theorem 3.11 does not suffice to establish either of these relationships. Does BPP_{path} have complete sets? There is a relativized world in which BPP lacks complete sets [14]; we conjecture that the same holds for BPP_{path}.

Regarding secure computation, does there exist a structural condition that completely characterizes the conditions under which $(\forall D)\,[\text{secureBPP}^D = \text{obliviousBPP}^D]$ or that completely characterizes the conditions under which $(\forall D)\,[\text{secureBPP}^D_{\text{path}} = \text{obliviousBPP}^D_{\text{path}}]$? The study, mentioned in section 2.2, of classes between BPP^D and secureBPP^D and of classes between $\text{BPP}^D_{\text{path}}$ and $\text{secureBPP}^D_{\text{path}}$ also remains an interesting open area.

Appendix. Randomized databases do not strengthen secure probabilistic computation. The secure probabilistic computation of Definition 2.7 can be considered a special case of two-player interactive computation. In particular, the database can be considered a powerful player that truthfully answers difficult questions asked by a polynomial-time player. When the powerful player in a secure probabilistic computation answers a query, it is unable to take the past history of transactions into consideration. In contrast, players in the usual interactive computation models can remember the history of past transactions. Nonetheless, the secure probabilistic computation model is quite powerful. Even if the database is replaced with a deterministic player that has unlimited computation power and memory, it is clear that the resulting interactive computation can be simulated by a polynomial-time player with a new database that is merely a set.

In this section, we consider the effect of allowing the powerful player to be probabilistic. The resulting model is called a one-oracle instance-hiding scheme that leaks at most the length of its input [5]. We present a slightly modified but equivalent definition.

DEFINITION A.1 (one-oracle instance-hiding scheme that leaks at most the length of its input). *For a set L, a* one-oracle instance-hiding scheme that leaks at most the length of its input *is a synchronous protocol executed by two players, M_A and M_B. The number of rounds is bounded by a polynomial in the length of the input. In each round, M_A does a randomized polynomial-time local computation and sends a message (i.e., query) to M_B. Upon receiving the query from M_A, M_B does an unbounded amount of local computation (possibly using an oracle and a random tape) and sends a message (i.e., answer) to M_A. The round is completed when M_A receives the answer sent by M_B. Let τ denote the sequence of messages sent and received by M_A along a computation path, and let T_A denote the random tape of M_A. After the last round, M_A uses τ, T_A, and the input x to compute a value $M_A(x)$. The interactive computation scheme should satisfy the following two conditions:*

1. [*Probability of acceptance is bounded away from $\frac{1}{2}$*]. *There exists an $\epsilon > 0$ such that for all $x \in \Sigma^*$ it holds that $\Pr[M_A(x) = L(x)] > \frac{1}{2} + \epsilon$. (Note that the probability depends on the combined effect of the randomness of both M_A and M_B.)*

2. [*The messages reveal no information to an observer other than perhaps the length of the input.*] *For every* $k \in \{0, 1, 2, \ldots\}$, *every vector* $\boldsymbol{v} = (q_1, a_1, q_2, a_2, \ldots, q_k, a_k)$, $q_1, a_1, q_2, a_2, \ldots, q_k, a_k \in \Sigma^*$, *and every pair of strings* $x \in \Sigma^*$ *and* $y \in \Sigma^*$ *such that* $|x| = |y|$, *it holds that*

$$\Pr[\tau = \boldsymbol{v} \text{ on input } x] = \Pr[\tau = \boldsymbol{v} \text{ on input } y].$$

For any polynomial $p(\cdot)$, the above probability $\frac{1}{2} + \epsilon$ can be amplified to $1 - 2^{-p(|x|)}$ via the standard technique of repeating computations and using the most frequent result. Clearly, if $L \in \text{secureBPP}^D$ for some database D, then L has a one-oracle instance-hiding scheme that leaks at most the length of its input. The following theorem, pointed out to us by an anonymous conference referee, shows that the converse is also true.

THEOREM A.2. *If* L *is a language that has a one-oracle instance-hiding scheme that leaks at most the length of its input, then there exists a database* D *such that* $L \in \text{secureBPP}^D$.

Proof. Let L be a language that has a one-oracle instance-hiding scheme that leaks at most the length of its input. In this proof, we use the notation of Definition A.1. Following [1], we use the term *transcript* to denote τ, the sequence of queries and answers along a computation path. Without loss of generality, we assume that no transcript is a proper prefix of another transcript and that the length of an input is passed to M_B as the first query. In this proof, we first show that M_B can be modified so that it needs only a polynomial number of random bits. Then we show that these random bits can be supplied by M_A, thereby eliminating the need for M_B to be random. It follows that the resulting powerful but deterministic player can be replaced with a set as claimed in the theorem. In the rest of the proof, we call the machines M_A and M_B the *client* and the *server*, respectively.

Given an input of length n, the set of transcripts that have nonzero probabilities define a tree whose depth is bounded by a polynomial in n. Let's call this a *strategy tree*. (As will become clear later in this proof, the strategy tree effectively defines the strategy of the server. Also, it serves as a convenient template for modifying the strategy of the server.) There are two types of nodes in a strategy tree: server nodes and client nodes. These two types of nodes alternate in each path from the root to a leaf. The root is a client node. The leaves are also client nodes. Each edge from a client node is labeled with a query string; each edge from a server node is labeled with an answer string. Each leaf represents a transcript that has a nonzero probability; the transcript consists of labels read from the edges along the path from the root to the leaf. Edges from the same node have distinct labels so that a transcript defines a unique path in a strategy tree. Corresponding to each internal node in a strategy tree, there exists a *partial transcript* that consists of the labels that are read from the edges along the path from the root to the node.

Associated with each leaf is the probability with which the transcript corresponding to the leaf occurs. Clearly, based on this probability distribution, we can associate with each internal node the probability with which the partial transcript corresponding to the node occurs. To each edge from a node, we associate the conditional probability with which its label occurs as the next query or answer in a computation, given that the current partial transcript of the computation is the one represented by the node. Note that the sum of the probabilities associated with all the edges from a node is one and that the probability associated with each node is the product of the probabilities associated with the edges along the path from the root to the node.

It is easy to see that an interactive computation reveals at most the length of the input (in the sense of part 2 in Definition A.1) if and only if its strategy tree is the same for all inputs of the same length. In particular, the strategy of the server (that is, the probability distribution among edges from each server node) is the same for all inputs of the same length. Further, if we modify the server but (i) we do not add new transcripts to the strategy tree and (ii) the client is not changed, then the resulting strategy tree is the same for all inputs of the same length. Hence we may arbitrarily adjust the probability distribution among the existing edges from each server node without affecting the instance-hiding nature of the computation. However, such change could affect the acceptance probabilities of input strings. Therefore, in the following, we carefully modify the behavior of the server so that the acceptance of each input string remains intact. In particular, assuming without loss of generality that the probability of correctness (in the sense of part 1 in Definition A.1) of the original instance-hiding computation is greater than $\frac{3}{4}$, we will ensure that the probability of correctness of the modified instance-hiding computation is greater than $\frac{5}{8}$.

Let $q(n)$ be a polynomial that bounds both the length of the label of each edge and the depth of the strategy tree. The main obstacle in transforming the randomized server to a deterministic one is the fact that the probability of an edge from a server node can be an arbitrary value. In order to get around the obstacle, we adjust the probability of each edge from server nodes so that it is an integral multiple of $2^{-q^2(n)-q(n)-3}$ and that it differs from the original probability by less than $2^{-q^2(n)-q(n)-3}$. Thus the probability change at each leaf of the strategy tree is less than $q(n)2^{-q^2(n)-q(n)-3}$. Since there are at most $2^{q^2(n)}$ leaves, it is easy to see that the change in the probability of correctness of the whole computation is less than $\frac{1}{8}$. Therefore, the probability of correctness of the modified secure computation is greater than $\frac{5}{8}$. Note that the resulting strategy tree can be constructed by the server upon receiving the first query (i.e., the length of the input). The server uses this strategy tree to answer all the queries.

The server modified in this way needs at most a polynomial number $(q(n)(q^2(n)+q(n)+3))$ of random bits. Hence the necessary random bits can be supplied to the server by the client at the beginning of a computation. Note that this modification affects neither the instance-hiding nature of the computation nor the probability of correctness of the computation. The resulting server is deterministic, but it may not yet be considered a deterministic function oracle since it may give different answers to different instances of the same queried string. By prefixing each query with an ppropriate public information with which the server can uniquely locate the current stage of computation in the strategy tree (for example, $\langle q_1, \ldots, q_{i-1} \rangle$ can be used as a prefix to the ith query along a computation path on which q_j $(0 < j < i)$ is the jth query), the server can be transformed into a deterministic function oracle. It is easy to see that we can further modify the client so that it securely accepts the same language with a set oracle (D) instead of a function oracle. Clearly, the resulting computation is a secure probabilistic computation. □

Acknowledgments. For helpful discussions, we are grateful to F. Ablayev, G. Brassard, J. Cai, L. Fortnow, F. Green, J. Seiferas, and S. Toda. We thank an anonymous conference referee for pointing out Theorem A.2 and for helpful pointers to the literature.

REFERENCES

[1] M. Abadi, J. Feigenbaum, and J. Kilian, *On hiding information from an oracle*, J. Comput. System Sci., 39 (1989), pp. 21–50.

[2] L. Babai, *Trading group theory for randomness*, in Proc. 17th ACM Symposium on Theory of Computing, ACM, New York, 1985, pp. 421–429.

[3] T. Baker, J. Gill, and R. Solovay, *Relativizations of the P=?NP question*, SIAM J. Comput., 4 (1975), pp. 431–442.

[4] J. Balcázar, R. Book, and U. Schöning, *The polynomial-time hierarchy and sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 603–617.

[5] D. Beaver and J. Feigenbaum, *Hiding instances in multioracle queries*, in Proc. 7th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 415, Springer-Verlag, Berlin, 1990, pp. 37–48.

[6] R. Beigel, *Perceptrons, PP, and the polynomial hierarchy*, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 14–19.

[7] R. Book, *Restricted relativizations of complexity classes*, in Computational Complexity Theory, J. Hartmanis, ed., Proceedings of Symposia in Applied Mathematics 38, AMS, Providence, RI, 1989, pp. 47–74.

[8] R. Boppana, J. Håstad, and S. Zachos, *Does co-NP have short interactive proofs*?, Inform. Process. Lett., 25 (1987), pp. 127–132.

[9] J. Feigenbaum, L. Fortnow, C. Lund, and D. Spielman, *The power of adaptiveness and additional queries in random-self-reductions*, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 338–346; final version appears as Comput. Complexity, 4 (1994), pp. 158–174.

[10] S. Fenner, L. Fortnow, and S. Kurtz, *Gap-definable counting classes*, J. Comput. System Sci., 48 (1994), pp. 116–148.

[11] S. Fenner, L. Fortnow, S. Kurtz, and L. Li, *An oracle builder's toolkit*, in Proc. 8th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 120–131.

[12] L. Fortnow, personal communication, 1994.

[13] J. Gill, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675–695.

[14] J. Hartmanis and L. Hemachandra, *Complexity classes without machines: On complete languages for UP*, Theoret. Comput. Sci., 58 (1988), pp. 129–142.

[15] L. Hemachandra, *The strong exponential hierarchy collapses*, J. Comput. System Sci., 39 (1989), pp. 299–322.

[16] U. Hertrampf, C. Lautemann, T. Schwentick, H. Vollmer, and K. Wagner, *On the power of polynomial time bit-reductions (extended abstract)*, in Proc. 8th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 200–207.

[17] B. Jenner, P. McKenzie, and D. Thérien, *Logspace and logtime leaf languages*, in Proc. 9th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 242–253.

[18] M. Jerrum, L. Valiant, and V. Vazirani, *Random generation of combinatorial structures from a uniform distribution*, Theoret. Comput. Sci., 43 (1986), pp. 169–188.

[19] R. Karp and R. Lipton, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th ACM Symposium on Theory of Computing, ACM, New York, 1980, pp. 302–309; extended version has also appeared as *Turing machines that take advice*, Enseign. Math. (2nd series), 28 (1982), pp. 191–209.

[20] K. Ko, *Some observations on the probabilistic algorithms and NP-hard problems*, Inform. Process. Lett., 14 (1982), pp. 39–43.

[21] J. Köbler, U. Schöning, S. Toda, and J. Torán, *Turing machines with few accepting computations and low sets for PP*, J. Comput. System Sci., 44 (1992), pp. 272–286.

[22] R. Ladner, N. Lynch, and A. Selman, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–124.

[23] C. Lautemann, *BPP and the polynomial hierarchy*, Inform. Process. Lett., 14 (1983), pp. 215–217.

[24] T. Long and A. Selman, *Relativizing complexity classes with sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 618–627.

[25] A. Meyer and L. Stockmeyer, *The equivalence problem for regular expressions with squaring requires exponential space*, in Proc. 13th IEEE Symposium on Switching and Automata Theory, IEEE, Piscataway, NJ, 1972, pp. 125–129.

[26] M. Ogiwara and L. Hemachandra, *A complexity theory for closure properties*, J. Comput.

System Sci., 46 (1993), pp. 295–325.
[27] J. Simon, *On some central problems in computational complexity*, Ph.D. thesis, 1975; available as Technical Report TR75-224, Department of Computer Science, Cornell University, Ithaca, NY, Jan. 1975.
[28] M. Sipser, *A complexity theoretic approach to randomness*, in Proc. 15th ACM Symposium on Theory of Computing, ACM, New York, 1983, pp. 330–335.
[29] L. Stockmeyer, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.
[30] L. Stockmeyer, *On approximation algorithms for #P*, SIAM J. Comput., 14 (1985), pp. 849–861.
[31] S. Toda, *PP is as hard as the polynomial-time hierarchy*, SIAM J. Comput., 20 (1991), pp. 865–877.
[32] S. Toda and M. Ogiwara, *Counting classes are at least as hard as the polynomial-time hierarchy*, SIAM J. Comput., 21 (1992), pp. 316–328.
[33] N. Vereshchagin, *On the power of PP*, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 138–143.
[34] S. Zachos, *Robustness of probabilistic complexity classes under definitional perturbations*, Inform. Comput., 54 (1982), pp. 143–154.
[35] S. Zachos, *Probabilistic quantifiers and games*, J. Comput. System Sci., 36 (1988), pp. 433–451.
[36] S. Zachos and H. Heller, *A decisive characterization of BPP*, Inform. Control, 69 (1986), pp. 125–135.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 79–92, February 1997

005

DISJOINT ROOTED SPANNING TREES WITH SMALL DEPTHS IN DEBRUIJN AND KAUTZ GRAPHS*

ZHENGYU GE† AND S. LOUIS HAKIMI†

Abstract. The problem of broadcasting long messages on store-and-forward communication networks, where a processor (node) can send and receive messages simultaneously to and from all its neighbors, was studied by Bermond and Fraigniaud. In such networks, the delays encountered by a message from a node v to all other nodes over a broadcast spanning tree is directly proportional to the length of the paths in the tree over which the message is sent. Furthermore, the speed of the broadcast can be improved by the segmentation of the message at v into equal-length segments and then the broadcast of these segments over arc-disjoint broadcast spanning trees simultaneously. These observations lead Bermond and Fraigniaud to look for the maximum number of arc-disjoint spanning trees in a deBruijn network rooted at an arbitrary node with small depths. This paper improves and extends the results of the above authors.

Key words. broadcasting, communication networks, interconnection architectures, deBruijn networks, Kautz networks, arc-disjoint spanning trees, fault-tolerant networks

AMS subject classifications. 05C05, 68M10, 68M15, 94

PII. S0097539793244198

1. Introduction.

The deBruijn and Kautz networks represent a useful class of interconnection architecture for multiprocessor systems [1, 2, 3, 4]. The normal definitions of the deBruijn and Kautz networks yield digraphs; interesting and challenging problems also arise when the directions of the arcs of these digraphs are ignored, thus converting all arcs into edges and digraphs into graphs. We will denote the normal deBruijn and Kautz digraphs by $B(\Delta, D)$ and $K(\Delta, D)$, where Δ is the out-degree (and in-degree) of each node and D is the diameter of these digraphs. We will also denote the associated undirected graphs by $UB(\Delta, D)$ and $UK(\Delta, D)$.

The deBruijn and Kautz digraphs have many desirable properties such as large number of nodes (Δ^D for deBruijn and $\Delta^D + \Delta^{D-1}$ for Kautz digraphs), small diameter D, nearly optimal connectivities ($\Delta - 1$ for $B(\Delta, D)$ and Δ for $K(\Delta, D)$), and very simple routing procedures [1, 2, 6].

The arcs in digraphs represent directional communication links while edges in graphs represent communication links that permit communication in either direction but in one direction at a time. We will consider the store-and-forward model for communication between nodes [5]. We will also assume that a node can simultaneously send (receive) messages to (from) all its neighbors.

Broadcasting, that is, sending a message from a given node v to all other nodes in the network, is an important network function that is often encountered in distributed computing or paralleled algorithms. To effectively broadcast a message from a node v to all other nodes, one must accomplish this task at high transmission rates (throughput) and with small delays. In a store-and-forward network, broadcasting a message from node v occurs over the arcs of a (broadcast) spanning tree rooted at v. The maximum delay that the message at v encounters is directly proportional to the depth of this broadcast tree. Thus if we wish to broadcast a message at v to all

* Received by the editors February 8, 1993; accepted for publication (in revised form) April 13, 1995. This research was supported by National Science Foundation grant NCR-91-02534.
http://www.siam.org/journals/sicomp/26-1/24419.html
† Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA 95616 (slhakimi@ucdavis.edu).

other nodes with the least delay, we select a shortest-path spanning tree rooted at v in the network as our broadcast tree. In fact, this choice would guarantee that the message from v is received at every node with the least amount of time delay. Suppose that we are also interested in the rate of transmission of the message at v. Barring improvements in hardware, the way to achieve the improvement is through parallel broadcasting. Parallel broadcasting involves the segmentation of the message at v into d equal-length segments and then the broadcast of each of these d segments over one of the d arc-disjoint broadcast trees in the network rooted at v. Improvement in delay can also be achieved by pipelining the packets in the segmentation of the message in each of these trees [1, 5, 9]. Since pipelining can be carried out independently of the parallel-broadcasting technique discussed here, it will not be further elaborated.

The above observation lead Bermond and Fraigniaud [1] to look for the maximum number of arc-disjoint broadcast trees rooted at node v with small depths in deBruijn digraphs. More precisely, they showed that in $B(\Delta, D)$, there are $\Delta - 1$ arc-disjoint broadcast trees rooted at any node v, with the depth of each tree not exceeding $D + 2\lfloor \frac{D}{2} \rfloor + 1 (\geq 2D)$. This represents the speedup of the rate of transmission by a factor of $\Delta - 1$ and an increase in the maximum delay by $2\lfloor \frac{D}{2} \rfloor + 1$ over the shortest-path broadcast tree.

In this paper, we say that two spanning trees are *arc disjoint* if no arc appears in both trees, and we say that two trees are *pseudo node disjoint* if no node can be internal nodes in both trees, where an *internal* node in a tree is a node which is neither the root nor a leaf node of that tree. Thus in a group of rooted pseudo-node-disjoint spanning trees, if a node is an internal node in one tree, it will be a leaf node in all other trees. It is easy to see that if two spanning trees rooted at node v are pseudo node disjoint, then they are also arc disjoint, with the possible exception of arcs emanating from root v. However pseudo-node-disjoint trees generated in this paper never share any arcs emanating from the root. Thus pseudo-node-disjoint trees generated here are also arc disjoint.

In this paper, we present methods for generating pseudo-node-disjoint spanning trees (PNDSTs) rooted at an arbitrary node in $B(\Delta, D)$ and arc-disjoint spanning trees (ADSTs) rooted at an arbitrary node in $K(\Delta, D)$. More precisely, we will demonstrate that there are $\Delta - 1$ PNDSTs of depth no greater than $\lceil \frac{3D}{2} \rceil$ in $B(\Delta, D)$ and there are Δ ADSTs of depth no greater than $\lceil \frac{3D}{2} \rceil + 1$ in $K(\Delta, D)$.

One of the shortcomings of the above results is that they are obtained for digraphs which are neither the normal model of the computer networks nor the interconnection architecture of multiprocessor systems. We will present a proof for the existence of Δ PNDSTs in $UB(\Delta, D)$, with the bound on the depths of these trees being at most $2D$. Although the removal of the directions on the arcs of $K(\Delta, D)$ may actually reduce the depths of the Δ spanning trees in $UK(\Delta, D)$, we have not been able to prove that the bound on the depths can always be reduced.

In the concluding section of this paper, the relations between the results of this paper and certain measures of reliability of networks are explained.

2. Disjoint spanning trees in deBruijn and Kautz digraphs: Preliminary considerations. $B(\Delta, D)$ is a digraph whose node set corresponds the set of all sequences of length D on the alphabet $\{1, 2, \ldots, \Delta\}$. For the Kautz digraph $K(\Delta, D)$, the node set corresponds to all sequences of length D over the alphabet $\{1, 2, \ldots, \Delta + 1\}$ provided no two consecutive letters in such a sequence can be the same. We will refer to the sequences with this feature as *Kautz sequences*. The set of arcs of $B(\Delta, D)$ corresponds the set of all sequences of length $D + 1$ on

the alphabet $\{1, 2, \ldots, \Delta\}$ and the set of arcs in $K(\Delta, D)$ corresponds to the set of all Kautz sequences of length $D+1$ on the alphabet $\{1, 2, \ldots, \Delta+1\}$. The arc $e = (a_1, a_2, \ldots, a_{D+1})$ in $B(\Delta, D)$ and in $K(\Delta, D)$ is from node $(a_1, a_2, \ldots, a_D)$ to node $(a_2, a_3, \ldots, a_{D+1})$. There are exactly Δ arcs into and out of each node in $B(\Delta, D)$ and $K(\Delta, D)$.

Let $R = (r_1, r_2, \ldots, r_D)$ be a node in $B(\Delta, D)$ or $K(\Delta, D)$. We intend to broadcast the message at R to all other nodes. We are, in fact, seeking $\Delta - 1$ PNDSTs (respectively, Δ ADSTs) rooted at R in $B(\Delta, D)$ (respectively, in $K(\Delta, D)$). We refer to these sets of trees as $TB(D, R)$ and $TK(D, R)$, respectively.

The *depth* of a tree is the length of a longest path in the tree starting at the root R. Furthermore, if a node u (respectively, an arc e) is the ith node (arc) in a path in the tree from R, then the node u is said to be at the $(i-1)$th level and the arc e is at the ith level, while the root R itself is considered at the 0th level.

We call $u = (a_1, a_2, \ldots, a_D) \in B(\Delta, D)$ a *corner* node if $a_1 = a_2 = \cdots = a_D$. There are exactly Δ corner nodes in $B(\Delta, D)$. If $R = (r_1, r_2, \ldots, r_D)$ is a corner node, then R has $\Delta - 1$ outgoing arcs, excluding the loop, in $B(\Delta, D)$ and thus each of the $\Delta - 1$ trees in $TB(D, R)$ rooted at R has exactly one outgoing arc incident at R. Otherwise, R will have Δ outgoing arcs; thus all but one of the $\Delta - 1$ trees in $TB(D, R)$ have exactly one outgoing arc incident at R and the exceptional tree in $TB(D, R)$ may have two arcs incident at R. We call the other end nodes of these outgoing arcs from R the *subroots*. For $K(\Delta, D)$, which does not have the corner nodes, all the Δ trees in $TK(D, R)$ have exactly one subroot.

Let $G(V, E)$ be a digraph with node set $V = V(G)$ and arc set $E = E(G)$. For $v \in V$, let $\Gamma v = \{u \in V \mid (v, u) \in E\}$ and $\Gamma^{-1} v = \{u \in V \mid (u, v) \in E\}$. We will call Γv the *successors* of v and $\Gamma^{-1} v$ the *predecessors* of v. If G is a rooted directed tree, the successors of v are called *child* nodes of v and the predecessor of v is called the *parent* node of v.

The line digraph of digraph G, denoted by $L(G)$, is a digraph with $V(L(G)) = E(G)$, that is, there is a node in $L(G)$ for each arc in $E(G)$; and, furthermore, there is an arc in $L(G)$ from e_1 to e_2 if the sequence of arcs $e_1 e_2$ forms a directed path in G.

LEMMA 2.1. (See [3, 6, 7].) *Let $B(\Delta, D)$ and $K(\Delta, D)$ be the deBruijn and Kautz digraphs as previously defined. Then for each $D \geq 1$, $B(\Delta, D+1) = L(B(\Delta, D))$ and $K(\Delta, D+1) = L(K(\Delta, D))$ and consequently $B(\Delta, D) = L^D(B(\Delta, 1))$ and $K(\Delta, D) = L^D(K(\Delta, 1))$, where $L^D(G) = L(L^{D-1}(G))$, $B(\Delta, 1)$ is a complete digraph on Δ nodes with a loop on each node, and $K(\Delta, 1)$ is a complete digraph on $\Delta + 1$ nodes without any loops.*

Let $a, b \in \{1, 2, \ldots, \Delta\}$ and $\Delta > 2$. (Since our problem is trivial when $\Delta = 2$, this case will not be considered.) An *$a - b$ node* is a node $u = (a_1, a_2, \ldots, a_D)$ such that $a_i \in \{a, b\}$ for $i = 1, 2, \ldots, D$. Note that a corner node is a special case of the $a - b$ nodes when $a = b$. Thus the following theorem is a generalization of a result in [1].

THEOREM 2.2. *Let $R = (r_1, r_2, \ldots, r_D)$ be an $a - b$ node in $B(\Delta, D)$. Then there is a set $TB(D, R)$ of $\Delta - 1$ PNDSTs rooted at R in $B(\Delta, D)$ with depths equal to $D + 1$.*

Proof. An algorithm is offered in this proof which provides the $\Delta - 1$ PNDSTs described in Theorem 2.2. Actually, we will prove a stronger result as follows. If u is an internal node in one tree of $TB(D, R)$, then it must be a node at the last level, $D + 1$, in all other trees in $TB(D, R)$. Thus the trees in $TB(D, R)$ are definitely pseudo node disjoint.

Consider the set of $\Delta - 2$ shortest-path spanning trees $T'(s_i)$ in $B(\Delta, D)$ rooted at the $\Delta - 2$ nodes $S_i = (r_2, \ldots, r_D, s_i)$ with $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$. We first prove that these trees $\{T'(s_i) \mid s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}\}$ are pseudo node disjoint.

Consider two such trees $T'(s_i)$ and $T'(s_j)$ with $s_i \neq s_j$. Note that the depths of these trees are D. For convenience, though an abuse of notation, we say that the root S_i is at level 1 in $T'(s_i)$ and thus the last level nodes in $T'(s_i)$ are at level $D + 1$. Let u and v be nodes in $T'(s_i)$ and $T'(s_j)$ at levels p and q from the roots S_i and S_j, respectively. Assume that both u and v are not nodes at the last levels, $D + 1$. Then if $1 \leq p \leq D - 1$ and $1 \leq q \leq D - 1$, we can write

$$\begin{array}{lll} u = (r_{p+1}, \ldots, r_D, \mathbf{s_i}, x_1, \ldots, x_{p-1}), & x_s \in \{1, 2, \ldots, \Delta\} & \text{for } s = 1, \ldots, p-1, \\ v = (r_{q+1}, \ldots, r_D, \mathbf{s_j}, y_1, \ldots, y_{q-1}), & y_t \in \{1, 2, \ldots, \Delta\} & \text{for } t = 1, \ldots, q-1, \end{array}$$

and if $p = D$ and $q = D$,

$$\begin{array}{lll} u = (\mathbf{s_i}, x_1, \ldots, x_{D-1}), & x_s \in \{1, 2, \ldots, \Delta\} & \text{for } s = 1, \ldots, D-1; \\ v = (\mathbf{s_j}, y_1, \ldots, y_{D-1}), & y_t \in \{1, 2, \ldots, \Delta\} & \text{for } t = 1, \ldots, D-1. \end{array}$$

Because R is an $a - b$ node, $r_k \in \{a, b\}$ $\forall k, s_i, s_j \in \{1, 2, \ldots, \Delta\} - \{a, b\}$, and $s_i \neq s_j$, we have $u \neq v$. Therefore, any node $u \in B(\Delta, D)$ could be an internal node in at most one of the trees in $\{T'(s_i) \mid s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}\}$. This implies that $T'(s_i)$ and $T'(s_j)$ are pseudo node disjoint.

Since $s_i \notin \{a, b\}$, all $a - b$ nodes, including node R, are in level $D + 1$ of $T'(s_i)$, and thus they are leaves. This implies that if we remove the arc into R from $T'(s_i)$ and add the arc $(R, S_i) \in B(\Delta, D)$ to it, we obtain a new tree denoted by $T(s_i)$. It is easy to see that $\{T(s_i) \mid s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}\}$ are directed spanning trees rooted at R and that they are pseudo node disjoint of depths $D + 1$.

Before proceeding to construct the last tree in $TB(D, R)$, we would like to clarify the relation between the levels of the nodes in the trees in $\{T(s_i) \mid s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}\}$ and the labels (i.e., sequences) associated with the nodes. For example, consider the nodes in $T(s_i)$ in the following order. The root R at level 0 followed by the subroot S_i at level 1, a node at level k, $1 < k < D$, a node at level D, and finally a node at level $D + 1$ will have the following labels:

$$\begin{aligned} R = (r_1, r_2, \ldots, r_D) \to S_i = (r_2, \ldots, r_D, \mathbf{s_i}) \to \cdots \to (r_{k+1}, \ldots, r_D, \mathbf{s_i}, x_1, \ldots, x_{k-1}) \\ \to \cdots \to (\mathbf{s_i}, x_1, \ldots, x_{D-1}) \to (x_1, \ldots, x_D). \end{aligned}$$

We will use this notation to designate a node at a particular level, say in tree $T(s_i)$.

We will now prove that there is one more tree rooted at R in $B(\Delta, D)$ of depth $D + 1$ which is pseudo node disjoint from the trees in $\{T(s_i) \mid s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}\}$. (The special case of the above claim, when $a = b$, has already been established in [1]). Toward this goal, we proceed as follows. Let $S_a = (r_2, \ldots, r_D, a)$ and $S_b = (r_2, \ldots, r_D, b)$, $a \neq b$, be the two subroots, and let $T'(ab)$ be a constrained shortest-path spanning tree rooted at R in $B(\Delta, D)$ in which every path starting at R passes through the subroot S_a or S_b. Note that the length of a path in $T'(ab)$ from R to any node whose first letter is in $\{a, b\}$ is at most D, and otherwise, this length is $D + 1$. It is easy to see that the depth of $T'(ab)$ will be $D + 1$.

Let $T'(a)$ and $T'(b)$ be the partial trees of $T'(ab)$ rooted at nodes S_a and S_b, respectively. Note that $V(T'(ab)) = V(T'(a) \cup T'(b) \cup R)$. Without loss of generality, let $u \in T'(a)$ be a node of $T'(ab)$ at level p, $1 \leq p \leq D$. If u is identical to a node $v \in T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$, at level q, $1 \leq q \leq D$, then we have

$$v = (r_{q+1}, r_{q+2}, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, r_D, \mathbf{s_i}, x_1, \ldots, x_{q-1}),$$

where $x_1, \ldots, x_{q-1}$ is an arbitrary sequence over $\{1, 2, \ldots, \Delta\}$. Since $u = v$ and $s_i \notin \{a, b\}$, we may write

$$u = (r_{p+1}, r_{p+2}, \ldots, r_D, \mathbf{a}, a_1, a_2, \ldots, a_{p-q-1}, \mathbf{s_i}, x_1, \ldots, x_{q-1}),$$

where $a_1, a_2, \ldots, a_{p-q-1}$ is an $a - b$ sequence. Since the length of the $a - b$ sequence, $a_1, a_2, \ldots, a_{p-q-1}$, is nonnegative, we have $p - q - 1 \geq 0$, or

$$q \leq p - 1.$$

Note that u and v may or may not be internal nodes in their respective trees.

Suppose an internal node u' at level h, $h \leq D$, of $T'(ab)$ is a node at level $D+1$ in all other trees $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$. Then we claim that all nodes on the path in $T'(ab)$ from R to u' excluding R are also nodes at level $D+1$ in all other trees $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$. To see this, suppose otherwise; that is, there is at least one node on the path from R to u' in $T'(ab)$ that is not at level $D+1$ in all other trees $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$. Among all such nodes, let w be the closest one to u'. Suppose w is at level p, $p \leq h - 1 \leq D - 1$, in $T'(ab)$, and assume w is at level q, $q \leq D$, in some other tree $T(s_i)$. As before, we must have $q \leq p - 1 \leq D - 2$. Since $T(s_i)$ was produced from the shortest-path spanning tree $T'(s_i)$ rooted at S_i, the successor nodes of w in $B(\Delta, D)$ are at a level at most $q + 1 \leq D - 1$ in $T(s_i)$. This implies the child node, say z, of w which is on the path from w to u' in $T'(ab)$ is also not at level $D + 1$ in all other trees. This is a contradiction since w was the closest one to u'.

The above claim states that if $T'(ab)$ contains some internal nodes, denoted by $I_n(T'(ab))$, that are not at level $D + 1$ in all other trees $T(s_i)$ $\forall s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$, then the child nodes of any node in $I_n(T'(ab))$ are either in $I_n(T'(ab))$ or are leaves of $T'(ab)$. This in turn implies that there is a node in $I_n(T'(ab))$ whose child nodes are all leaves. Without loss of generality, let $u \in T'(a)$ be such an internal node at level p, $p \leq D$, of $T'(ab)$ whose child nodes are all leaves. Suppose u is identical to $v \in T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$, at level q, $q \leq p - 1$. Let the node $u^* \in T'(ab)$ be defined as follows:

$$u^* = (\mathbf{r^*_{p+1}}, r_{p+2}, \ldots, r_D, \mathbf{a}, a_1, \ldots, a_{p-q-1}, \mathbf{s_i}, x_1, \ldots, x_{q-1})$$

or, equivalently,

$$u^* = (\mathbf{r^*_{q+1}}, r_{q+2}, \ldots, \ldots, \ldots, \ldots, r_D, \mathbf{s_i}, x_1, \ldots, x_{q-1}),$$

where $r^*_{p+1} = r^*_{q+1} \in \{a, b\}$, $r^*_{p+1} \neq r_{p+1}$. Note that u^* and u have the same successor nodes in $B(\Delta, D)$. Let $T''(ab)$ be obtained from $T'(ab)$ by transferring the child nodes from u in $T'(ab)$ to u^* in $T''(ab)$. We will establish that (i) the depth of $T''(ab)$ is still $D + 1$ and (ii) all nodes on the path from R to u^*, excluding R but including u^*, correspond to nodes at level $D + 1$ in all other trees $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$.

We will explain shortly that the validity of the two statements above enables one to use the above process of transferring child nodes to eliminate all internal nodes in $T'(ab)$ which could possibly be internal nodes in any other tree $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$.

(i) Since the first letter in u^* is in $\{a, b\}$, the distance from R to u^* in $T'(ab)$ is at most D. Since the child nodes of u in $T'(ab)$ are leaves, the depth of $T''(ab)$ is not more than $D + 1$.

(ii) Let w be a node on path from R to u^* in $T'(ab)$ at level, say k. Note that if $k \geq p-q+1$, then

$$w = (r_{k+1}, r_{k+2}, \ldots, r_p, \mathbf{r^*_{p+1}}, r_{p+2}, \ldots, r_D, \mathbf{a}, a_1, \ldots, a_{p-q-1}, \mathbf{s_i}, x_1, \ldots, x_{k+q-p-1})$$

or, equivalently,

$$(1) \quad w = (r_{k+q-p+1}, \ldots, r_q, \mathbf{r^*_{q+1}}, r_{q+2}, \ldots, \ldots, \ldots, \ldots, r_D, \mathbf{s_i}, x_1, \ldots, x_{k+q-p-1}).$$

On the other hand, if $k < p-q+1$, then w will be an $a-b$ node. However, if w is an $a-b$ node, we already know that w will be at level $D+1$ in all trees $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$. Therefore, we need to examine the case when w is not an $a-b$ node. To the contrary, suppose w is at level m, $m \leq D$, in some other tree, say, $T(s_h)$, $s_h \in \{1, 2, \ldots, \Delta\} - \{a, b\}$. Then we have

$$(2) \qquad w = (r_{m+1}, r_{m+2}, \ldots, \ldots, \ldots, \ldots, r_D, \mathbf{s_h}, y_1, y_2, \ldots, y_{m-1}),$$

where $y_1, y_2, \ldots, y_{m-1}$ is an arbitrary sequence over $\{1, 2, \ldots, \Delta\}$. Since the subsequence in equation (1) up to s_i and the subsequence in equation (2) up to s_h are $a-b$ sequences, we have $s_i = s_h$. This implies that w is in $T(s_i) = T(s_h)$ at level $m = k+q-p$, but this is impossible because then equation (2) becomes

$$(3) \quad w = (r_{k+q-p+1}, \ldots, r_q, \mathbf{r_{q+1}}, r_{q+2}, \ldots, \ldots, \ldots, \ldots, r_D, \mathbf{s_i}, x_1, \ldots, x_{k+q-p-1});$$

this is a contradiction to (1) as $r_{q+1} \neq r^*_{q+1}$.

At this stage, we can conclude that if $u \in T'(ab)$ whose child nodes are all leaves in $T'(ab)$ is not at level $D+1$ of all other trees $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$, we can always transfer the child nodes of u to u^*, consequently forcing u to be a leaf in $T''(ab)$, and the path length from R to the child nodes will not exceed $D+1$.

We do this examination and adjustment from all parent nodes of the leaf nodes in $T'(ab)$ backward to lower levels until we meet the internal nodes such as u' which are at level $D+1$ of all other trees $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta\} - \{a, b\}$. The resulting directed tree, denoted by $T(ab)$, will be pseudo node disjoint from the other $\Delta - 2$ trees. This completes the proof of Theorem 2.2. □

We will now present the counterpart of Theorem 2.2 for $K(\Delta, D)$. Let a and b belong to $\{1, 2, \ldots, \Delta, \Delta+1\}$, $a \neq b$, and $\Delta \geq 2$. Note that there are exactly two $a-b$ nodes in $K(\Delta, D)$ for the specified a and b, $(\ldots abab)$ and $(\ldots baba)$. Thus the total number of $a-b$ nodes in $K(\Delta, D)$ is $2\binom{\Delta+1}{2}$. For simplicity, we will denote the Kautz sequence $(\ldots abab)$ of length p ending with letter b by $\{\mathbf{ab}\}_\mathbf{p}$ and the Kautz sequence $(\ldots baba)$ of length q ending with letter a by $\{\mathbf{ba}\}_\mathbf{q}$.

THEOREM 2.3. *Let R be an $a-b$ node in $K(\Delta, D)$. Then 1. there are $\Delta - 1$ PNDSTs rooted at R in $K(\Delta, D)$ of depths $D+1$ and 2. there are Δ ADSTs rooted at R in $K(\Delta, D)$ of depths $D+2$.*

Proof. We begin with an algorithm for constructing the $\Delta - 1$ PNDSTs rooted at R in $K(\Delta, D)$ of depths $D+1$ for the proof of part 1 of Theorem 2.3.

1. Without loss of generality, let $R = (\ldots abab) = (\{ab\}_D)$. For $s_i \in \{1, 2, \ldots, \Delta+1\} - \{a, b\}$, let $S_i = (\ldots ababs_i) = (\{ab\}_{D-1}s_i)$, and let $T'(s_i)$ be the shortest-path spanning tree in $K(\Delta, D)$ rooted at S_i. The depth of $T'(s_i)$ is D, the root S_i is considered at level 1, and the last level is $D+1$. Note that there are $\Delta - 1$ such trees. Consider two of them, $T'(s_i)$ and $T'(s_j)$ with $s_i \neq s_j$. We will now show that $T'(s_i)$ and $T'(s_j)$ are pseudo node disjoint. Let u and v be possible internal nodes in $T'(s_i)$

and $T'(s_j)$ at levels p and q from the roots S_i and S_j respectively, $1 \le p \le D$ and $1 \le q \le D$.

$$u = \big(\{ab\}_{D-p} s_i \{1, 2, \ldots, \Delta+1\}_{p-1}^{-s_i}\big),$$
$$v = \big(\{ab\}_{D-q} s_j \{1, 2, \ldots, \Delta+1\}_{q-1}^{-s_j}\big),$$

where $\{1, 2, \ldots, \Delta+1\}_{p-1}^{-s_i}$ is an arbitrary Kautz sequence of length $p-1$ on $\{1, 2, \ldots, \Delta+1\}$ whose first letter is not s_i. It is now easy to see that because $s_i \neq s_j, s_i \notin \{a, b\}$, and $s_j \notin \{a, b\}$, we have $u \neq v$. This implies that $T'(s_i)$ and $T'(s_j)$ are pseudo node disjoint.

Note that all nodes at levels from 1 to D in $T'(s_i)$ contain the letter s_i in their labels; thus since R is an $a-b$ node, R is at level $D+1$ of $T'(s_i)$, $s_i \in \{1, 2, \ldots, \Delta+1\} - \{a, b\}$. Then the arc into R is removed from $T'(s_i)$ and the arc (R, S_i) of $K(\Delta, D)$ is added to $T'(s_i)$. The resulting tree is denoted by $T(s_i)$, $s_i \in \{1, 2, \ldots, \Delta+1\} - \{a, b\}$. The set $\{T(s_i) \mid s_i \in \{1, 2, \ldots, \Delta+1\} - \{a, b\}\}$ constitutes a set of $\Delta - 1$ PNDSTs rooted at R in $K(\Delta, D)$ of depths $D+1$.

Similar to the deBruijn case, the nodes, say in $T(s_i)$, at level 0, level 1, level $p, 1 < p < D$, level D, and finally at level $D+1$ will have the following labels:

$$R = (\{ab\}_D) \to S_i = (\{ab\}_{D-1}\mathbf{s_i}) \to \cdots \to (\{ab\}_{D-p}\mathbf{s_i}\{1, 2, \ldots, \Delta+1\}_{p-1}^{-s_i})$$
$$\to \cdots \to (\mathbf{s_i}\{1, 2, \ldots, \Delta+1\}_{D-1}^{-s_i}) \to (\{1, 2, \ldots, \Delta+1\}_D^{-s_i}).$$

2. We will build a new tree of depth $D+2$ denoted by $T(ab)$ which is arc disjoint from all trees in the above set. Let $T''(ab)$ be the shortest-path spanning tree rooted at the subroot $S_{ab} = (\ldots ababa) = (\{ba\}_D)$. We now wish to characterize the nodes in $T''(ab)$. We say that the node $S_{ab} = (\{ba\}_D)$ is at level 1 of $T''(ab)$, and it is easy to see that the nodes at level 2 of $T''(ab)$ are

$$(\{ba\}_{D-1} s_k), s_k \in \{1, 2, \ldots, \Delta+1\} - \{a\}.$$

Note that $R = (\{ab\}_D)$ is at level 2 with $s_k = b$. Generally, the nodes at level p of $T''(ab)$, $2 \le p \le D+1$, have the form

$$(\{ba\}_{D-p+1} s_k \{1, 2, \ldots, \Delta+1\}_{p-2}^{-s_k}), s_k \in \{1, 2, \ldots, \Delta+1\} - \{a\}, \quad 2 \le p \le D+1.$$

By removing the arc (Sab, R) from $T''(ab)$, discarding the partial tree rooted at R in $T''(ab)$, and adding the arc (R, S_{ab}) of $K(\Delta, D)$ to $T''(ab)$, we build a subtree in $K(\Delta, D)$, denoted by $T'(ab)$. $T'(ab)$ can be characterized as follows. Beginning with root R and then subroot $S_{ab} = (\ldots ababa) = (\{ba\}_D)$, the tree $T'(ab)$ proceeds as the maximum breadth-first subtree with depth up to $D+1$ which contains as many nodes of $K(\Delta, D)$ as possible. At this stage, $T'(ab)$ is not a spanning tree, and every path from R to any node u in it is the shortest path in $K(\Delta, D)$ which begins with R and then S_{ab}. In fact, the node set at level p, $2 \le p \le D+1$, in $T'(ab)$, denoted by $V_{\text{in}}(T'(ab))$, has the form

$$V_{\text{in}}(T'(ab))$$
$$= \{(\{ba\}_{D-p+1} s_i \{1, 2, \ldots, \Delta+1\}_{p-2}^{-s_i}) \mid s_i \in \{1, 2, \ldots, \Delta+1\} - \{a, b\}, 2 \le p \le D+1\}.$$

It can be easily seen that the nodes that are missing from $T'(ab)$ may be described by

$$V_{\text{mis}}(T'(ab))$$
$$= \{(\{ab\}_{D-p} s_i \{1, 2, \ldots, \Delta+1\}_{p-1}^{-s_i}) \mid s_i \in \{1, 2, \ldots, \Delta+1\} - \{a, b\}, 1 \le p \le D-1\},$$

which are the nodes of the partial tree in $T''(ab)$ rooted at R. Note that $V(K(\Delta, D)) = \{R \cup S_{ab} \cup V_{\text{in}}(T'(ab)) \cup V_{\text{mis}}(T'(ab))\}$.

We first claim that the subtree $T'(ab)$ is pseudo node disjoint from those trees in $\{T(s_i) \mid s_i \in \{1, 2, \ldots, \Delta + 1\} - \{a, b\}\}$. To see this, we observe that the set of the nodes at level q, $1 \le q \le D$, say in $T(s_i)$, may be described as follows:

$$V_{\text{in}}(T(s_i)) = \{(\{ab\}_{D-q} s_i \{1, 2, \ldots, \Delta + 1\}_{q-1}^{-s_i}) \mid s_i \in \{1, 2, \ldots, \Delta + 1\} - \{a, b\}, 1 \le q \le D\}.$$

Now let us consider the above set of nodes. If $1 \le q \le D - 1$, it is easy to see that these nodes belong to $V_{\text{mis}}(T'(ab))$. If $q = D$, these nodes all belong to $V_{\text{in}}(T'(ab))$ at level $p = D + 1$. This proves the claim.

To complete the proof, we will show that for each node $u \in V_{\text{mis}}(T'(ab))$, there exists a node $u' \in V_{\text{in}}(T'(ab))$ at level $D + 1$ of $T'(ab)$ such that the arc (u', u) is not in any tree in the set $\{T(s_i) \mid s_i \in \{1, 2, \ldots, D + 1\} - \{a, b\}\}$. Let

$$u = (\{ab\}_{D-p} s_i \{1, 2, \ldots, \Delta + 1\}_{p-1}^{-s_i}), \quad 1 \le p \le D - 1.$$

If $p = 1$, we select

$$u' = (s_i \{ab\}_{D-1});$$

if $2 \le p \le D - 1$, we select

$$u' = (s_i \{ab\}_{D-p} s_i \{1, 2, \ldots, \Delta + 1\}_{p-2}^{-s_i}).$$

We first observe that u' is at level $D + 1$ in the trees $T(s_j)$, $s_j \ne s_i$. Thus the arc (u', u) cannot exist in these trees $T(s_j)$ in this case. In $T(s_i)$, node u is at level p, $p \le D - 1$, and u' is at level D, which implies that the arc (u', u) cannot belong to $T(s_i)$ either. Note, however, that u' is an internal node in $T(s_i)$ and $T(ab)$. □

3. Disjoint spanning trees in deBruijn and Kautz networks: General case. We first consider the problem of constructing $\Delta - 1$ PNDSTs in $B(\Delta, D)$ rooted at an arbitrary node $R = (r_1, r_2, \ldots, r_D)$. This set of trees will be denoted by $TB(D, R)$. We call the reader's attention to two groups of nodes around R in $B(\Delta, D)$. The first group consists of the *subroots*, denoted by $\mathbf{S_b(R)}$, that are the successor nodes of R; more precisely,

$$S_b(R) = \Gamma(R) = \{u \mid u = (r_2, \ldots, r_D, s_i), s_i \in \{1, 2, \ldots, \Delta\}\}.$$

The second group is the other $\Delta - 1$ predecessors of the subroots, called *adopters*, denoted by $\mathbf{A_d(R)}$,

$$A_d(R) = \{u \mid u = (d, r_2, \ldots, r_D), d \in \{1, 2, \ldots, \Delta\} \text{ and } d \ne r_1\}.$$

Note that $R \cup A_d(R)$ are the whole set of the Δ predecessors of the subroots.

Let p be the minimum nonnegative integer such that the last $D - p$ letters in $R = (r_1, r_2, \ldots, r_D)$ are an $a - b$ subsequence for some letters a and b which belong to $\{1, 2, \ldots, \Delta\}$. Let $R(0) = (r_{p+1}, r_{p+2}, \ldots, r_D)$ represent this subsequence and note that $D - p \ge 2$. Our process for obtaining the $\Delta - 1$ PNDSTs in $TB(D, R)$ is based on a recursive algorithm involving $p + 1$ stages.

Let $D(i)$, $R(i)$, and $B(\Delta, D(i))$ denote the diameter, the root, and the deBruijn digraph in stage i, $0 \le i \le p$. Note that $D(i) = D - p + i$ and $R(i) =$

$(r_{p-i+1}, r_{p-i+2}, \ldots, r_D)$. Initially, in the 0th stage of our algorithm, let $D(0) = D - p$, $R(0) = (r_{p+1}, r_{p+2}, \ldots, r_D)$; we find the $\Delta - 1$ PNDSTs in $TB(D(0), R(0))$ in $B(\Delta, D(0))$. Since $R(0)$ is an $a - b$ sequence, Theorem 2.2 establishes that the $\Delta - 1$ PNDSTs in $TB(D(0), R(0))$ exist, and these trees have depths $D - p + 1$. In the following stages, $1 \leq i \leq p$, we recursively use the following algorithm, whose ith stage involves finding the $\Delta - 1$ PNDSTs, $TB(D(i), R(i)) = \{t^j(i), j = 1, 2, \ldots, \Delta - 1\}$, in digraph $B(\Delta, D(i))$, where $t^j(i)$ denotes a tree in $TB(D(i), R(i))$. Note that when $i = p$, we obtain the desired set $TB(D(p), R(p))$, where $D(p) = D$, $R(p) = R$.

Assuming that we have obtained $TB(D(i-1), R(i-1))$, which are PNDSTs for $B(\Delta, D(i-1))$, we will now present the ith stage of the algorithm, which consists of three steps. Also, we present a number of simple observations within the description of the algorithm; most of the proofs for the observations are elementary and are left out.

ALGORITHM. (For the ith stage, $1 \leq i \leq p$.)

Step 1. Note that the root $R(i) = (r_{p-i+1}, r_{p-i+2}, \ldots, r_D) \in B(\Delta, D(i))$ corresponds to the arc directed toward $R(i-1)$ in $B(\Delta, D(i-1))$. We add an auxiliary node $x(i-1) = (r_{p-i+1}, r_{p-i+2}, \ldots, r_{D-1})$ with the arc $R(i) = (x(i-1), R(i-1))$ to each of the $\Delta - 1$ PNDSTs in $TB(D(i-1), R(i-1))$. We denote these augmented trees by $TB^*(D(i-1), R(i-1)) = \{t^{*j}(i-1), j = 1, 2 \ldots, \Delta - 1\}$, and we say that the arc $R(i)$ is at the 0th level of trees in $TB^*(D(i-1), R(i-1))$. We find the line digraph $t_L^j(i)$ of each tree in $TB^*(D(i-1), R(i-1))$, denote the resulting set by $TB_L(D(i), R(i)) = \{t_L^j(i), j = 1, 2 \ldots, \Delta - 1\}$, and refer to the set of the trees as the *line trees*.

We make the following observations: 1. Let $t_L^j(i) \in TB_L(D(i), R(i))$ be the line tree of the tree $t^{*j}(i-1) \in TB^*(D(i-1), R(i-1))$; then we have $V(t_L^j(i)) = E(t^{*j}(i-1))$, where $E(t^{*j}(i-1))$ is the arc set of tree $t^{*j}(i-1)$ which contains all of the arcs in $t^j(i-1)$ plus the arc $R(i)$. 2. Each digraph $t_L^j(i)$ in $TB_L(D(i), R(i))$ is a subtree of $B(\Delta, D(i))$ rooted at $R(i)$. 3. Since the trees in $TB(D(i-1), R(i-1))$ are pseudo node disjoint and therefore also arc disjoint, the line trees in $TB_L(D(i), R(i))$ are strictly node disjoint except at $R(i)$, and thus each node of $B(\Delta, D(i))$ except $R(i)$ is in at most one tree in $TB_L(D(i), R(i))$. 4. The depth of the trees $TB_L(D(i), R(i))$ are the same as those of the trees in $TB(D(i-1), R(i-1))$, and an arc at level m in a tree in $TB^*(D(i-1), R(i-1))$ becomes a node at the same level in a tree in $TB_L(D(i), R(i))$.

Step 2. We extend the line trees $t_L^j(i) \in TB_L(D(i), R(i))$ as follows. We take each node, except $R(i)$, in each $t_L^j(i)$ and add as many as possible outgoing arcs which are available at that node in $B(\Delta, D(i))$ and whose other end nodes are not already in $t_L^j(i)$, to this node. The resulting digraph is denoted by $t_{LE}^j(i)$. We do this for all line trees $t_L^j(i) \in TB_L(D(i), R(i))$, and we denote the set of $t_{LE}^j(i)$ for all j by $TB_{LE}(D(i), R(i))$. We call this set of digraphs the *extended line trees*, and we will show later that the extended line trees are in fact trees within the observation 1 below and its proof.

We now make the following observations; the proofs for observations 1 and 5 are given at the end of this algorithm and the proofs of the other observations are simple and thus are left out: 1. Each extended line tree $t_{LE}^j(i) \in TB_{LE}(D(i), R(i))$ is a subtree of $B(\Delta, D(i))$ rooted at $R(i)$. 2. The extended line trees in $TB_{LE}(D(i), R(i))$ are pseudo node disjoint because the line trees in $TB_L(D(i), R(i))$ are strictly node disjoint and we only do breadth-first extension to the nodes in $TB_L(D(i), R(i))$; thus no nodes can be internal nodes in more than one tree. 3. From observation 1 in Step

1, these nodes added to a line tree $t_L^j(i)$ correspond to the arcs which did not belong to the tree $t^{*j}(i-1)$ in $TB^*(D(i-1), R(i-1))$, and they are leaf nodes in the extended line tree $t_{LE}^j(i)$ in $TB_{LE}(D(i), R(i))$. 4. The depths of the extended line trees increase by one from those of the trees in $TB(D(i-1), R(i-1))$. 5. The nodes of $B(\Delta, D(i))$ which do not already belong to an extended line tree $t_{LE}^j(i) \in TB_{LE}(D(i), R(i))$, denoted by $V_{\text{mis}}(t_{LE}^j(i))$, are given by

$$V_{\text{mis}}(t_{LE}^j(i)) = \{u = (r_{p-i+2}, \ldots, r_D, x) \mid x \in \{1, 2, \ldots, \Delta\}, u \text{ is not a subroot of } t_{LE}^j(i)\}.$$

Note that the set $V_{\text{mis}}(t_{LE}^j(i))$ corresponds to the outgoing arcs from $R(i-1)$ in $B(\Delta, D(i-1))$.

Step 3. Consider the set of adopters

$$A_d(R(i)) = \{a_d^j(i) = (dj, r_{p-i+2}, \ldots, r_D) \mid d_j \in \{1, 2, \ldots, \Delta\} \text{ and } d_j \neq r_{p-i+1}\}.$$

Note that the set of adopters corresponds to the incoming arcs into $R(i-1)$ in $B(\Delta, D(i-1))$, and such arcs do not belong to any tree in $TB(D(i-1), R(i-1))$. From observation 3 in Step 2, the adopters are leaves in every tree of $TB_{LE}(D(i), R(i))$. Furthermore, because $R(i-1)$ is not a corner node and thus has no loop at it, $A_d(R(i)) \cap V_{\text{mis}}(t_{LE}^j(i)) = \emptyset \ \forall j$. Also note that the $\Delta - 1$ adopters are the other predecessors of $V_{\text{mis}}(t_{LE}^j(i))$, we can easily establish a 1–1 mapping between nodes in $A_d(R(i))$ and the extended line trees in $TB_{LE}(D(i), R(i))$ by arbitrarily selecting one adopter $a_d^j(i) \in A_d(R(i))$ to map to one tree $t_{LE}^j(i) \in TB_{LE}(D(i), R(i)) \forall j$ and add the missing nodes $V_{\text{mis}}(t_{LE}^j(i))$ to $t_{LE}^j(i)$ by adding the arcs from $a_d^j(i)$ to each missing node of $V_{\text{mis}}(t_{LE}^j(i))$, thus making $t_{LE}^j(i)$ a spanning tree, denoted by $t^j(i)$. The set of these new trees $t^j(i)$ is denoted by $TB(D(i), R(i))$. From observation 2 in Step 2, it is clear that the trees in $TB(D(i), R(i))$ are PNDSTs. Obviously, if adopter $a_d^j(i)$ at the last level of $t_{LE}^j(i)$ is selected for $t_{LE}^j(i)$, then after adding the missing nodes $V_{\text{mis}}(t_{LE}^j(i))$ to $a_d^j(i)$ in $t_{LE}^j(i)$, the depth of $t^j(i)$ will increase by one from that of $t_{LE}^j(i)$. Otherwise, the depth of $t^j(i)$ will remain the same as that of $t_{LE}^j(i)$.

Proof of observations 1 *and* 5 *of Step* 2. To prove observation 1 in Step 2, we must show that the resulting graph $t_{LE}^j(i) \in TB_{LE}(D(i), R(i))$ is a rooted tree, in particular, we are going to show that there is no node, say $w \in B(\Delta, D(i))$, which is not in $t_L^j(i)$ that can have two or more distinct predecessors in $t_L^j(i)$. Suppose otherwise; then there must be nodes $u \in t_L^j(i)$ and $v \in t_L^j(i)$ and a node $w \notin t_L^j(i)$ such that arcs (u, w) and $(v, w) \in B(\Delta, D(i))$. Note that the nodes u, v, and w correspond to arcs, say e_u, e_v, and e_w in $B(\Delta, D(i-1))$, and that both arcs e_u and e_v are directed to the tail node of arc e_w in $B(\Delta, D(i-1))$. On the other hand, since nodes u and $v \in t_L^j(i)$, from observation 1 of Step 1, we have that both arcs e_u and e_v belong to $t^j(i-1)$. This is impossible because $t^j(i-1)$ is a tree and thus each node has only one incoming arc in it.

We will now prove observation 5 of Step 2. Note that each node u of $B(\Delta, D(i))$ corresponds to an arc $e_u = (u', u'')$ of $B(\Delta, D(i-1))$. We consider two cases. *Case* 1: node $u' \neq R(i-1)$. Since $u' \in B(\Delta, D(i-1))$ belongs to every tree in $TB(D(i-1), R(i-1))$, u' has $\Delta - 1$ incoming arcs in $TB(D(i-1), R(i-1))$, one arc belonging to each tree in the set. Note that these $\Delta - 1$ arcs are adjacent to arc e_u through node u' in $B(\Delta, D(i-1))$. Therefore, these $\Delta - 1$ arcs become $\Delta - 1$ nodes, one node in each tree in $TB_L(D(i), R(i))$ after Step 1, and thus node u belongs to all of the $\Delta - 1$ trees

of $TB_{LE}(D(i), R(i))$ definitively after Step 2 in this case. *Case* 2: if $u' = R(i-1)$, those incoming arcs to $u' = R(i-1)$ correspond to the set of nodes $R(i) \cup A_d(R(i))$, and these arcs do not belong to the trees in $TB(D(i-1), R(i-1))$. In this case, node u must belong to the set $S_b(R(i))$. For any $t^j_{LE}(i) \in TB_{LE}(D(i), R(i))$, note that $t^j_{LE}(i)$ has already had one (or possibly two) member(s) of $S_b(R(i))$ in it as its subroot(s). From observation 3 of Step 2, the nodes in $A_d(R(i))$ are leaf nodes without children in $TB_{LE}(D(i), R(i))$; thus all the other members of $S_b(R(i))$ have to be absent from $t^j_{LE}(i) \in TB_{LE}(D(i), R(i))$. This proves that $V_{\text{mis}}(t^j_{LE}(i))$ is as claimed. □

A trivial bound of $2D-1$ on the depths of the trees in $TB(D, R)$ may be established as follows. From the above algorithm, observe that the depths increase by one in Step 2 and by at most one in Step 3. Since initially the depths in $TB(D(0), R(0))$ are $D-p+1$, after p stages, the depths will increase to at most $D-p+1+2p = D+p+1 \le 2D-1$ since $p \le D-2$. However, we will prove a much stronger result in the following theorem.

THEOREM 3.1. *There are* $\Delta-1$ *PNDSTs rooted at any node in* $B(\Delta, D)$ *of depths not exceeding* $\lceil \frac{3D}{2} \rceil$.

Proof. We prove this theorem by induction on the stage number, i, $0 \le i \le p$. Note the that the diameter of $B(\Delta, D(0))$ is $D(0) = D-p$, the depth of the trees in $TB(D(0), R(0))$ is $D-p+1$ by Theorem 2.2, and since $D-p \ge 2$, $D-p+1 \le \lceil \frac{3}{2}(D-p) \rceil$. Thus the theorem is correct when $i = 0$.

Assume that Theorem 3.1 is true for $TB(D(i-1), R(i-1))$. The proof of observations 1 and 5 of Step 2 of the algorithm implies that we will be able to obtain the trees in $TB(D(i), R(i))$ from $TB(D(i-1), R(i-1))$; this would in turn imply the correctness of the algorithm. We will now prove the bound on the depth of the trees in $TB(D, R)$.

If in the ith stage, an adopter $a^j_d(i) = (dj, r_{p-i+2}, \ldots, r_D)$, $dj \ne r_{p-i+1}$, at the last level of a tree $t^j_{LE}(i) \in TB_{LE}(D(i), R(i))$ is chosen, then the depth of $t^j(i) \in TB(D(i), R(i))$ will increase by two from the depth of $t^j(i-1) \in TB(D(i-1), R(i-1))$ in stage i. In this case, note that the nodes in the last level of $t^j(i)$ are the nodes $u \in V_{\text{mis}}(t^j_{LE}(i))$, that is, $u = (r_{p-i+2}, \ldots, r_D, s_h)$ for some s_h, and u is not a subroot of $t^j_{LE}(i)$. Then after Step 2 of the next stage, stage $i+1$, the nodes at the last level of tree $t^j_{LE}(i+1) \in TB_{LE}(D(i+1), R(i+1))$ will have the form

$$u' = (r_{p-i+2}, r_{p-i+3}, \ldots, r_D, s_h, x), \quad x \in \{1, 2, \ldots, \Delta\}. \tag{4}$$

We claim that the nodes at the last level of $t^j_{LE}(i+1)$ cannot belong to the adopter set $A_d(R(i+1))$. This would mean that the depth of $t^j_{LE}(i+1)$ will not increase by the action of Step 3 of stage $i+1$, which in turn implies that the depth of $t^j(i+1)$ will increase by at most three in two contiguous stages from that of $t^j(i-1)$. Therefore, after $p \le D-2$ stages, the depths will increase by at most $\lceil \frac{3p}{2} \rceil$. Since the depths of the initial tree rooted at an $a-b$ node are $D-p+1$, the depths of the trees in $TB(D, R)$ are less than or equal to $D-p+1+\lceil \frac{3p}{2} \rceil = D+\lceil \frac{p}{2} \rceil + 1 \le D+\lceil \frac{D-2}{2} \rceil + 1 \le \lceil \frac{3D}{2} \rceil$.

To prove this claim, assume $u' \in A_d(R(i+1))$. By the definition of the adopter set,

$$u' = (d_j, r_{p-i+1}, \ldots, r_D), \quad d_j \ne r_{p-i}. \tag{5}$$

From equations (4) and (5), we have $r_{p-i+1} = r_{p-i+3}$, $r_{p-i+2} = r_{p-i+4}$, and $r_{p-i+3} = r_{p-i+5}, \ldots, r_{D-2} = r_D$. However, this would imply that the root $R(i) = (r_{p-i+1}, r_{p-i+2}, \ldots, r_D)$ is an $a-b$ node like $(abab\ldots)$. This is a contradiction since $R(i)$ is not an $a-b$ node for $i \ge 1$. □

Our next goal in this section is to show that there are Δ ADSTs in $K(\Delta, D)$ rooted at an arbitrary node $R = (r_1, r_2, \ldots, r_D)$. This set of trees will be denoted by $TK(D, R)$. Two sets of nodes around R in $K(\Delta, D)$ are of particular interest (as in the deBruijn digraph case): the subroots $\mathbf{S_b(R)}$,

$$S_b(R) = \Gamma(R) = \{u \mid u = (r_2, \ldots, r_D, s_i), s_i \in \{1, 2, \ldots, \Delta + 1\} \text{ and } s_i \neq r_D\};$$

and the adopters $\mathbf{A_d(R)}$,

$$A_d(R) = \{u \mid u = (d, r_2, \ldots, r_D), d \in \{1, 2, \ldots, \Delta + 1\}, d \neq r_1 \text{ and } d \neq r_2\},$$

where the sequences are Kautz sequences.

Let p be the minimum nonnegative integer such that the last $D-p$ letters in $R = (r_1, r_2, \ldots, r_D)$ form an $a-b$ Kautz sequence of two letters a and b in $\{1, 2, \ldots, \Delta+1\}$. Let $R(0) = (r_{p+1}, r_{p+2}, \ldots, r_D)$ represent this sequence and note that $D-p \geq 2$. Our process for obtaining the Δ ADST in $TK(D, R)$ is similar to that in the previous case.

THEOREM 3.2. *There are Δ ADSTs rooted at any node in $K(\Delta, D)$ of depths not exceeding $\lceil \frac{3D}{2} \rceil + 1$.*

Proof. We follow the same algorithm as in the previous case with the following two exceptions: 1. We begin with the set $TK(R(0), D(0))$, where $R(0)$ is an node represented by an $a - b$ Kautz sequence. By Theorem 2.3, $TK(R(0), D(0))$ consists of Δ ADSTs of depths $D(0) + 2 = D - p + 2$. 2. In this case, in Step 3 of the ith stage, $1 \leq i \leq p$,

$$A_d(R(i))$$
$$= \{a_d^j(i) = (d_j, r_{p-i+2}, \ldots, r_D) \mid d_j \in \{1, 2, \ldots, \Delta + 1\}, d_j \neq r_{p-i+1} \text{ and } d_j \neq r_{p-i+2}\}.$$

Thus we have $|A_d(R(i))| = \Delta - 1$ adopters, but we have a set of Δ extended line trees in $TK_{LE}(D(i), R(i))$. Therefore, we cannot expect to establish the 1–1 mapping between the adopters $A_d(R(i))$ and the extended line trees $TK_{LE}(D(i), R(i))$ as before. To overcome this difficulty, for $j = 1, \ldots, \Delta - 1$, we chose one adopter, say $a_d^j(i) \in A_d(R(i))$, for tree $t_{LE}^j(i) \in TK_{LE}(D(i), R(i))$ with subroot $S_j = (r_{p-i+2}, \ldots, r_D, s_j)$ to add the missing node set $V_{\text{mis}}(t_{LE}^j(i)) = \Gamma(R(i)) - \{S_j\}$ to $t_{LE}^j(i)$. This leads to the spanning trees $t^j(i) \in TK(D(i), R(i))$ for $j = 1, \ldots, \Delta - 1$.

We now consider the last extended line tree $t_{LE}^{\Delta}(i) \in TK_{LE}(D(i), R(i))$. Observe that the arc $(a_d^j(i), S_j)$ is not used for constructing $t^j(i)$ for $j = 1, \ldots, \Delta - 1$ and that $V_{\text{mis}}(t_{LE}^{\Delta}(i)) = \Gamma(R(i)) - S_{\Delta} = \{u \mid u = S_j, j = 1, \ldots, \Delta - 1\}$. To construct $t^{\Delta}(i)$, we start with $t_{LE}^{\Delta}(i)$ and then add the arcs $(a_d^j(i), S_j), j = 1, \ldots, \Delta - 1$, to $t_{LE}^{\Delta}(i)$ to obtain the spanning tree $t^{\Delta}(i)$. This produces the set of Δ ADSTs $TK(D(i), R(i)) = \{t^j(i), j = 1, \ldots, \Delta\}$.

Since the remaining parts in the proof are identical to the proof of Theorem 3.1, they will not be given. Due to difference 1 above, the bound on the depths is $\lceil \frac{3D}{2} \rceil + 1$. □

We now briefly consider the disjoint spanning trees in the undirected deBruijn and Kautz graphs. We define undirected deBruijn graphs $UB(\Delta, D)$ and Kautz graphs $UK(\Delta, D)$ as follows. We begin with the digraph $B(\Delta, D)$ (respectively, $K(\Delta, D)$). Then 1. ignore the directions of the arcs, thus replacing arcs with edges; 2. if for some nodes u and v in $B(\Delta, D)$ (respectively, in $K(\Delta, D)$), both arcs (u, v) and (v, u) belong to $B(\Delta, D)$ (respectively, to $K(\Delta, D)$), we would have both edges joining u and

v in $UB(\Delta, D)$ (respectively, in $UK(\Delta, D)$); and 3. remove the loops from $B(\Delta, D)$. We note that graphs $UB(\Delta, D)$ and $UK(\Delta, D)$ are not simple graphs and $UB(\Delta, D)$ is no longer a regular graph because the corner nodes have degree equal to $2(\Delta - 1)$ while the others have degree equal to 2Δ.

By Theorem 3.1, it is clear that there are $\Delta - 1$ PNDSTs of depth at most $\lceil \frac{3D}{2} \rceil$ in $UB(\Delta, D)$ rooted at an arbitrary node R. We will now show that there are Δ PNDSTs in $UB(\Delta, D)$. We note that there are $|V| = \Delta^D$ nodes and $|E| = \Delta^{D+1} - \Delta = \Delta(\Delta^D - 1) = \Delta(|V| - 1)$ edges in $UB(\Delta, D)$, and thus total number of edges $|E|$ in $UB(\Delta, D)$ is exactly the number of edges required in the Δ edge-disjoint spanning trees. Thus we know that there are at most Δ PNDST in $UB(\Delta, D)$. Actually, we have the following result, which is a direct consequence of a result in [1].

COROLLARY 3.3. (See [1, Proposition 5.1].) *There are Δ PNDSTs rooted at any node R in $UB(\Delta, D)$ of depths no larger than $2D$.*

Proof. We begin with the digraph $B(\Delta, D)$. For $a \in \{1, 2, \ldots, \Delta\}$, let $T(a)$ be the shortest-path spanning tree rooted at the corner node $(aa \ldots a)$. It is known that $T(1), T(2), \ldots, T(\Delta)$ are all arc disjoint and of depth D [1]. Furthermore, it can be seen that each node in $T(a)$ except the root $(aa \ldots a)$ is of degree either $\Delta + 1$ or 1. Let the undirected tree $UT(a)$ be obtained from $T(a)$ by ignoring the directions of arcs in $T(a)$. Note that the diameter of the tree $UT(a)$ is exactly $2D$. Let R be an arbitrary node in $UB(\Delta, D)$; then the set of trees $\{UT(1), UT(2), \ldots, UT(\Delta)\}$ can be considered to be a set of edge-disjoint spanning trees rooted at R of depths less than or equal to $2D$. Furthermore, since each internal node in $T(a)$ has degree $\Delta + 1$, no node can be an internal node in more than one tree; thus the above set of trees are pseudo-node-disjoint trees in $UB(\Delta, D)$. □

We now consider the case of $UK(\Delta, D)$. Note that the numbers of nodes and edges in $UK(\Delta, D)$ are $|V| = \Delta^D + \Delta^{D-1}$ and $|E| = \Delta^{D+1} + \Delta^D$, respectively, and the Δ ADSTs $TK(\Delta, R)$ have used $T = \Delta(|V| - 1) = |E| - \Delta$ edges; thus there are only Δ free edges which do not belong to the directed trees in $TK(\Delta, R)$. Actually, they are the Δ incoming arcs to the root R. The following result is a direct consequence of Theorem 3.2.

COROLLARY 3.4 (corollary to Theorem 3.2). *There are Δ edge-disjoint spanning trees rooted at any node R in $UK(\Delta, D)$ of depths not exceeding $\lceil \frac{3D}{2} \rceil + 1$.*

4. Conclusions and fault-tolerance considerations. The deBruijn and Kautz graphs and digraphs have become contenders for interconnect architectures of multiprocessor systems and computer networks. Since broadcasting is an important function in such systems, the results of this paper further enhance the viability of these graphs for such applications.

The results of this paper also have fault-tolerance implications. The fact that there are $\Delta - 1$ pseudo-node-disjoint trees rooted at a particular node R in $B(\Delta, D)$ implies that the network can tolerate up to $\Delta - 2$ node failures and still perform its broadcast function. More precisely, let $\mathbf{DN}(\mathbf{f}, \mathbf{G})$ (respectively, $\mathbf{DL}(\mathbf{f}, \mathbf{G})$) be the maximum depth of a spanning tree rooted at arbitrary node R in a graph $\mathbf{G}$ if up to $\mathbf{f}$ nodes (respectively, links) failed. This notion is similar to the previously introduced notion of f-node-diameter vulnerability [6, 8]. Our results imply that $DN(f, B(\Delta, D)) \leq \lceil \frac{3D}{2} \rceil$ if $f = \Delta - 2$ and $DN(f, UB(\Delta, D)) \leq 2D$ if $f = \Delta - 1$. The same relations apply to $DL(f, B(\Delta, D))$ and $DL(f, UB(\Delta, D))$. It is easy to see that $DL(f, K(\Delta, D)) \leq \lceil \frac{3D}{2} \rceil + 1$ if $f = \Delta - 1$, and a similar statement would also hold for $UK(\Delta, D)$. However, the situation for $DN(f, K(\Delta, D))$ is a bit more complex.

We claim that $DN(f, K(\Delta, D)) \leq \lceil \frac{3D}{2} \rceil + 1$ if $f = \Delta - 2$. To see the validity of

this claim, we refer the reader to the last stage, stage p, in our algorithm. After Step 2 of the algorithm, the trees in $TK_{LE}(D(p+1), R(p+1))$ are pseudo node disjoint. However, after Step 3, each adopter, say $a_d^j(p+1)$, becomes an internal node in both the tree $t^j(p+1)$ and the tree $t^{\Delta}(p+1)$. Thus the failure of adopter $a_d^j(p+1)$ would destroy the two spanning trees $t^j(p+1)$ and $t^{\Delta}(p+1)$. However, each additional node failure would destroy at most one more tree. Thus even if $\Delta - 2$ nodes failed, there is at least one remaining tree of depth at most $\lceil \frac{3D}{2} \rceil + 1$. The above result also remain valid for $UK(\Delta, D)$. In all of the above cases, if the number of failures $f' < f$, then we would have $f - f' + 1$ remaining rooted spanning trees with the above depths.

Finally, the bounds on the depths in this paper are not tight bounds. In particular, we believe that the bound on the depths in Theorem 3.1 can be further improved if a more suitable 1–1 mapping between the adopters $A_d(R(i))$ and the extended line trees $TB_{LE}(D(i), R(i))$ in Step 3 of each stage of our algorithm is selected.

REFERENCES

[1] J-C. Bermond and P. Fraigniaud, *Broadcasting and gossiping in deBruijn networks*, SIAM J. Comput., 23 (1994), pp. 212–225.

[2] A.-H. Esfahanian and S. L. Hakimi, *Fault-tolerant routing in deBruijn communication networks*, IEEE Trans. Comput., 34 (1985), pp. 777–788.

[3] J-C. Bermond and C. Peyrat, *DeBruijn and Kautz networks: Competitor for the hypercube?*, in Hypercube and Distributed Computers, F. Andre and J. P. Verjus, eds., Elsevier–North–Holland, Amsterdam, 1989, pp. 279–293.

[4] M. R. Samatham and D. K. Pradhan, *The deBruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI*, IEEE Trans. Comput., 38 (1989), pp. 567–581.

[5] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed., Prentice–Hall, Englewood Cliffs, NJ, 1992.

[6] D.-Z. Du, Y.-D. Lyuu, and D. F. Hsu, *Line digraph iterations and the spread concept: With applications to graph theory, fault tolerance, and routing*, in Graph Theoretic Concepts in Computer Science, G. Schmidt and R. Berghammer, eds., Springer-Verlag, Berlin, 1991, pp. 169–179.

[7] R. L. Hemminger and L. W. Beineke, *Line graphs and line digraphs*, in Selected Topics In Graph Theory, L. W. Beineke and R. J. Wilson, eds., Academic Press, London, 1978.

[8] M. Imase, T. Soneoka, and K. Okada, *Fault-tolerant processor interconnection networks*, Systems Comput. Japan, 17 (1986), pp. 21–30.

[9] Q. F. Stout and B. Wagar, *Intensive hypercube communication*, J. Parallel Distrib. Comput., 10 (1990), pp. 167–181.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 93–109, February 1997

006

POLYNOMIAL-TIME RECOGNITION OF 2-MONOTONIC POSITIVE BOOLEAN FUNCTIONS GIVEN BY AN ORACLE*

ENDRE BOROS†, PETER L. HAMMER†, TOSHIHIDE IBARAKI‡, AND KAZUHIKO KAWAKAMI‡

Abstract. We consider the problem of identifying an unknown Boolean function f by asking an oracle the functional values $f(a)$ for a selected set of test vectors $a \in \{0,1\}^n$. Furthermore, we assume that f is a positive (or monotone) function of n variables. It is not yet known whether or not the whole task of generating test vectors and checking if the identification is completed can be carried out in polynomial time in n and m, where $m = |\min T(f)| + |\max F(f)|$ and $\min T(f)$ (respectively, $\max F(f)$) denotes the set of minimal true (respectively, maximal false) vectors of f. To partially answer this question, we propose here two polynomial-time algorithms that, given an unknown positive function f of n variables, decide whether or not f is 2-monotonic and, if f is 2-monotonic, output both sets $\min T(f)$ and $\max F(f)$. The first algorithm uses $O(nm^2 + n^2m)$ time and $O(nm)$ queries, while the second one uses $O(n^3m)$ time and $O(n^3m)$ queries.

Key words. 2-monotonic Boolean function, oracle, polynomial-time identification

AMS subject classifications. 68T05, 68Q25, 90C09

PII. S0097539793269089

1. Introduction.

In this paper, we investigate the problem of identifying an unknown Boolean function f by successively constructing test vectors $a \in \{0,1\}^n$ and asking an oracle their functional values $f(a)$ (i.e., membership queries). We propose two polynomial-time algorithms for a specific class of 2-monotonic positive Boolean functions.

Recall that a Boolean function (or simply a *function*) f of n variables is a mapping $f{:}\{0,1\}^n \to \{0,1\}$. We shall write $g \leq f$ if $g(a) = 1$ implies $f(a) = 1$ for all vectors $a \in \{0,1\}^n$. If $g \leq f$ and there exists a vector a satisfying $g(a) = 0$ and $f(a) = 1$, we shall write $g < f$. A function f is called *positive* (or *monotone*) if $a \leq b$ (i.e., $a_i \leq b_i$ for $i = 1, 2, \ldots, n$) always implies $f(a) \leq f(b)$. A vector $a \in \{0,1\}^n$ is a *true* (resp. *false*) vector if $f(a) = 1$ (resp. $f(a) = 0$) holds. The set of true vectors and false vectors of f are, respectively, denoted $T(f)$ and $F(f)$. A true vector is called a *minimal* (resp. *maximal*) if there is no true vector (resp. false vector) b such that $b \leq a$ (resp. $b \geq a$) and $b \neq a$. The sets of minimal true vectors and maximal false vectors are, respectively, denoted $\min T(f)$ and $\max F(f)$. The definition of 2-monotonicity is given in section 2. We only point out here that the class of 2-monotonic positive functions properly includes the class of positive threshold functions [20, 26].

The problem of identifying Boolean functions arises in various settings of theory and practice. A first application is the testing of logic circuits, i.e., the identification of

* Received by the editors May 4, 1993; accepted for publication (in revised form) April 17, 1995. This research was partially supported by AFOSR grants 89-0512B and F49620-93-1-0041 and ONR grants N00014-92-J-1375 and N00014-92-J-4083. A preliminary version of this paper appeared in *ISA* '91 *Algorithms*, Lecture Notes in Comput. Sci. 557, Springer-Verlag, Berlin, 1991, pp. 104–115 [7].

http://www.siam.org/journals/sicomp/26-1/26908.html

† RUTCOR, Rutgers University, New Brunswick, NJ 08903 (boros@rutcor.rutgers.edu, hammer@rutcor.rutgers.edu).

‡ Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, Kyoto 606, Japan (ibaraki@kuamp.kyoto-u.ac.jp, kawakami@kuamp.kyoto-u.ac.jp). The research of these authors was partially supported by a Scientific Grant-in-Aid of the Ministry of Education, Science, and Culture of Japan.

the Boolean function realized by the circuit. This identification requires the selection of a (possibly small) set of binary vectors on which the circuit is to be tested. Another example is found in the process of forming a "concept" from partially observed data [6, 11], in which hypotheses for the functional form of a hidden Boolean function are generated. It is assumed that the hidden function belongs to a specified subclass of Boolean functions, known a priori.

Probably the most rigorous mathematical basis for our problem is provided by the recent development of computational learning theory. The problem discussed in this paper is an example of exact learning (see, e.g., [1]), in which only membership queries are allowed. Our result shows that a polynomial-time exact learning of this type is possible for the class of 2-monotonic positive Boolean functions.

If no a priori information is available about the Boolean function f, it is obvious that it cannot be identified unless the values $f(a)$ for all 2^n vectors $a \in \{0,1\}^n$ are tested. Therefore, the problem becomes interesting only when some knowledge about f is at hand. An important class in the above problem setting of concept formation and learning theory is that of positive functions.

In this paper, a positive function will be considered *identified* if both the sets of minimal true vectors, $\min T(f)$, and maximal false vectors, $\max F(f)$, are explicitly obtained. We shall call these two sets the output of the identification algorithm, where the length of output is denoted by

$$m = |\min T(f)| + |\max F(f)|. \tag{1}$$

This definition, while it may appear somewhat redundant since either of these sets can be computed from the other, is justified by the following two reasons. First of all, in general, the computation of one of these sets from the other can take exponential time in the size of these sets. Second, the knowledge of both sets, $\min T(f)$ and $\max F(f)$, is necessary to ensure that the function obtained at the end of the algorithm is indeed completely specified. It is known that the total size m of these sets can become as large as

$$\binom{n}{\lfloor \frac{n}{2} \rfloor} + \binom{n}{\lfloor \frac{n}{2} + 1 \rfloor},$$

and therefore polynomiality in n only cannot be expected. However, the complexity of an identification algorithm in terms of both the input and the output sizes is not known.

The papers by Angluin [1] and Gainanov [14] contain an algorithm that identifies $\min T(f)$ of a positive function f of n variables by issuing $O(n|\min T(f)|)$ membership and equivalence queries. (Equivalence queries test whether the target function f is equivalent to the hypothesis f'.) This means that the identification can be done in polynomial time if both membership and equivalence queries are allowed. An explanation for this is the fact that if f and f' are not equivalent, the equivalence oracle returns a binary vector a at which f and f' disagree and which is then sent to the membership oracle. However, when only membership queries are allowed, the time to generate such query vectors must be taken into account, and, in spite of intensive studies on this topic (see, e.g., [4, 14, 15, 16, 24]), it is not yet known whether or not this can be accomplished in polynomial time. Note, however, that upon measuring the complexity only by n and $m_1 = |\min T(f)|$ (instead of m), Angluin obtained a negative result (see [1]), showing that there exists a positive Boolean function f which cannot be identified by a polynomial (in n and m_1) number of membership queries

even if unlimited computation time is allowed. The underlying reason for this is that, for any vector $a \in \max F(f)$, there exists another positive Boolean function f' for which $\min T(f') = \min T(f) \cup \{a\}$. Hence the information necessary to distinguish between f and f' assumes a membership query for all such vectors a. This implies that the identification of f requires at least as many as $m_0 = |\max F(f)|$ membership queries, and this quantity m_0 is known to be exponential in n and m_1 for many positive Boolean functions. This is another reason to introduce in this paper the parameter $m = m_0 + m_1$ for evaluating the complexity of an identification algorithm.

It is being realized that the problem of identifying a general positive function by membership queries is equivalent to many other problems in the sense that the former is solvable in polynomial time in n and m if and only if the latter problems are solvable in polynomial time. Among the many problems of this type (see, e.g., [4, 12]), we mention the dualization of positive Boolean functions, the recognition of self-dual positive functions, the recognition of saturated simple hypergraphs, and so forth. Although the exact complexity of these problems is still open, the recent result by Fredman and Khachiyan [13] shows that these problems can be solved in $O(m^{o(\log m)})$ time, suggesting that it is quite unlikely that they are NP-hard.

The main results in this paper are two polynomial-time algorithms (more precisely, incrementally polynomial-time algorithms [17, 18]), which, for a given unknown positive function f of n variables, decide whether or not f is 2-monotonic or not and, if it is 2-monotonic, output both $\min T(f)$ and $\max F(f)$. The first algorithm uses $O(nm^2 + n^2m)$ time and $O(nm)$ queries, while the second one uses $O(n^3m)$ time and $O(n^3m)$ queries. (Throughout this paper, the stated computation time does not include the time spent on the oracle to answer the given membership queries.) The proposed algorithms make use of the results of Gainanov [14] and Valiant [25] to generate a vector in $\min T(f) \cup \max F(f)$ and of a new characterization of 2-monotonic positive functions in terms of their sets of minimal true and maximal false vectors. They are also related to the results in [3, 5, 10, 21, 22] showing that the dualization of 2-monotonic positive functions can be done in $O(mn)$ time. We also note that there are some other classes of positive functions for which polynomial-time identification algorithms are known [19], which are based on the concept of maximum latency of function classes.

In concluding this section, we comment that there is a wide spectrum of research about the exact learning of Boolean functions. Most of this research, however, is based on the model of using both membership and equivalence queries. In this model, in addition to the class of positive functions, there are a number of classes such as read-once functions (see, e.g., [2, 9]), which are learnable in polynomial time in n and the length of the formula expressing the function. Recently, Bshouty [8] showed that any Boolean function is polynomially learnable either as DNF (disjunctive normal form) or CNF (conjunctive normal form).

2. Definitions and basic properties. Let f be a positive function of n variables. f is completely characterized by one of the sets $\min T(f)$ and $\max F(f)$ since f is defined, for example, by

$$f(a) = \begin{cases} 1 & \text{if } a \geq b \text{ for some } b \in \min T(f), \\ 0 & \text{otherwise.} \end{cases}$$

It is known in Boolean algebra that another characterization of a positive function f is that f has a disjunctive form in which all literals appear uncomplemented. In this case, each prime implicant of f corresponds one to one to a minimal true vector of f.

The *dual* f^d of a function f is defined by

$$f^d(x) = \bar{f}(\bar{x}),$$

where $\bar{f}$ (respectively, $\bar{x}$) denotes the complement of f (respectively, x). The Boolean expression of f^d is obtained from that of f by exchanging $\wedge$ (and) and $\vee$ (or) as well as the constants 1 and 0.

An assignment A of binary values 0 or 1 to k variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ out of all n variables is called a *k-assignment* and is denoted by

$$A = \{x_{i_1} \leftarrow a_1, x_{i_2} \leftarrow a_2, \ldots, x_{i_k} \leftarrow a_k\}, \tag{2}$$

where each of the values $a_1, \ldots, a_k$ is either 1 or 0. Let the complement of A, denoted by $\bar{A}$, represent the assignment obtained from A by complementing all the 1's and 0's of A. When a function $f(x)$ of n variables and a k-assignment A are given,

$$f_A(x) = f(x; x_{i_1} \leftarrow a_1, x_{i_2} \leftarrow a_2, \ldots, x_{i_k} \leftarrow a_k)$$

denotes the function of $(n-k)$ variables obtained by fixing the variables x_{i_1}, x_{i_2}, ..., x_{i_k} as specified by A.

Let f be a Boolean function of n variables. If either $f_A \leq f_{\bar{A}}$ or $f_A \geq f_{\bar{A}}$ holds for every k-assignment A, then f is said to be *k-comparable*. If a function f is k-comparable for every k such that $1 \leq k \leq m$, then f is said to be *m-monotonic*. (For more detailed discussion on these topics, see, e.g., [20, 26].) In particular, f is 1-monotonic if $f_{(x_i \leftarrow 1)} \geq f_{(x_i \leftarrow 0)}$ or $f_{(x_i \leftarrow 1)} \leq f_{(x_i \leftarrow 0)}$ holds for any $i \in \{1, 2, \ldots, n\}$. It can be shown that if f is positive, then it is 1-monotonic and $f_{(x_i \leftarrow 1)} \geq f_{(x_i \leftarrow 0)}$ holds for every i.

Now consider a 2-assignment $A = \{x_i \leftarrow 1, x_j \leftarrow 0\}$. The relation $f_A \geq f_{\bar{A}}$ (respectively, $f_A > f_{\bar{A}}$) will be denoted by $x_i \succeq_f x_j$ (respectively, $x_i \succ_f x_j$). Two variables x_i and x_j are said to be *comparable* if either $x_i \succeq_f x_j$ or $x_i \preceq_f x_j$ holds. When $x_i \succeq_f x_j$ and $x_i \preceq_f x_j$ hold simultaneously, we shall write $x_i \approx_f x_j$. If f is 2-monotonic, the binary relation $\succeq_f$ over the set of variables is known to be a total preorder. A 2-monotonic positive function f of n variables is called *regular* if

$$x_1 \succeq_f x_2 \succeq_f x_3 \succeq_f \cdots \succeq_f x_n. \tag{3}$$

Any 2-monotonic positive function becomes regular by permuting the variables.

As an example, consider a function $f = x_2 \vee x_1 x_3$ of three variables. It can be easily checked that $f(x; x_i \leftarrow 1) \geq f(x; x_i \leftarrow 0)$ for $i = 1, 2, 3$, and hence f is 1-monotonic and positive. The 2-monotonicity of f can be checked in the same way, and $x_2 \succ_f x_1 \approx_f x_3$ holds. Although f is not regular, after the relabeling $x'_1 = x_2$, $x'_2 = x_1$, and $x'_3 = x_3$, it becomes regular.

The property of 2-monotonicity was originally introduced in conjunction with threshold functions (e.g., [20, 26]). A positive function f is called *threshold* if there exist $n+1$ nonnegative real numbers $c_1, c_2, \ldots, c_n \geq 0$ and t such that:

$$f = \begin{cases} 1 & \text{if } \sum_{i=1}^n c_i x_i \geq t, \\ 0 & \text{if } \sum_{i=1}^n c_i x_i < t. \end{cases}$$

Since $c_i > c_j$ implies $x_i \succeq_f x_j$ and $c_i = c_j$ implies $x_i \approx_f x_j$, a threshold function is always 2-monotonic. The converse, however, is not true.

3. Outline of the algorithms. We shall present an outline of our algorithms which, for a given unknown positive function f, decide whether or not f is 2-monotonic and, if the function f is 2-monotonic, output $\min T(f)$ and $\max F(f)$. Both of the presented algorithms will be based on an oracle to obtain the values $f(a)$ for a selected set of vectors a, i.e., on membership queries. The details of the steps of the algorithms and the analysis of their time complexity will be given in the subsequent sections.

In the proposed algorithms, we shall maintain two subsets of vectors MT and MF such that

$$MT \subseteq \min T(f) \quad \text{and} \quad MF \subseteq \max F(f). \tag{4}$$

Let us define the sets of vectors T and F by

$$\begin{aligned} T &= \{a \in \{0,1\}^n | a \geq a' \text{ for some } a' \in MT\}, \\ F &= \{b \in \{0,1\}^n | b \leq b' \text{ for some } b' \in MF\}, \end{aligned} \tag{5}$$

and let us call a vector a *unknown* if

$$a \notin T \cup F$$

since $f(a)$ for such a vector a cannot be deduced from the knowledge of MT and MF. Given a binary vector $a = (a_1, a_2, \ldots, a_n)$, let $\bar{a} = (\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_n)$ denote its *complement*. Let us further define two positive functions g_1 and g_0 by

$$\begin{aligned} g_1(a) &= \begin{cases} 1 & \text{if } a \in T, \\ 0 & \text{otherwise,} \end{cases} \\ g_0(a) &= \begin{cases} 1 & \text{if } \bar{a} \in F, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{6}$$

In each iteration of the algorithms, given the current sets MT and MF, we test whether the two functions g_0 and g_1 satisfy the following two conditions:

(a) Both g_1 and g_0 are 2-monotonic.

(b) The orders of variables for g_1 and g_0 coincide, i.e., $x_i \preceq_{g_1} x_j$ if and only if $x_i \preceq_{g_0} x_j$ for any $i \neq j$.

If these conditions do not hold, we shall distinguish two cases:

(i) We can conclude that f is not 2-monotonic and the algorithms stop.

(ii) We can find an unknown vector a with respect to the current MT and MF. In this case, with the aid of membership queries, another vector c is generated from a for which we have

$$c \in (\min T(f) \cup \max F(f)) \setminus (MT \cup MF). \tag{7}$$

Then MT or MF is augmented with c, and the algorithms proceed to the next iteration.

On the other hand, if g_1 and g_0 satisfy both conditions (a) and (b) above, we test whether the current MT and MF satisfy

$$MT = \min T(f) \text{ and } MF = \max F(f). \tag{8}$$

The following outcomes are possible:

(iii) Condition (8) holds. Then $g_1 = f$ and f is identified (also $g_0 = g_1^d = f^d$ holds). Our algorithms stop here.

(iv) Condition (8) does not hold. Then an unknown vector a is found and the algorithms proceed as in (ii) above.

The above procedure is repeated until it stops in (i) or (iii).

The key tasks in an efficient implementation of these steps are the following:

Task 1: initializing the sets MT and MF.

Task 2: checking whether both conditions (a) and (b) hold for the current g_1 and g_0 of (6) and, if not, either concluding that f is not 2-monotonic or providing an unknown vector a.

Task 3: checking whether or not the termination condition (8) holds and, if not, providing an unknown vector a.

Task 4: given an unknown vector a, finding a vector c which satisfies (7).

These points will be separately discussed in the subsequent sections. As we shall see, Task 1 is quite easy, and a polynomial-time solution for Task 4 is already well known. The contribution of this paper consists mainly in providing polynomial-time algorithms for Tasks 2 and 3 above.

4. Construction of a minimal true vector or a maximal false vector. Given an unknown vector a of a positive function f, Gainanov [14] and Valiant [25] present an algorithm to enlarge the set $MT \cup MF$ by finding a vector c satisfying (7). It proceeds as follows.

Assume that an unknown vector $a^0 = a$ satisfies $f(a) = 1$. For $i = 1, 2, \ldots, n$, define

$$a^i = \begin{cases} a^{i-1} & \text{if } a_i = 0, \text{ or if } a_i = 1 \text{ and } f(a^{i-1} - e^i) = 0, \\ a^{i-1} - e^i & \text{if } a_i = 1 \text{ and } f(a^{i-1} - e^i) = 1, \end{cases}$$

where e^i denotes the ith unit vector. Then $c = a^n$ satisfies $c \in \min T(f)$ and $c \leq a$, implying that c satisfies (7).

The case of $f(a) = 0$ is treated symmetrically, and the algorithm produces a vector $c \in \max F(f)$. In either case, at most $n + 1$ vectors are tested by membership queries and the total time required for this task is $O(n)$.

As an example, consider a positive function f of five variables and assume that $MT = \emptyset$ and $\min T(f) = \{10100, 01010\}$. Let us test the vector $a = (11110)$. Clearly, $f(a) = 1$ (since $a > 10100$), and the following sequence is generated:

$$\begin{array}{ll} a^0 = (11110), & f(a^0) = 1, \\ a^1 = (01110) & \text{since } f(a^0 - e^1) = f(01110) = 1, \\ a^2 = (01110) & \text{since } f(a^1 - e^2) = f(00110) = 0, \\ a^3 = (01010) & \text{since } f(a^2 - e^3) = f(01010) = 1, \\ a^4 = (01010) & \text{since } f(a^3 - e^4) = f(01000) = 0, \\ a^5 = (01010) & \text{since } a_5 = 0. \end{array}$$

Consequently, $c = a^5 = (01010)$ is a minimal true vector.

5. An algorithm for identifying 2-monotonic functions. In this section, we present an algorithm IDENTIFY-1 for identifying a 2-monotonic positive function in $O(nm^2 + n^2m)$ time by asking $O(nm)$ queries. Another procedure that requires $O(n^3m)$ time and $O(n^3m)$ queries will be presented in the next section.

5.1. Initialization. If $MT = \emptyset$ and $MF = \emptyset$, any a is an unknown vector. It is convenient to start with $a^1 = (111\ldots1)$. If $f(a^1) = 0$, then the positivity of f implies that f is identically 0 (i.e., f is identified). Let us assume therefore that $f(a^1) = 1$,

and let c^1 be the vector (satisfying (7)) obtained by the algorithm described in section 4. A similar procedure is then applied to $a^0 = (000\ldots0)$. If $f(a^0) = 1$, then f is identically 1 (i.e., f is identified); otherwise, a vector c^0 satisfying (7) is produced. Our algorithms initialize MT and MF as

$$MT := \{c^1\} \quad \text{and} \quad MF := \{c^0\}.$$

5.2. Checking the 2-monotonicity of g_i. Since g_1 and g_0 of (6) can be treated in a similar manner, in this subsection, we shall refer to either of g_1 and g_0 as g and to either of the corresponding sets MT and

$$CMF = \{\bar{a} | a \in MF\}$$

as M. In other words, $\min T(g) = M$ holds. The algorithm described below decides if g is 2-monotonic or not, and if g is 2-monotonic, it also computes the $\preceq_g$ order of the variables. The cases in which g is not 2-monotonic or the orders $\preceq_{g_1}$ and $\preceq_{g_0}$ do not coincide will be discussed in the next two subsections.

Let us note that the existence of a pair of vectors a and b for which

$$a \in M, \qquad g(b) = 0, \tag{9}$$

and

$$\begin{array}{llll} a_i = 1, & a_j = 0, & b_i = 0, & b_j = 1, \\ a_k = b_k & \text{for } k \neq i, j & & \end{array} \tag{10}$$

implies $g_A \not\leq g_{\bar{A}}$ for the assignment $A = \{x_i \leftarrow 1, x_j \leftarrow 0\}$, that is, $x_i \not\preceq x_j$. Conversely, if $x_i \not\preceq x_j$, then there exist vectors a' and b' such that $g(a') = 1$ (possibly $a' \notin M$), $g(b') = 0$, and (10) holds for a' and b'. By the definition of g, there exists a vector $a \leq a'$ for which $a \in M$. For this vector a, $g(a) = 1$ holds and hence $a \not\leq b'$, which implies that $a_i = 1$ and $a_j = 0$. Let us then define a vector b by

$$b_k = \begin{cases} 0 & \text{if } k = i, \\ 1 & \text{if } k = j, \\ a_k & \text{otherwise.} \end{cases}$$

Since $b \leq b'$ follows from $a \leq a'$, the vectors a and b satisfy conditions (9) and (10). Therefore, the existence of a pair of vectors a and b satisfying (9) and (10) is a necessary and sufficient condition for $x_i \not\preceq x_j$. In other words, $x_i \preceq x_j$ if and only if there is no pair a and b that satisfy (9) and (10), which is then equivalent to saying that, for every vector $a \in M$ and indices $i \neq j$ for which

$$a_i = 1 \quad \text{and} \quad a_j = 0, \tag{11}$$

there exists a vector $d \in M$ satisfying

$$d_i = 0, \qquad d_j = 1, \quad \text{and} \quad d_k \leq a_k \quad \text{for } k \neq i, j. \tag{12}$$

We obtain the following.

LEMMA 5.1. *Given a positive Boolean function g with $M = \min T(g)$ and indices $i \neq j$, the following three conditions are equivalent:*

(i) $x_i \preceq_g x_j$.

(ii) *There is no pair of vectors a and b satisfying* (9) *and* (10).

(ii) *For every vector* $a \in M$ *satisfying* (11)*, there exists a vector* $d \in M$ *satisfying* (12).

Let us observe that the existence of indices $i \neq j$ for vectors $a, d \in M$ satisfying (11) and (12) is equivalent to

$$\begin{array}{ll} d_j = 1, & a_j = 0 \quad \text{hold for exactly one } j, \\ d_k \leq a_k & \text{for all } k \neq j \end{array} \tag{13}$$

since the minimality of vectors in M guarantees the existence of an index i for which $d_i < a_i$.

To test conditions (11) and (12), we shall construct for each $a \in M$ an $n \times n$ matrix $P(a)$ such that

$$P_{ij}(a) = \begin{cases} 1 & \text{if } a \text{ satisfies (11) and there is no } d \in M \text{ satisfying (12),} \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

and let

$$P = \sum_{a \in M} P(a). \tag{15}$$

The next lemma follows immediatly from Lemma 5.1 and from the definition of P.

LEMMA 5.2. *For a positive function* g*, we have*

(i) $x_i \prec_g x_j$ *if and only if* $P_{ij} = 0$ *and* $P_{ji} > 0$,

(ii) $x_i \approx_g x_j$ *if and only if* $P_{ij} = 0$ *and* $P_{ji} = 0$,

(iii) x_i *and* x_j *are not comparable* (*implying that* g *is not* 2*-monotonic*) *if and only if* $P_{ij} > 0$ *and* $P_{ji} > 0$.

The 2-monotonicity of g can be tested by constructing the matrix P and applying Lemma 5.2. If g is 2-monotonic, the order $\preceq_g$ of the variables can also be obtained from P.

Let us now consider the computation of the matrices $P(a)$ and P. Initially, we start with $P \equiv 0$ corresponding to $M = \emptyset$. Let us assume in the general step that P for the current M has already been computed, and a new vector c is added to M.

Step 1. Initialize the matrix $P(c)$ by setting

$$P_{ij}(c) := \begin{cases} 1 & \text{if } c_i = 1 \text{ and } c_j = 0, \\ 0 & \text{otherwise,} \end{cases} \tag{16}$$

and let $M := M \cup \{c\}$ and $P := P + P(c)$.

Step 2. Compare c with each $e(\neq c) \in M$ to see if condition (13) holds. If c and e can be regarded as a and d in (13), respectively, then let $P_{ij}(c) := 0$ for all i and j satisfying (11) and (12). Similarly, if c and e can be regarded as d and a in (13), respectively, then let $P_{ij}(e) := 0$ for all i and j satisfying (11) and (12).

Step 3. Update the matrix P to reflect all modifications in Step 2.

The time required in Step 1 is clearly $O(n^2)$. As noted in condition (13), there is a unique index j used in modifying matrices $P_{ij}(c)$ or $P_{ij}(e)$ in Step 2, and thus Step 2 requires $O(n)$ time for each vector $e \in M$. Hence the total time for Step 2 is $O(n|M|)$. Finally, Step 3 to update P as a result of the changes of $P_{ij}(c)$ and $P_{ij}(e)$ in Step 2 can be done in $O(n|M|)$ time. Therefore, the time required to increment M by a vector c is $O(n|M| + n^2)$.

Since the sets MT and MF can be incremented by at most $|\min T(f)|$ and $|\max F(f)|$ times, respectively, the total time required for this part is $O(n(|\min T(f)|^2 + |\max F(f)|^2) + n^2(|\min T(f)| + |\max F(f)|)) = O(nm^2 + n^2 m)$.

5.3. When g is not 2-monotonic. Assuming that the g tested in the previous subsection is not 2-monotonic, we show here that one can either conclude that f is not 2-monotonic or find an unknown vector (as discussed in (i) and (ii) of section 3).

If g is not 2-monotonic, there must exist indices $i \neq j$ for which $P_{ij} > 0$ and $P_{ji} > 0$. This implies by the definition of P that there exist vectors $a, b \in M$ for which

$$a_i = 1, \qquad a_j = 0, \qquad b_i = 0, \qquad b_j = 1.$$

Let us define the vectors a' and b' by complementing the ith and jth components of a and b, respectively. We shall consider the cases $g = g_1$ and $g = g_0$ separately.

If $g = g_1$ (and hence $M = MT$), then the vectors a' and b' do not belong to T, i.e., there is no $a'' \in MT$ or $b'' \in MT$ such that $a' \geq a''$ or $b' \geq b''$ since the existence of such an $a'' \in MT$ (respectively, $b'' \in MT$) would contradict the assumption $P_{ij}(a) > 0$ (respectively, $P_{ji}(b) > 0$) (in view of condition (13) with $d = a''$ or $d = b''$).

(i) If $f(a') = f(b') = 0$, then we can conclude that f is not 2-monotonic. Indeed, $x_i \not\preceq_f x_j$ follows from $f(a) = 1$ and $f(a') = 0$, and $x_j \not\preceq_f x_i$ follows from $f(b) = 1$ and $f(b') = 0$.

(ii) If at least one of $f(a')$ and $f(b')$ is 1, say $f(a') = 1$, then $a' \notin T \cup F$ (since $a' \notin T$, as seen above, and $a' \notin F$ because $f(a') = 1$), and hence a' is an unknown vector.

On the other hand, if $g = g_0$ (and hence $M = CMF$), we consider $\bar{a}$, $\bar{b}$, $\bar{a'}$, and $\bar{b'}$, where $\bar{a}$ denotes the complement of a, etc. By definition (6) of g_0, we have the following.

(i′) If $f(\bar{a'}) = f(\bar{b'}) = 1$, then f is not 2-monotonic.

(ii′) If at least one of $f(\bar{a'})$ and $f(\bar{b'})$ is 0, say $f(\bar{a'}) = 0$, then $\bar{a'}$ is an unknown vector.

5.4. When the variable orders of g_1 and g_0 do not coincide. Assume that g_1 and g_0 of (6) are both 2-monotonic but the orders $\preceq_{g_1}$ and $\preceq_{g_0}$ of the variables do not coincide. In this case, we shall identify an unknown vector.

For simplicity, let $x_i \succ_{g_1} x_j$ but $x_i \approx_{g_0} x_j$ or $x_i \prec_{g_0} x_j$. In order to avoid confusion, let us denote $P_{ij}(a)$ for g_1 (respectively, for g_0) by $P^1_{ij}(a)$ (respectively, by $P^0_{ij}(a)$). Then as discussed in section 5.2, there is a vector $a \in MT$ with

$$P^1_{ij}(a) > 0 \quad \text{and} \quad a_i = 1, \qquad a_j = 0, \tag{17}$$

and the vector a' obtained from a by complementing the components a_i and a_j does not belong to T (since there is no $d \in MT$ satisfying (13) for this a). This a' does not belong to F either, i.e., it is an unknown vector. Indeed, if $a' \in F$, then there is a vector $a'' \geq a'$ with $a'' \in MF$. It is easy to see that $a'_i = 0, a'_j = 1, a''_i = 0, a''_j = 1$, and $a''_k \geq a'_k$ for $k \neq i, j$. Therefore, $\bar{a''}_i = 1, \bar{a''}_j = 0, \bar{a}_i = 0, \bar{a}_j = 1$, and $\bar{a''}_k \leq \bar{a}_k$ for $k \neq i, j$. Then $\bar{a''} \in CMF$ (i.e., $g_0(\bar{a''}) = 1$) and $a \in MT$ (i.e., $g_0(\bar{a}) = 0$) together imply that there is no vector d for which the pair $\bar{a''}$ and d satifies (13). Thus $P^0_{ij}(\bar{a''}) > 0$, in contradiction to the assumption that $x_i \approx_{g_0} x_j$ or $x_i \prec_{g_0} x_j$.

Similarly, if $x_i \succ_{g_0} x_j$ but $x_i \approx_{g_1} x_j$ or $x_i \prec_{g_1} x_j$, then there is a vector $a \in CMF$ satisfying (17). Defining a' similarly as above, we see that $\bar{a'}$ is an unknown vector.

5.5. Checking if $g_1 = f$. Assume now that both g_1 and g_0 are 2-monotonic, and the orders $\preceq_{g_1}$ and $\preceq_{g_0}$ coincide. We show how to test condition (8) of section 3

and how to obtain an unknown vector if (8) does not hold (i.e., steps (iii) and (iv) of section 3). For simplicity of discussion, we assume in this subsection that

$$x_1 \succeq_{g_i} x_2 \succeq_{g_i} \cdots \succeq_{g_i} x_n \quad \text{for } i = 0, 1, \tag{18}$$

i.e., g_1 and g_0 are regular. The order (18) can be obtained in $O(n \log n)$ time by applying a sorting algorithm to the preorder $\preceq_{g_i}$. If the sets T and F are defined by (5), then the definition of g_1 and g_0 shows that

$$T \cap F = \emptyset. \tag{19}$$

From assumption (18) and the definition of the $\preceq_{g_i}$ order, it follows that the set T is *left-shift stable*, i.e.,

$$\begin{aligned} &a \in T \quad \text{implies} \quad a + e^i - e^j \in T \quad \text{for any } \ i < j \\ &\qquad \text{such that } \ a_i = 0 \quad \text{and} \quad a_j = 1, \end{aligned} \tag{20}$$

where e^i is the ith unit vector, and similarly, F is *right-shift stable*, i.e.,

$$\begin{aligned} &b \in F \quad \text{implies} \quad b - e^i + e^j \in F \quad \text{for any } \ i < j \\ &\qquad \text{such that } \ b_i = 1 \quad \text{and} \quad b_j = 0. \end{aligned} \tag{21}$$

Our test algorithm for condition (8) is based on the following lemma.

LEMMA 5.3. *Assume that both of the functions g_1 and g_0 defined from MT and MF are 2-monotonic and satisfy* (18). *If MT, MF, and T, F of* (5) *further satisfy the properties that*

(i) *$a - e^j \in F$ for all $a \in MT$ and for all j with $a_j = 1$, and*

(ii) *$b + e^j \in T$ for all $b \in MF$ and for all j with $b_j = 0$,*

then there is no unknown vector, i.e., $MT = \min T(f)$ and $MF = \max F(f)$ (hence $g_1 = f$).

Proof. Taking any vector $c \notin F$, we show that $c \in T$. (This proves that there is no unknown vector $c \notin T \cup F$.) Let us choose the maximum k such that the vector c^k defined by

$$c_j^k = \begin{cases} c_j, & j = 1, 2, \ldots, k, \\ 0, & j = k+1, \ldots, n \end{cases} \tag{22}$$

satisfies

$$c^k \in F.$$

Obviously, $k < n$ and

$$c_{k+1} = 1 \tag{23}$$

by the maximality of k. Let us then choose a vector $d \in MF$ such that $d \geq c^k$. By the maximality of k, this d satisfies $d_{k+1} = 0$. Moreover,

$$d_j = c_j, \quad j = 1, 2, \ldots, k, \tag{24}$$

since if $d_i = 1$ and $c_i = 0$ for some $i \leq k$, then property (21) shows that the vector d' obtained from d by complementing d_i and d_{k+1} belongs to F, implying that the vector c^{k+1} defined by (22) satisfies $c^{k+1} \leq d' \in F$, in contradiction with the maximality of k.

Now let the vector d'' be obtained from d by flipping $d_{k+1} = 0$ to 1. Because of assumption (ii), the vector d'' belongs to F; hence there is a $d^* \in MT$ such that

$$d^* \leq d''. \tag{25}$$

If there are several vectors in MT satisfying (25), we shall choose for d^* the "leftmost" one in the sense that

$$d^* + e^i - e^h \not\leq d'' \quad \text{for all } i < h \quad \text{such that } d^*_i = 0 \quad \text{and} \quad d^*_h = 1. \tag{26}$$

This d^* satisfies $d^*_{k+1} = 1$ since otherwise $d^* \leq d$ and hence $d^* \in F$, a contradiction.

We claim that

$$d^*_j = 0, \quad j = k+2, k+3, \ldots, n, \tag{27}$$

that is, $d^* \leq c$ by (23), (24), and (25). This implies $c \in T$, proving the lemma.

To prove (27), assume that $d^*_h = 1$ for some $h > k+1$. If there is an $i \leq k+1$ such that $c_i = 1$ and $d^*_i = 0$, we have

$$d^* + e^i - e^h \leq d'' \quad \text{(by (24) and the definition of } d''),$$

in contradiction with the selection rule (26) of d^*. Therefore,

$$d^*_j = c_j, \quad j = 1, 2, \ldots, k+1,$$

and hence $c^{k+1} \in F$ is implied by assumption (i) applied to $d^* \in MT$. However, this contradicts the maximality of k, and hence (27) holds. □

In order to check conditions (i) and (ii) of this lemma, we shall need the following characterization, which follows directly from the definition of regularity.

LEMMA 5.4. *Let us consider the sets MT and MF and functions g_1 and g_0 of Lemma* 5.3. *Then we have the following:*

(i) *Condition* (i) *of Lemma* 5.3 *holds if and only if, for every $a \in MT$, there exists a $b \in MF$ such that $b \geq a - e^j$, where j denotes the maximum index with $a_j = 1$.*

(ii) *Condition* (ii) *of Lemma* 5.3 *holds if and only if, for every $b \in MF$, there exists an $a \in MT$ such that $a \leq b + e^j$, where j is the maximum index with $b_j = 0$.*

Furthermore, as pointed out in [23], the existence of $a \in MT$ (respectively, $b \in MF$) satisfying $a \leq y$ (respectively, $b \geq y$) for a given vector y can be tested in $O(n)$ time if g_1 defined by MT (respectively, g_0 defined by MF) is regular. Such a procedure will be described below for the case of MT since MF can be handled analogously.

Let us store all the vectors of MT in a binary tree $B(MT)$ of height n, in which the left edge (respectively, right edge) from a node in depth $j-1$ represents the case $x_j = 0$ (respectively, $x_j = 1$). A leaf node v of $B(MT)$ in depth n stores the vector $a \in MT$, the components of which correspond to the edges of the path from the root to v. In order to have a compact representation, edges with no descendants are removed from $B(MT)$. An example for such a binary tree is shown in Fig. 1, corresponding to the set $MT = \{011, 101, 110\}$.

Given a vector y, the algorithm starts from the root v^0 of $B(MT)$ and follows the edges down to the leaf corresponding to a vector a satisfying $a \leq y$. At each node v^{j-1} of depth $j-1$ in $B(MT)$, an edge is selected by the following rule:

1. If $j \leq n$ and $y_j = 0$, then follow the left edge from v^{j-1} to the next node v^j. If there is no left edge to follow, then stop (there is no $a \in MT$ satisfying $a \leq y$).

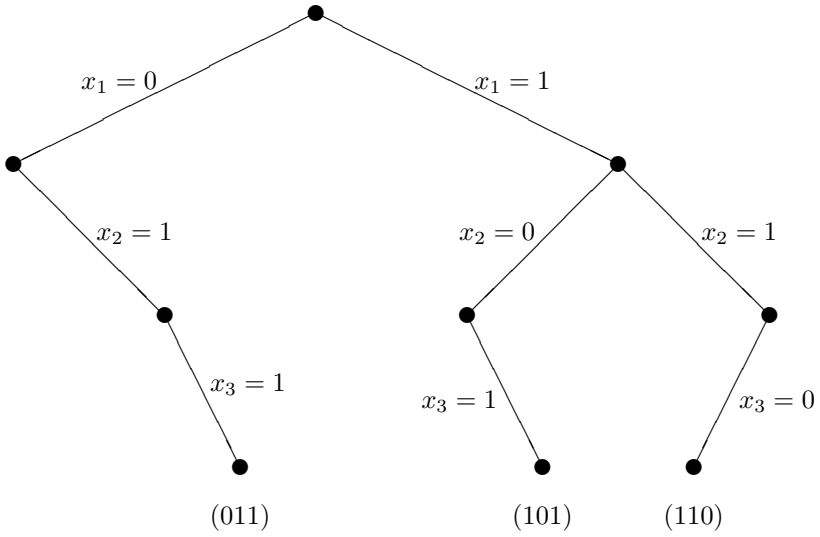


FIG. 1. *A data structure* $B(MT)$.

2. If $j \leq n$ and $y_j = 1$, then follow the right edge from v^{j-1}. If there is no right edge from v^{j-1}, then follow the left edge to the next node v^j.

3. If $j = n+1$, then stop (the vector a associated with the current leaf v^n satisfies $a \leq y$).

Based on Lemma 5.4 and on this algorithm, it is easy to see that condition (i) of Lemma 5.3 for a vector $a \in MT$ and condition (ii) for a vector $b \in MF$ can be checked in $O(n)$ time, respectively. In the algorithm of this section, the sets MT and MF are gradually augmented. Therefore, in each iteration, conditions (i) and (ii) have to be checked only for the newly added vectors $a \in MT$ and $b \in MF$. Therefore, the total time needed for this part is $O(n|MT| + n|MF|) = O(nm)$.

In this process, if condition (i) or (ii) of Lemma 5.3 fails to hold (i.e., either no $b \in MF$ satisfies $b \geq a - e^j$ or no $a \in MT$ satisfies $a \leq b + e^j$), then the vector $a - e^j$ or $b + e^j$ is an unknown vector for the current MT and MF. In other words, the computation in this subsection either concludes that $g_1 = f$ holds or provides a new unknown vector.

Remark. Although we did not need it in our algorithm, the condition $MT = \min T(f)$ can also be checked by utilizing a polynomial-time algorithm for dualizing a regular function [3, 5, 10, 21, 22]. Since

$$\max F(g_1) = \{\bar{a} \mid a \in \min T(g_1^d)\},$$

$\max F(g_1)$ can be computed by applying such a dualization algorithm to g_1. Then $MT = \min T(f)$ holds if and only if $\max F(g_1) = MF$. The time required for dualization is $O(n|MT| + n|MF|) = O(nm)$ [3, 22], and $\max F(g_1) = MF$ can be checked in $O(n|MF|)$ time if we sort both sets lexicographically in $O(n|MF|)$ time and then compare them.

5.6. Description of the algorithm. The algorithm described so far is summarized as follows.

1 PROGRAM IDENTIFY-1
2 **begin**
3 Initialize MT and MF as described in section 5.1.
{$f = 0$ or $f = 1$ may be concluded here. All vectors in $MT \cup MF$ are unscanned.}
4 **repeat**
5 **while** (g_1 or g_0 is not 2-monotonic, or the orders $\preceq_{g_1}$ and $\preceq_{g_0}$ do not coincide)
{The functions g_1 and g_0 are defined by $\min T(g_1) = MT$ and $\min T(g_0) = \{\bar{a} | a \in MF\}$, and the 2-monotonicity of g_i as well as the $\preceq_{g_i}$ orders are checked as in section 5.2.}
6 **do** {See sections 5.2, 5.3, and 5.4.}
8 **if** (f is concluded not to be 2-monotonic)
9 **then** {See (i) or (i′) of section 5.3.}
STOP.
10 **else**
Using an unknown vector a found as in (ii) or (ii′) of section 5.3 or in section 5.4, obtain a vector c satisfying (7) as described in section 4.
Set $MT := MT \cup \{c\}$ if $f(c) = 1$, and set $MF := MF \cup \{c\}$ if $f(c) = 0$.
{The added c is unscanned. Although not explicitly stated, the matrices $P(a)$ for $a \in M$ and P are updated as described in section 5.2.}
11 **endif**
12 **endwhile** {g_1 and g_0 are 2-monotonic, and satisfy (18).}
13 **repeat** {Check if $\min T(f) = MT$ and $\max F(f) = MF$ hold.}
14 **if** (there is an unscanned vector $a \in MT$ or $b \in MF$)
15 **then**
Test if conditions (i) and (ii) of Lemma 5.4 hold for the vector a as described in section 5.5 (vector a is now scanned).
16 **endif**
17 **until** (either new unknown vectors are found as explained in section 5.5,
or there are no more unscanned vectors in $MT \cup MF$)
18 **if** (unknown true vectors $a^1, a^2, \ldots, a^k$ and unknown false vectors $b^1, b^2, \ldots, b^h$ have been found in the above repeat-loop $\{k + h > 0\}$)
19 **then**
Construct minimal true vectors $c^1, c^2, \ldots, c^k$ from $a^1, a^2, \ldots, a^k$ as in section 4, and set $MT := MT \cup \{c^1, c^2, \ldots, c^k\}$. {These c^i's are unscanned.}
Construct maximal false vectors $d^1, d^2, \ldots, d^h$ from $b^1, b^2, \ldots, b^h$ as in section 4, and set $MF := MF \cup \{d^1, d^2, \ldots, d^h\}$. {These d^i's are unscanned.}
20 **endif**
21 **until** (no unknown vector remains)
22 STOP. {f is identified, i.e., $\min T(f) = MT$ and $\max F(f) = MF$.}
23 **endprogram**

Let us now analyze the time complexity. Each time the while-loop of lines 5–12 or the if-block of lines 18–20 is executed, MT or MF is augmented by new vectors. Thus these sets are updated at most $m = |\min T(f)| + |\max F(f)|$ times.

Let us note next that checking the 2-monotonicity of g_1 and g_0 and computing their orders $\preceq_{g_1}$ and $\preceq_{g_0}$ is done by maintaining the matrices $P(a)$ and P for MT and MF, as explained in section 5.2. As discussed there, the total time required for this computation is

$$O(nm^2 + n^2m).$$

Conditions (i) and (ii) of Lemma 5.4 are tested in the inner repeat-loop of lines 13–17. As explained in section 5.5, the total time needed for this step is

$$O(nm).$$

Finally, whenever an unknown vector is found, a new vector c in $MT \cup MF$ is computed by the algorithm of section 4. Since each execution requires $O(n)$ time, the total time needed for this is $O(nm)$. Summing these terms, we see that the time complexity of IDENTIFY-1 is

$$O(nm^2 + n^2m).$$

The number of queries, i.e., test vectors a for which $f(a)$ are evaluated, is $O(nm)$. This can be shown as follows. First, the values $f(a)$ are evaluated to find unknown vectors in the while-loop of lines 5–12 and in the repeat-loop of lines 13–17. From the discussion of sections 5.3, 5.4, and 5.5, it is easy to see that at most two vectors are evaluated to find one unknown vector. The number of unknown vectors obtained in the entire algorithm is m since one new vector in $MT \cup MF$ is generated from each unknown vector. In the process of obtaining a vector in $MT \cup MF$ from each unknown vector, $O(n)$ vectors are evaluated by the method of section 4. This argument proves the stated bound for the number of queries to the oracle.

THEOREM 5.5. *Given an unknown positive function f of n variables, algorithm* IDENTIFY-1 *decides whether or not f is 2-monotonic, and if f is 2-monotonic, it outputs* $\min T(f)$ *and* $\max F(f)$. *The time required is $O(nm^2+n^2m)$ and the number of queries to the oracle is $O(nm)$, where $m = |\min T(f)| + |\max F(f)|$.*

6. Another algorithm for identifying 2-monotonic functions. An algorithm that requires $O(n^3m)$ time and $O(n^3m)$ queries will be presented in this section.

6.1. Reducing time complexity of IDENTIFY-1. Since $m > n$ frequently holds, the computation of the matrices $P(a)$ and P for $M = MT$ and $M = CMF$ (see section 5.2), requiring $O(nm^2 + n^2m)$ time, represents the most time-consuming portions of IDENTIFY-1. We shall show here that in case $m > n^2$, the total complexity can be reduced at the cost of increasing the number of queries.

We first consider the computation of $P(a)$ and P. For simplicity, consider the case of $M = MT$ and $g = g_1$. Instead of conditions (9) and (10) of section 5.2, we look for the pairs of vectors a and b such that

$$\begin{gathered} a \in MT, \\ a_i = 1, \qquad a_j = 0, \\ b \text{ is obtained from } a \text{ by complementing } a_i \text{ and } a_j, \\ f(b) = 0 \end{gathered} \tag{28}$$

and compute the $n \times n$ matrices $P(a), a \in MT$, redefined here by

$$P_{ij}(a) = \begin{cases} 1 & \text{if } a \text{ and } b \text{ satisfy (28)}, \\ 0 & \text{otherwise.} \end{cases} \tag{29}$$

The matrix P is then defined by (15). Since here the condition $g(b) = 0$ in (9) is replaced by $f(b) = 0$, $P(a)$ is independent of other members of MT. Therefore, once $P(a)$ is computed at the time of generating $a \in MT$, it will not change later on. Also, it is not difficult to show that Lemma 5.2 of section 5.2 holds for the matrix P defined in this way.

The computation of the values $P_{ij}(a)$ for $a \in MT$ is carried out by generating vectors b obtained by complementing the components a_i and a_j of a for all pairs of indices $i \neq j$ for which $a_i = 1$ and $a_j = 0$ and then asking the oracle whether or not $f(b) = 0$ holds. Since there are $O(n^2)$ vectors b, the time and the number of queries required to construct $P(a)$ is $O(n^2)$ for each $a \in MT$.

If $f(b) = 1$ holds for the vector b generated as above, then there are two cases: either $b \in T$ (i.e., $g(b) = 1$) or $b \notin T$ (i.e., $g(b) = 0$). In the latter case, b is an unknown vector and can be treated as in (ii) of section 3. The task of checking if $b \in T$ requires $O(n|MT|)$ time for each b (e.g., by comparing b with every $c \in MT$ to check the inequality $b \geq c$). If we allow the use of additional queries, this task can be reduced to $O(n)$ time as follows. By applying the procedure of section 4 and using $O(n)$ queries, we can obtain a vector $c \in \min T(f)$ in $O(n)$ time from a vector b with $f(b) = 1$. Let us assume that all the vectors $a \in MT$ are stored in a binary tree $B(MT)$ of $O(nm)$ size, which was introduced after Lemma 5.4 in section 5.5. Then condition $c \in MT$ can be decided in $O(n)$ time by directly checking the existence of the path leading to the node that stores c. Obviously, $b \in T$ if and only if $c \in MT$. (Let us remark that the technique described after Lemma 5.4 in section 5.5 to check the condition $b \geq c$ directly on $B(TM)$ could not be used here since g may not be 2-monotonic.)

In conclusion, the total time and the number of queries required in this part are both $O(n^3)$.

The above argument can be easily modified to the case of $M = CMF$ and $g = g_0$. Condition (28) becomes

$$\begin{gathered} a \in MF, \\ a_i = 0, \qquad a_j = 1, \\ b \text{ is obtained from } a \text{ by complementing } a_i \text{ and } a_j, \\ f(b) = 1, \end{gathered} \tag{30}$$

and the rest of the procedure follows the above outline.

6.2. Description of the algorithm. The identification algorithm with the above modifications is very similar to IDENTIFY-1 given in section 5.6. The algorithm IDENTIFY-2 is obtained from IDENTIFY-1 by modifying only the while-loop of lines 5–12 in IDENTIFY-1.

while (g_1 or g_0 is not 2-monotonic, or orders $\preceq_{g_1}$ and $\preceq_{g_0}$ do not coincide)
do
  begin
    if (f is found not to be 2-monotonic) **then** STOP **else**
      begin
        Using an unknown vector a found as in (ii) or (ii′) of section 5.3 or in section 5.4, obtain a vector c satisfying (7) as in section 4. Set $MT := MT \cup \{c\}$ if $f(c) = 1$, and set $MF := MF \cup \{c\}$ if $f(c) = 0$.
          {The added c is unscanned.}
      end;
    while (there is an unscanned vector $a \in MT \cup MF$) **do**
      begin
        Compute $P(a)$ as described in section 6.1.
          {Vector a is now scanned.}

if (new vectors $c \notin MT \cup MF$ are generated while computing $P(a)$)
then {See section 6.1.}
If $f(c) = 1$, set $MT := MT \cup \{c\}$.
If $f(c) = 0$, set $MF := MF \cup \{c\}$.
{The added vector c is unscanned.}
end
end

The analysis of the time and the number of queries proceeds in a manner similar to that of section 5.6. The time required to construct each $P(a)$, $a \in MT \cup MF$, is $O(n^2)$ as discussed in section 6.1, and therefore the total time of these operations is

$$O(n^2|\min T(f)| + n^2|\max F(f)|) = O(n^2m).$$

The number of queries in this part is also $O(n^2m)$. The operations related to determining if the vector b is unknown or not and the generation of new vectors c require both $O(n^3)$ time and queries for every vector $a \in MT \cup MF$. Therefore, both the total time and the total number of queries of all these operations are

$$O(n^3|\min F(f)| + n^3|\max F(f)|) = O(n^3m).$$

The total time and the number of queries to check conditions (i) and (ii) of Lemma 5.4 is $O(nm)$ as described in section 5.5. The rest of the computation can be treated as in section 5.6 and will not increase the total complexity.

THEOREM 6.1. *Given an unknown positive function f of n variables, algorithm* IDENTIFY-2 *decides whether or not f is 2-monotonic, and if f is 2-monotonic, it outputs* $\min T(f)$ *and* $\max F(f)$. *The time required is $O(n^3m)$ and the number of queries to the oracle is $O(n^3m)$.*

7. Discussion. Two polynomial-time identification algorithms are presented in this paper for 2-monotonic positive functions. It would be important to reduce the time complexity and the number of queries further. It appears not unreasonable to conjecture that there is an algorithm with $O(n^2m)$ time and $O(n^2m)$ queries.

Another more ambitious goal is to develop a polynomial-time algorithm for identifying positive (not necessarily 2-monotonic) functions (or to disprove its existence). However, it is known for this case that Lemma 5.3 is no longer true [19], and hence different novel approaches are needed.

Acknowledgments. The discussion with Yves Crama of Universite de Liege, and Kazuhisa Makino of Kyoto University was very beneficial. The authors also appreciate the comments given by the anonymous reviewers, which helped improve the readability of this paper.

REFERENCES

[1] D. ANGLUIN, *Queries and concept learning*, Mach. Learning, 2 (1988), pp. 319–342.
[2] D. ANGLUIN, L. HELLERSTEIN, AND M. KARPINSKI, *Learning read-once formulas with queries*, J. Assoc. Comput. Mach., 40 (1993), pp. 185–210.
[3] P. BERTOLAZZI AND A. SASSANO, *An $O(mn)$ time algorithm for regular set-covering problems*, Theoret. Comput. Sci., 54 (1987), pp. 237–247.
[4] J. C. BIOCH AND T. IBARAKI, *Complexity of identification and dualization of positive Boolean functions*, Inform. and Comput., 123 (1995), pp. 50–63.

[5] E. Boros, *Dualization of aligned Boolean functions*, Research report 9-94, RUTCOR, Rutgers University, New Brunswick, NJ, 1994.
[6] E. Boros, P. L. Hammer, and J. N. Hooker, *Predicting cause-effect relationships from incomplete discrete observations*, SIAM J. Discrete Math., 7 (1994), pp. 531–543.
[7] E. Boros, P. L. Hammer, T. Ibaraki, and K. Kawakami, *Identifying 2-monotonic positive Boolean functions in polynomial time*, in ISA '91 Algorithms, W. L. Hsu and R. C. T. Lee, eds., Lecture Notes in Comput. Sci. 557, Springer-Verlag, Berlin, 1991, pp. 104–115.
[8] N. H. Bshouty, *Exact learning via the monotone theory*, in Proc. 34th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 302–311.
[9] N. H. Bshouty, T. Hancock, and L. Hellerstein, *Learning arithmetic read-once formulas*, in Proc. 24th ACM Symposium on Theory of Computing, ACM, New York, 1992, pp. 370–381.
[10] Y. Crama, *Dualization of regular Boolean functions*, Discrete Appl. Math., 16 (1987), pp. 79–85.
[11] Y. Crama, P. L. Hammer, and T. Ibaraki, *Cause–effect relationships and partially defined boolean functions*, Ann. Oper. Res., 16 (1988), pp. 299–326.
[12] T. Eiter and B. Gottlob, *Identifying the minimal transversals of a hypergraph and related problems*, Technical report CD-TR 91/16, Christial Doppler Labor für Expertensysteme, Technische Universität Wien, Vienna, 1991.
[13] M. Fredman and L. Khachiyan, *On the complexity of dualization of monotone disjunctive normal forms*, Technical report LCSR-TR-225, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1994.
[14] D. N. Gainanov, *On one criterion of the optimality of an algorithm for evaluating monotonic Boolean functions*, Comput. Math. Math. Phys., 24 (1984), pp. 176–181.
[15] A. V. Genkin and P. N. Dubner, *Aggregation algorithm for finding the informative features*, Automat. Remote Control, 49 (1988), pp. 81–86.
[16] J. Hansel, *On the number of monotonic Boolean functions of n variables*, Cybernet. Collect., 5 (1968), pp. 53–58.
[17] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *On generating all maximal independent sets*, Inform. Process. Lett., 27 (1988), pp. 119–123.
[18] E. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, *Generating all maximal independent sets: NP-hardness and polynomial-time algorithms*, SIAM J. Comput., 9 (1980), pp. 558–565.
[19] K. Makino and T. Ibaraki, *The maximum latency and identification of positive Boolean functions*, in ISAAC '94 Algorithms and Computation, D. Z. Du and X. S. Zhang, eds., Lecture Notes in Comput. Sci. 834, Springer-Verlag, Berlin, 1994, pp. 324–332.
[20] S. Muroga, *Threshold Logic and Its Applications*, John Wiley, New York, 1971.
[21] U. N. Peled and B. Simeone, *Polynomial-time algorithms for regular set-covering and threshold synthesis*, Discrete Appl. Math., 12 (1985), pp. 57–69.
[22] U. N. Peled and B. Simeone, *An $O(nm)$-time algorithm for computing the dual of a regular Boolean function*, Discrete Appl. Math., 49 (1994), pp. 309–323.
[23] J. S. Provan and M. O. Ball, *Efficient recognition of matroids and 2-monotonic systems*, in Applications of Discrete Mathematics, R. Ringeisen and F. Roberts, eds., SIAM, Philadelphia, 1988, pp. 122–134.
[24] N. A. Sokolov, *On the optimal evaluation of monotonic Boolean functions*, Comput. Math. Math. Phys., 22 (1979), pp. 207–220.
[25] L. G. Valiant, *A theory of learnable*, Comm. Assoc. Comput. Mach., 7 (1984), pp. 1134–1142.
[26] R. O. Winder, *Threshold Logic*, Ph.D. dissertation, Department of Mathematics, Princeton University, Princeton, NJ, 1962.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 110–137, February 1997

007

NAVIGATING IN UNFAMILIAR GEOMETRIC TERRAIN*

AVRIM BLUM†, PRABHAKAR RAGHAVAN‡, AND BARUCH SCHIEBER§

Abstract. Consider a robot that has to travel from a start location s to a target t in an environment with opaque obstacles that lie in its way. The robot always knows its current absolute position and that of the target. It does not, however, know the positions and extents of the obstacles in advance; rather, it finds out about obstacles as it encounters them. We compare the distance walked by the robot in going from s to t to the length of the shortest (obstacle-free) path between s and t in the scene. We describe and analyze robot strategies that minimize this ratio for different kinds of scenes. In particular, we consider the cases of rectangular obstacles aligned with the axes, rectangular obstacles in more general orientations, and wider classes of convex bodies both in two and three dimensions. For many of these situations, our algorithms are optimal up to constant factors. We study scenes with nonconvex obstacles, which are related to the study of maze traversal. We also show scenes where randomized algorithms are provably better than deterministic algorithms.

Key words. robot navigation, computational geometry, on-line algorithms

AMS subject classifications. 68Q25, 68T05, 52C05

PII. S0097539791194931

1. Motivation and results.

Practical work on robot motion planning falls into two categories: motion planning through a *known scene*, in which the robot has a complete map of the environment, and motion planning through an *unknown scene*, in which an autonomous robot must find its way through a new environment (see, for example, [9, 13, 15, 21, 24] and references therein). Virtually all previous *theoretical* work (see [32] and references therein) has focused on the former problem. Papadimitriou and Yannakakis [26] studied the latter problem, which is also the subject of this paper: the design and evaluation of strategies for navigation in an unknown environment. The unfamiliar environment may be either a warehouse or factory floor whose contents are frequently moved or a remote terrain such as Mars [30]. The design and evaluation of algorithms for such navigation is a natural algorithmic problem that deserves more theoretical study.

1.1. Model.

A *scene* $\mathcal{S}$ is a region (of $\mathbf{R}^2$ or $\mathbf{R}^3$) containing a start point s and a target t together with a set of opaque, impenetrable, nonoverlapping obstacles, none of which contains s or t. Most of this paper will consider two-dimensional scenes. The target t may be a point, a polygon/polyhedron, or an infinite wall. To avoid certain degeneracies, we assume that a unit diameter circle (unit cube in three dimensions) can be inscribed in each obstacle; this guarantees that the obstacles have a certain minimum "thickness."

A point robot has to travel from s to t, and it knows both its current absolute position and the position of t. In walking towards t, it must circumvent the obstacles

* Received by the editors February 11, 1991; accepted for publication (in revised form) April 20, 1995.

http://www.siam.org/journals/sicomp/26-1/19493.html

† School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (avrim@theory.cs.cmu.edu). Part of this research was performed while this author was visiting the IBM T. J. Watson Research Center, and part was performed while this author was at the Massachusetts Institute of Technology and supported by an NSF graduate fellowship.

‡ IBM Research Division, Almaden Research Center, San Jose, CA 95120 (pragh@almaden.ibm.com).

§ IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598 (sbar@watson.ibm.com).

in $\mathcal{S}$. The robot does not know the positions and extents of these obstacles in advance; rather, it finds out about obstacles as it encounters them. Where two obstacles touch, we assume that the robot can "squeeze" between them. Thus a scene that consists only of convex obstacles cannot have a nonconvex obstacle composed of abutting convex obstacles.

The most natural mechanism for the robot to learn about a scene is vision: the robot discovers obstacles as they come into its view and uses this information to decide how to proceed towards t. For simplicity of exposition, we describe our algorithms assuming that when the robot first sees an obstacle, it is given the shape, size, and position of the obstacle (even though much of that obstacle may be invisible from where it stands). However, we show how many of our algorithms can be made to work with essentially the same upper bounds (up to a constant factor) under a considerably weaker assumption—a *tactile robot* that learns about obstacles only by bumping into them and moving along them. For this we use variants on the "doubling" strategies of Baeza-Yates et al. [1].

Let $R(\mathcal{S})$ be the total distance walked by a robot R in going from s to t in scene $\mathcal{S}$, and let $d(\mathcal{S})$ denote the length of the shortest (obstacle-free) path in the scene between s and t. (Because of the obstacles, this may be substantially larger than the Euclidean distance between s and t.) Let $\mathcal{S}(n)$ denote the set of scenes in which the Euclidean distance between s and t is n. Following the lead of [26], we use as the figure of merit for the robot the *ratio*

$$\rho(R, n) = \sup_{\mathcal{S} \in \mathcal{S}(n)} \frac{R(\mathcal{S})}{d(\mathcal{S})}$$

and study its growth as a function of n.

For convenience, we put the scene in Cartesian coordinates, using "north"/"south" to denote the direction of increasing/decreasing y value, "east"/"west" for the direction of increasing/decreasing x value, and "up"/"down" for the direction of increasing/decreasing z value, respectively. In two dimensions, we also use "vertical" to mean parallel to the y-axis and "horizontal" to mean parallel to the x-axis. The start point s is always assumed to be at the origin, and unless otherwise specified, we will assume that the current scene belongs to $\mathcal{S}(n)$. Finally, we use $\log n$ to denote $\log_2 n$.

1.2. Summary of results. For most of this paper, we consider two-dimensional scenes where t is a point and the obstacles are rectangles with sides parallel to the axes (rather than squares as in [26]). Surprisingly, even this problem turns out to be quite complicated. We solve this problem by breaking it into the following two subproblems:

The wall problem: scenes in which t is an *infinite vertical line* and the obstacles are oriented rectangles. The goal is to reach a point on t of the robot's choosing.

The room problem: scenes in which the obstacles are oriented rectangles that are confined to lie within a square "room." Here s is a point on a wall of the room and t is the point at the center of the room. The robot can "squeeze" between any two obstacles or between the walls and any obstacle. This intriguing special case is of interest in its own right as a model for navigation in a bounded region such as a warehouse.

Section 2 describes an optimal algorithm for the wall problem. The algorithm achieves an upper bound of $O(\sqrt{n})$ on the ratio $\rho(R, n)$, matching the lower bound of [26]. To devise this algorithm, we develop a general "sweep" paradigm that is fairly natural: a human lost in a strange city would probably do a similar search.

Section 3 considers the room problem. The algorithm for this problem achieves a ratio $\rho(R, n)$ that is $O(2^{\sqrt{3 \log n}})$. Following and building upon our result, Bar-Eli et al. [2] have established a tight bound of $\Theta(\log n)$ on the ratio of deterministic algorithms for the room-problem. The approach taken by the room-problem algorithm is different from the one taken for the wall problem. Here we develop a "caliper" method that pins the target down to lie within a sequence of advancing paths. Intriguingly, in the room problem, the shortest path from s to t has length $O(n)$. To see this, suppose that s is the southwest corner of the room. Therefore, the *greedy* path from the target t that travels due south if possible and otherwise due west will reach s and have as its length the L_1 distance between s and t. (If s is not in the corner, then by traveling along the room boundaries, one can reach s at an additional constant factor cost.) In contrast, greedy paths from s are not guaranteed to go anywhere near t. Thus getting out of a room is easy, but getting in towards a small target seems to be hard.

Section 4 shows how to combine our solutions for the wall and room problems to obtain a tight bound of $\Theta(\sqrt{n})$ for point-to-point navigation in scenes consisting of oriented rectangular obstacles.

Section 5 describes how our algorithms work (with at worst a constant factor degradation in ratio) when the robot is tactile: it learns about obstacles by "feeling" them. In this case, our algorithms bump into obstacles and slide along their edges in a manner reminiscent of *compliant* motion planning [7] in the context of navigation with a map.

Section 6.1 considers the room problem with arbitrary rectangular obstacles. We show that $d(\mathcal{S})$ can now be $\Omega(n^{3/2})$. Unlike the case of oriented rectangles, the greedy path is no longer guaranteed to find an inexpensive way out of the room. For these scenes, we give lower and upper bounds on $\rho(R, n)$.

Section 6.2 extends our algorithms for the room problem to the case of more general convex polygonal obstacles.

Section 7 gives extensions of our algorithm for the wall problem to three dimensions and also for point-to-point navigation in three dimensions. Both of these algorithms work provided the obstacles are oriented rectangular cuboids, achieving optimal ratios. (A cuboid is a rectangular parallelepiped.)

In section 8, we give a randomized algorithm for certain cases of the wall problem. We show that the (expected) ratio of our algorithm is $2^{O(\sqrt{\log n \log \log n})}$, which is much smaller than the corresponding deterministic lower bound. This demonstrates the power of randomization in navigation.

Section 9 deals with nonconvex obstacles (and therefore mazes). We give a lower bound for randomized algorithms and show that a deterministic algorithm of Rao et al. [28] meets this bound. The algorithm is memory intensive, and so we offer an alternative algorithm that is very simple, memoryless, and randomized and that achieves the same upper bound in the plane.

We conclude with a list of some open problems in section 10.

1.3. Related theoretical work. The ratio $\rho(R, n)$ is studied by Papadimitriou and Yannakakis [26] and independently by Eades, Lin, and Wormald [14]. Papadimitriou and Yannakakis proved that when s and t are points in the plane and all obstacles are squares, $\rho(R, n)$ is at least 1.5, and they complement this with an algorithm that attains $\rho(R, n) \leq 1.5 + o(1)$ for all n. It is also shown in [14, 26] that when t is an infinite wall at distance n from s and the obstacles are oriented rectangles, then $\rho(R, n)$ is $\Omega(\sqrt{n})$. Coffman and Gilbert [12] study the performance of simple heuris-

tics in the presence of randomly placed obstacles. Kalyanasundaram and Pruhs [16] and Mei and Igarashi [22] consider scenes in which all obstacles have bounded aspect ratios. Klein [18] has given a small constant upper bound on the ratio for scenes that are *streets*, a class of simple polygons. Earlier, Lumelsky and Stepanov [21] gave a simple navigation algorithm that guarantees $R(\mathcal{S})$ to be bounded by $d(\mathcal{S})$ plus the sum of the perimeters of all obstacles with no restrictions on the aspect ratios or the convexity of the obstacles. Their algorithm does not minimize the ratio ρ. Several papers (see [25, 28, 29] and references therein) give algorithms for building up a map of a scene by exploring it entirely. Maze traversal has received considerable attention in the past in various papers [5, 19, 27], none of which considers the ratio metric. The reader is referred to [20] for a comprehensive survey of the results in these papers.

The ratio measure $\rho(R, n)$ has close connections to the *competitiveness* measure used in the study of on-line algorithms [6, 23, 31]; indeed, our problem resembles an on-line setting in which the obstacles encountered by the robot form a sequence of "requests," and we compare its total cost $R(\mathcal{S})$ to the "off-line cost" $d(\mathcal{S})$. It is therefore worth pointing out some key differences between the models: (a) In the navigation problem, the robot has a definite target towards which it moves, while there is no such notion in on-line paging [31], for example. (b) The robot can move back and forth through the scene, revisiting previously seen obstacles, thus having some control on the requests it encounters in the future. (c) Competitive analysis deals with request sequences of arbitrary (possibly infinite) length, whereas here we have a fixed number of obstacles in the scene. Thus we cannot cast our navigation problem in a standard on-line framework such as the server problem [23] or metrical task systems [6]. Nevertheless, the analogy with on-line algorithms proves useful in the study of randomized navigation (section 8).

2. The wall problem. In this section, we consider scenes in which t is an infinite vertical wall at distance n to the east of s and the obstacles are rectangles whose sides are parallel to the axes. At the end of the section, we show how to modify our algorithm to work also when t is not vertical. We call the *width* of an obstacle its length in the x-direction and the *height* of an obstacle its length in the y-direction. To make the presentation clearer, we assume below that $\sqrt{n}$ is an integer. However, our algorithm and analysis can trivially be adapted to the general case.

We present an algorithm that achieves ratio $\rho(R, n) = O(\sqrt{n})$. This matches the lower bound proven in [26], so our algorithm is optimal up to constant factors.

The algorithm maintains four variables: the *window size* W, a *threshold* τ, a *sweep direction*, and a *sweep counter*. Initially, W is set to n, the sweep direction is south, and the sweep counter is set to zero. The threshold τ is always set to $W/\sqrt{n}$.

We begin with a high-level view of the algorithm and its analysis. The algorithm maintains a window of varying size around the x-axis. The robot makes $\sqrt{n}$ sweeps in directions alternating between north and south for each window size. Upon completion of these $\sqrt{n}$ sweeps, the window size is doubled. Given a window of size W (which ranges from $y = +W/2$ to $y = -W/2$), the distance walked by the robot in sweeping is $O(W\sqrt{n})$. We show that the shortest path that cuts through all of the $\sqrt{n}$ sweeps has length $\Omega(\sqrt{n}\tau) = \Omega(W)$. Let W_f be the window size at the time the robot reaches t. We prove that the total distance walked by the robot is $O(W_f\sqrt{n})$, while $d(\mathcal{S}) = \Omega(W_f)$.

We now describe the algorithm. Starting from point s, the robot travels due east until it either reaches t or hits an obstacle, say at (x, y). Below, we assume that the current sweep direction is south; the other case is symmetric. The next steps are

determined by the following rules:

Rule 1. If the distance to the nearest corner is less than τ, then the robot just goes "around" the obstacle. Specifically, it travels either south or north to the nearest corner, then east along the width of the obstacle to the opposite corner, and finally back along the height of the obstacle to the point $(x + w, y)$, where w is the width of the obstacle. (See Fig. 1(a).) From this point, it continues to travel due east until it hits the next obstacle (or reaches t and stops).

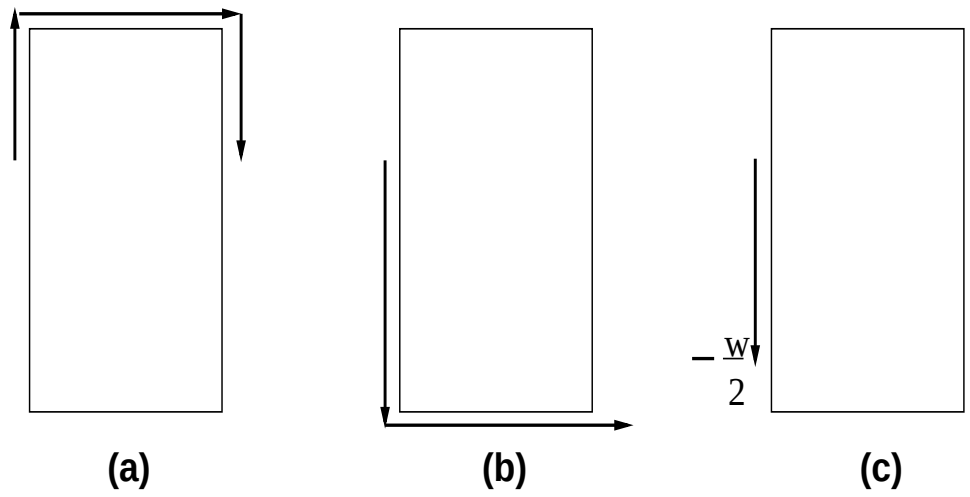


FIG. 1. *Going around an obstacle in the sweep algorithm.*

Rule 2. If the obstacle extends past both sides of the window (i.e., its north edge has y-coordinate greater than $W/2$ and its south edge has y-coordinate less than $-W/2$), then the robot doubles the window width W and threshold τ and resets the sweep counter to 0 and the sweep direction to south. Note that the ratio W/τ remains $\sqrt{n}$.

Rule 3. Otherwise (i.e., the distance to the nearest corner is more than τ, and the obstacle does not extend past both sides of the window), the robot travels south along the obstacle until it either hits the obstacle's southwest corner or reaches the window boundary (y-coordinate $-W/2$). In the first case, the robot just continues due east to the next obstacle (or t). (See Fig. 1(b).) In the second case, the robot increments the sweep counter by 1 and flips the sweep direction. If the counter is greater than $\sqrt{n}$, the robot resets the counter to zero and doubles the window size and the threshold. (See Fig. 1(c).)

Let W_f be the window size at the time the robot arrives at t.

THEOREM 2.1. *The total distance walked by the robot is* $O(W_f\sqrt{n})$.

Proof. To prove that the distance is bounded by $O(W_f\sqrt{n})$, we divide the path taken by the robot into three components: (1) horizontal segments, (2) segments walked south and north "along" obstacles using Rule 1, and (3) segments walked south and north using Rule 3.

Notice that (i) the total distance walked east is $n \leq W_f\sqrt{n}$ since $W_f \geq n$, and (ii) since the width of each obstacle is at least one unit, the total distance walked south and north using Rule 1 is bounded by $2n\tau_f \leq 2W_f\sqrt{n}$, where τ_f is the final threshold. It suffices to bound the third component as well. Fix a window size W. The distance walked by the robot using Rule 3 to complete one sweep is $O(W)$. Since

the robot makes at most $\sqrt{n}+1$ sweeps for each window size, the total distance for a fixed window size is $O(W\sqrt{n})$. The window size is doubled each time it is changed, and thus the total distance traveled over all window sizes is also $O(W_f\sqrt{n})$. □

THEOREM 2.2. *The length of the shortest path from s to t, $d(\mathcal{S})$, is $\Omega(W_f)$.*

Proof. Since we are just interested in the length of the shortest path up to a constant factor, we may assume that the path consists only of horizontal and vertical segments. The length of the horizontal segments is clearly at least n. If $W_f = n$, then we are done. Also, if W_f is determined by Rule 2 (there is an obstacle that extended past both sides of the previous window), we are done as well. Therefore, assume that neither of these is the case, which means that the robot has completed at least $\sqrt{n}$ full sweeps for some window size $W \geq \frac{1}{2}W_f$. We now show that the vertical component of the shortest path has length $\Omega(W) = \Omega(W_f)$.

Consider a point on the shortest path with y-coordinate of maximum absolute value. If this absolute value is at least $W/2 - \tau$, then clearly the shortest path has length at least $W/2 - \tau = \Omega(W)$. Suppose that this is not the case. Given a shortest path, for each of the $\sqrt{n}$ sweeps, define its first entry point to be the first point on the shortest path whose x-coordinate is the same as the x-coordinate of the starting point of the sweep. Similarly, define its first exit point to be the first point on the shortest path whose x-coordinate is the same as that of the end point of the sweep. Note that since sweeps do not overlap in their x-coordinates, the exit point of sweep i appears before the entry point of sweep $i+1$. Thus to lower bound the length of the shortest path, we can add together for each sweep i the vertical components of the shortest path from the ith entry point to the ith exit point. Consider the obstacles touched by the robot during some sweep—let's say it is a "south" sweep—that require the use of Rule 3. Each such obstacle extends at least τ to the north of the southernmost point of the previous such obstacle. Therefore, to travel from the entry point of the sweep (which is not between any two such obstacles) to the exit point of the sweep (also not between any two such obstacles) requires traveling a vertical distance of at least τ. Since there are $\sqrt{n}$ sweeps, the total vertical component is at least $\tau\sqrt{n} = W$. □

COROLLARY 2.3. *Our sweep algorithm achieves a ratio of $O(\sqrt{n})$ for the wall problem provided every obstacle is an oriented rectangle.*

A simple transformation of our algorithm allows it to achieve the same bounds even if the wall t is not vertical.

THEOREM 2.4. *The modified sweep algorithm below achieves a ratio of $O(\sqrt{n})$ for the wall problem with oriented rectangular obstacles, even if the wall is not vertical.*

Proof. The algorithm depends on the angle θ that t makes with the y-axis. Assume that $0 \leq \theta < \pi/4$ and that the wall runs from southwest to northeast. (The other three possibilities are analogous.) Here n is the shortest Euclidean distance between s and t. We distinguish between two cases:

Case 1: $\sin\theta \geq 1/\sqrt{n}$. In this case, the robot walks to t using the *greedy* east–south path from s (a path that travels due east if possible and otherwise due south). Observe that the length of this greedy path is the L_1 distance between s and the point of t that the path hits. The x-component of this L_1 distance is no more than $\sqrt{2}n$. Since $\sin\theta \geq 1/\sqrt{n}$, the y-component is bounded by $O(n^{1.5})$, implying that $\rho(R,n) = O(\sqrt{n})$.

Case 2: $\sin\theta < 1/\sqrt{n}$. We run the sweep algorithm exactly as described above until the first time the robot reaches a point (x_0, y_0) such that the point $(x_0, -W/2)$ is on or below t, where W is the width of the current window. Then the robot walks to t using the greedy east–south path. By Theorem 2.1, the distance walked by the

robot until reaching (x_0, y_0) is $O(W\sqrt{n})$, and it is clear that the length of the greedy path from there to t is $O(n + W)$. By an argument identical, to the one in the proof of Theorem 2.2, the shortest path from s to the vertical line $x = x_0$ has length $\Omega(W)$. By the requirement on θ, the shortest path from s to t has length $\Omega(W)$ as well. □

3. The room problem. In this section, we consider the *room problem*: scenes in which the obstacles are oriented rectangles confined to lie within a square room such that no obstacle touches the room walls; the point s is on the border of the room and t is in the center. (See Fig. 2.)

Later, we extend our results to rectangular rooms. Since travel along the room walls is "cheap," we may assume that s is in the southwest corner of the room, and for convenience we let t have coordinates (n, n), so the distance from s to t is in fact $n\sqrt{2}$.

Define a *greedy* $\langle +x, +y\rangle$ path to be a path that travels due east if possible and otherwise due north. Similarly, define greedy $\langle +y, +x\rangle$ paths, $\langle +x, -y\rangle$ paths, and so forth, to be ones that travel in the first direction if possible and otherwise the second direction. A *brute-force* $\langle +x\rangle$ path is one that travels due east, going around obstacles in its way along the shorter direction, but otherwise maintaining a constant y-coordinate. A *monotone* path from (x_1, y_1) to (x_2, y_2) is a path that does not both increase and decrease in any coordinate. For example, if $x_2 > x_1$ and $y_2 < y_1$, then the x-coordinate will never decrease and the y-coordinate will never increase. Notice that a greedy path is always monotone.

We now describe an algorithm that achieves $R(\mathcal{S}) = O(n^{3/2})$ and thus $\rho(R, n) = O(\sqrt{n})$. An improvement that uses this algorithm recursively achieves $\rho(R, n) = O(2^{\sqrt{3 \log n}})$.

The algorithm maintains the following invariant at the start of each iteration: the robot knows of a monotone obstacle-free path from a point (x_0, n) to a point (n, y_0), where $0 \le x_0, y_0 \le n$. Furthermore, the robot is positioned on a point of this monotone path. We begin with $x_0 = y_0 = 0$, where the known path is just a path along the room borders. Each loop through the algorithm will increase either x_0 or y_0 by at least an amount $\sqrt{n}$, walking a distance of only $O(n)$. (If the value increased (x_0 or y_0) is within $\sqrt{n}$ of n, then it is increased only up to n.) Since each of x_0 and y_0 can be increased by this amount only $\lceil\sqrt{n}\rceil$ times, the total distance walked by the robot to reach t is $O(n^{3/2})$.

For this first version of the algorithm, let $m = \sqrt{n}$. We will describe the algorithm as if t were allowed to be inside an obstacle, in which case the goal is simply to reach the obstacle containing t; this will allow for easier recursive application.

Algorithm Oriented-Room-Find (See Fig. 3.)

Initialization. Set x_0 and y_0 to 0. Set the monotone path to be the path along the room boundary from (x_0, n) to (n, y_0).

Step 1. Define $\tilde{t}$ to be the point with x-coordinate $\min\{x_0+m, n\}$ and y-coordinate $\min\{y_0 + m, n\}$. That is, unless we are close to t along some dimension, we have $\tilde{t} = (x_0 + m, y_0 + m)$. The goal of this step is to travel to some point t' not inside an obstacle that is to the northeast of $\tilde{t}$ and southwest of t inclusive. If no such point exists, we wish to travel to some point on the obstacle containing both $\tilde{t}$ and t.

For this (nonrecursive) version of the algorithm, we may reach t' as follows. First, traverse the monotone path to a point with y-coordinate equal to that of $\tilde{t}$. Then, if this is to the west of $\tilde{t}$, travel in a brute-force $\langle +x\rangle$ path until $\tilde{t}$ is reached or an obstacle containing $\tilde{t}$ is first encountered. In the latter case, unless the obstacle contains both

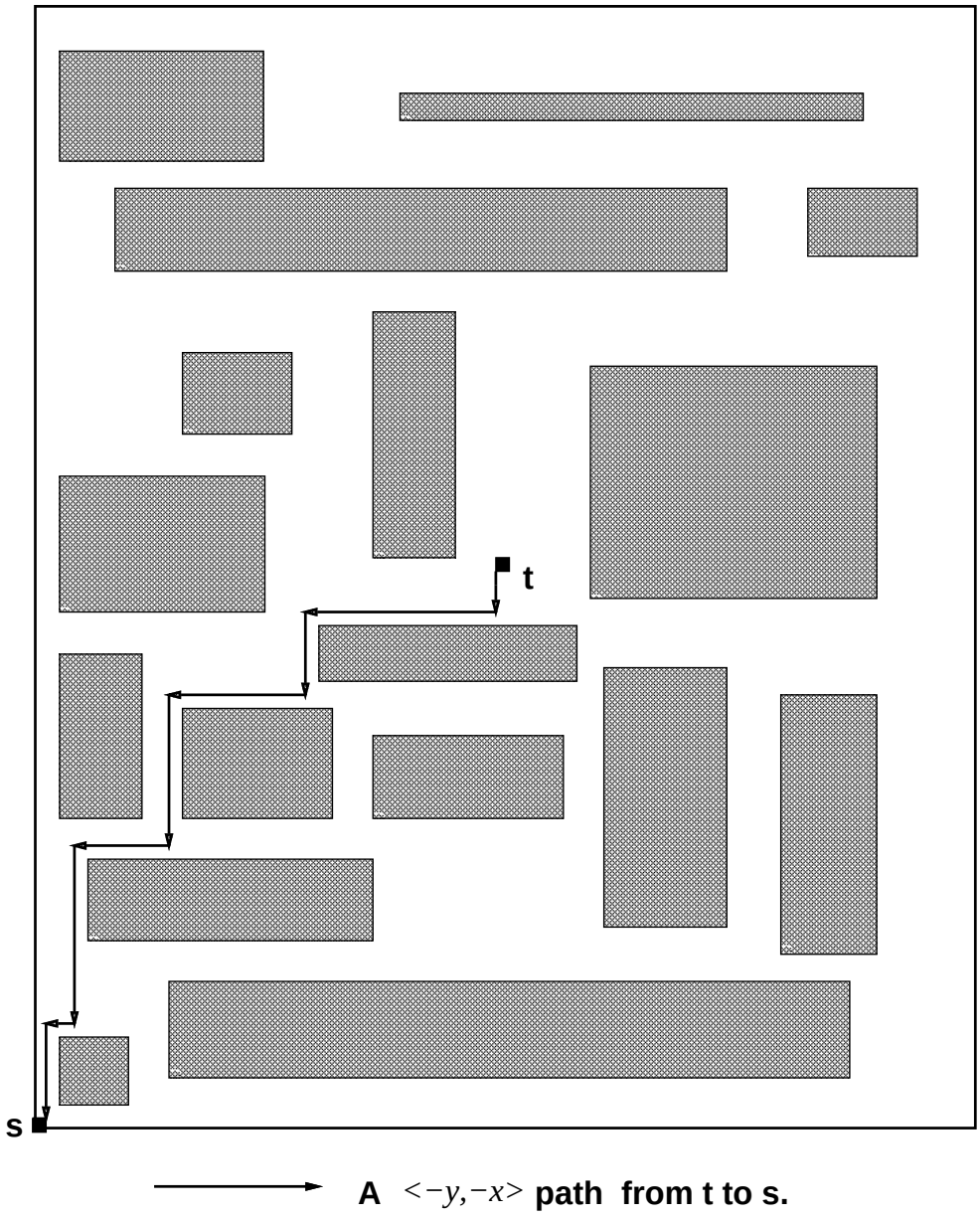


FIG. 2. *The room problem.*

$\tilde{t}$ and t (and we are done), we can just follow the obstacle boundary to a point in the desired region.

Step 2. Make a greedy $\langle +x, +y \rangle$ path from t' until either the x- or y-coordinate equals n. If we are at t, then halt. Otherwise, without loss of generality, assume that the robot has traveled to the west of t, so the current coordinates are $(\hat{x}, n)$ for $\hat{x} < n$. Notice that $\hat{x} \geq x_0 + m$ since the path was greedy.

Step 3. Let $x_0 = \hat{x}$. Travel a greedy $\langle +x, -y \rangle$ path from $(\hat{x}, n)$ until either a point

$(n, \hat{y})$ is reached (in which case let $y_0 = \hat{y}$) or the previous monotone path is hit. In either case, we have a new monotone path from (x_0, n) to (n, y_0) with x_0 increased by at least m. If the greedy $\langle +x, -y \rangle$ path was such that it reached (n, y_0) and in doing so followed the border of a single obstacle, then we must be at an obstacle containing t and so we are done. (Similarly, the robot would travel a greedy $\langle +y, -x \rangle$ path if it had hit to the south of target t in Step 2.)

Now go back to Step 1.

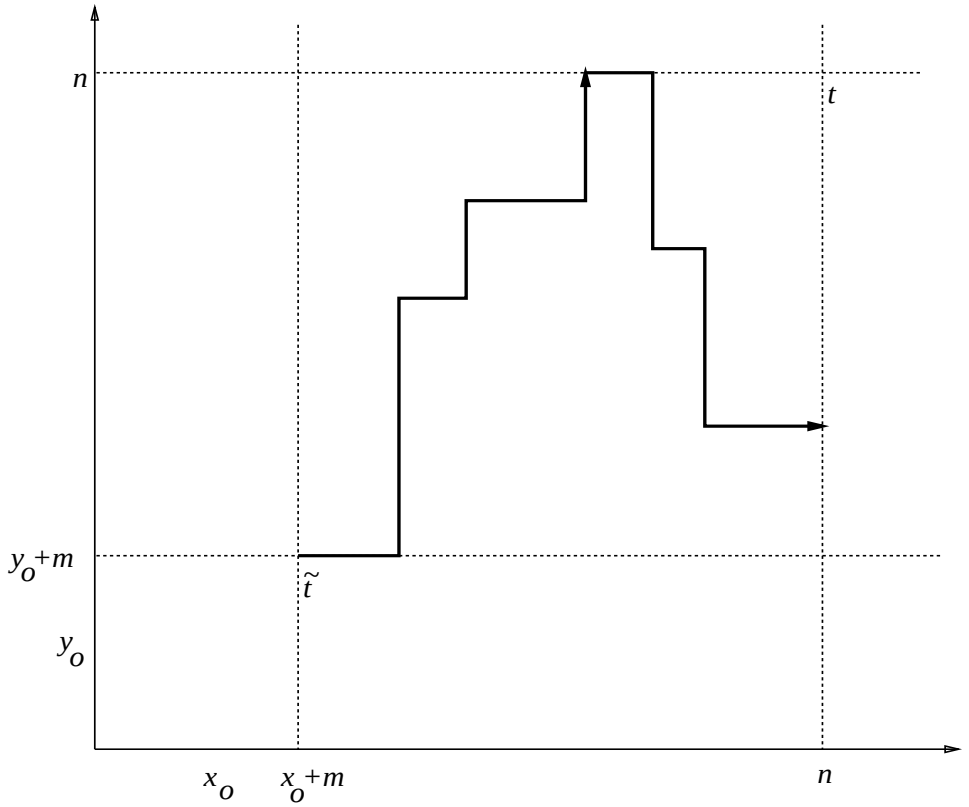


FIG. 3. *The algorithm for the room problem. Here* $\tilde{t} = t'$.

The distance walked performing Step 1 is $O(n)$ for traversing the monotone path and final obstacle and $O(m^2) = O(n)$ for the brute-force path traveled since the lower end of any obstacle encountered cannot extend below the monotone path (so there is at most $2m$ cost for every unit of progress made). Steps 2 and 3 together require traveling at most a distance $3n$ since each consists of a single greedy path, and in the two paths together, at least one coordinate is nondecreasing. Since these steps are iterated at most $2\lceil n/m \rceil$ times, the total distance traveled is at most $O(n^{3/2})$.

We can reduce the distance traveled to $O(n \cdot 2^{\sqrt{3 \log n}})$ by using algorithm Oriented-Room-Find recursively to reach $\tilde{t}$ (or an obstacle containing $\tilde{t}$) in Step 1 and optimizing the value of m. This works because to begin the algorithm needs only a monotone path of the form mentioned in the invariant, and not a true "room." In particular, there need not be any boundary "above" the obstacles at all. Define $T(n)$ to be the total distance traveled using this strategy to reach point $t = (n, n)$ (or a point on

the boundary of the obstacle containing t if t is inside some obstacle), given that the robot is on a known monotone obstacle-free path from $(0, n)$ to $(n, 0)$. As a base case, say if $n \leq 8$, we just use a brute-force path to reach t. Thus the distance traveled at each iteration of Step 1 is at most $2n$ for traversing the monotone path, $T(m)$ for the recursive call, and $3n$ for following the boundary of the final obstacle encountered.[1] Since the number of iterations is at most $2\lceil n/m \rceil$, we can bound the total cost $T(n)$ by

$$T(n) \;\leq\; 2\lceil n/m \rceil \, [T(m) + 8n] \quad \text{for } n > 8.$$

By substituting $m = n/(2^{\sqrt{3 \log n}})$ and using the inequality $\sqrt{x - k\sqrt{x}} \leq \sqrt{x} - k/2$ for $k > 0$, we get $T(n) \leq cn \cdot 2^{\sqrt{3 \log n}}$ for $c = 16(2 + \sqrt{2})$. We therefore have the following theorem.

THEOREM 3.1. *The algorithm for the room problem achieves* $\rho(R, n) = O(2^{\sqrt{3 \log n}})$.

If we consider a version of the room problem in which s and t are arbitrary points in the room, then the following strategy can be used to walk from s to t at a total cost that is $O(n2^{\sqrt{3 \log n}})$: simply walk from s out to a corner of the room, then use the above algorithm. Note, however, that in this case the length of the shortest path between s and t may be $o(n)$.

A generalization that will be used for the general point-to-point problem is when the room is rectangular with dimensions $2N \times 2n$, for $N \geq n$, and $t = (N, n)$. We use the same algorithm as for the square room, with one difference: we define point $\tilde{t} = (\min\{x_0 + mr, N\}, \min\{y_0 + m, n\})$ for $r = N/n$. (Again, if $n \leq 8$, we can just use a brute-force strategy to reach t traveling distance $O(N)$.) The value of m is optimized as follows. Define $T(n, r)$ to be the total distance traveled to reach point $t = (nr, n)$, given that the robot is on a known monotone obstacle-free path from $(0, n)$ to $(nr, 0)$. For a fixed value of m, the distance traveled at each iteration of Step 1 is at most $T(m, r) + 5nr$, while the distance traveled at each iteration of Steps 2 and 3 is at most $n + 2N \leq 3nr$. The number of iterations is at most $2\lceil n/m \rceil$, so we have

$$T(n, r) \leq 2\lceil n/m \rceil \, [T(m, r) + 8nr] \quad \text{for } n \geq 8.$$

The substitution used above $[m = n/(2^{\sqrt{3 \log n}})]$ results in $T(n, r) = O(rn \cdot 2^{\sqrt{3 \log n}})$.

Because we only needed a monotone path to start with, and not an entire room, we in fact have the following theorem.

THEOREM 3.2. *Given a monotone obstacle-free path between* $(0, n)$ *and* $(N, 0)$*, for* $N \geq n$*, the above algorithm will reach point* (N, n) *starting from that path with total cost* $O(N \cdot 2^{\sqrt{3 \log n}})$.

4. Point-to-point navigation. We combine the algorithms for the wall and room problems to obtain an algorithm for navigation in scenes where t is a point at (n, n) and the obstacles are oriented rectangles with no upper bounds on their extents.

The robot starts by taking a greedy $\langle +x, +y \rangle$ path from the start point s until it reaches a point s' with either x- or y-coordinate equal to that of t. Suppose that s' and t have the same y-coordinate. The robot now uses the sweep algorithm for the wall problem to travel to a point (n, y_0) with the same x-coordinate as t. Without

[1] The cost of traveling along the final obstacle can actually be amortized away at the expense of additional sentences of analysis.

loss of generality, assume $y_0 \geq n$. Notice that the path from s' to (n, y_0) taken by the robot in the sweep algorithm never decreases in the x-direction and that $y_0 - n$ is at most the final window width W_f. This path guarantees us that the *greedy* $\langle -y, -x \rangle$ path P from (n, y_0) will reach a point (x_0, n) with $x_0 \geq 0$. (In fact, x_0 is at least the x-coordinate of s'.) Now we invoke the algorithm for the room problem (Theorem 3.2) to arrive at t using the monotone path P as the room walls.

We analyze the distance walked by the robot. The distance traveled using the algorithm for the wall problem is at most $O(W_f\sqrt{n})$, where W_f is the size of the last window considered. The size of the (rectangular) room then created is at most $n \times W_f$. Therefore, using the algorithm of Theorem 3.2, the distance walked to reach t is $O(W_f\sqrt{n})$. By Theorem 2.2, the length of the shortest path from s' to t is at least cW_f for some constant $c > 0$. Now if $W_f \leq 4n/c$, then we have an $O(\sqrt{n})$ ratio since $d(\mathcal{S}) \geq n$. If $W_f > 4n/c$, then since the length of the shortest path from s' to s is at most $2n$, $d(\mathcal{S}) \geq cW_f - 2n \geq (c/2)W_f$, so we also have an $O(\sqrt{n})$ ratio. We therefore have the following theorem.

THEOREM 4.1. *For two-dimensional scenes $\mathcal{S}$ in which s and t are points and every obstacle is a rectangle whose sides are parallel to the axes, our algorithm achieves a ratio of $\rho(R, n) = O(\sqrt{n})$.*

5. A tactile robot suffices. In this section, we use a technique due to Baeza-Yates et al. [1] to demonstrate that all of our algorithms given so far can be modified to work with essentially the same ratio bounds even if the robot is tactile: it learns about obstacles on bumping into them and can infer the size of an obstacle only by moving along its boundary.

Suppose that the robot hits a side of a rectangular obstacle. Let d be the distance from its present position p to the nearest corner of the obstacle; it does not know d or the direction in which this nearest corner lies. The robot can reach this corner traveling a distance at most $9d + 2$ by applying the following "doubling" procedure suggested by Baeza-Yates et al. until a corner is reached. Walk along the side of the obstacle one unit in one direction, then turn back and walk two units past p in the other direction; turn back, and continue in this manner walking 2^{i-1} units past p on the ith iteration. If desired, in case a corner is reached which the robot is not certain is the nearest to p, the robot can simply walk an equal distance from p in the opposite direction to check. A simple analysis shows that the total length of the walk is at most $9d + 2$.

It remains to show that in each of the deterministic algorithms we have described, we can use this procedure to ensure that a tactile robot suffices (with a constant-factor overhead in the ratio). In our sweep algorithm for the wall problem of section 2, note that in negotiating an obstacle our decision is essentially based on the distance to the nearest corner of the obstacle. By using the above doubling procedure, we thus travel at most nine times the distance that the visual robot does, plus a low-order term for the additive constant.

Next, consider the room problem. In our algorithm Oriented-Room-Find, the only parts that required vision were the brute-force path, and finding a point between $\tilde{t}$ and t if $\tilde{t}$ was inside an obstacle in Step 1. Both can be handled with constant-factor overhead by the procedure of Baeza-Yates et al.

6. More general obstacle types.

6.1. Arbitrary rectangular obstacles. What if rectangular obstacles with sides not parallel to the axes are allowed in the room problem? We begin by proving

two theorems that demonstrate the difference between scenes containing only oriented rectangular obstacles and scenes containing arbitrary rectangular obstacles.

THEOREM 6.1. *For infinitely many n, there exist scenes $\mathcal{S}$ for the room problem containing rectangular obstacles whose sides are at arbitrary angles for which $d(\mathcal{S}) \geq \pi n^{3/2}/81$.*

Thus the length of the shortest path between s and t is not always bounded above by the L_1 distance as in the oriented case.

THEOREM 6.2. *For any deterministic robot R, there exist scenes $\mathcal{S}$ for the room problem containing rectangular obstacles whose sides are at arbitrary angles for which $\rho(R, n) = \Omega(\sqrt{n})$.*

Thus the upper bound for oriented rectangles cannot be achieved in this case.

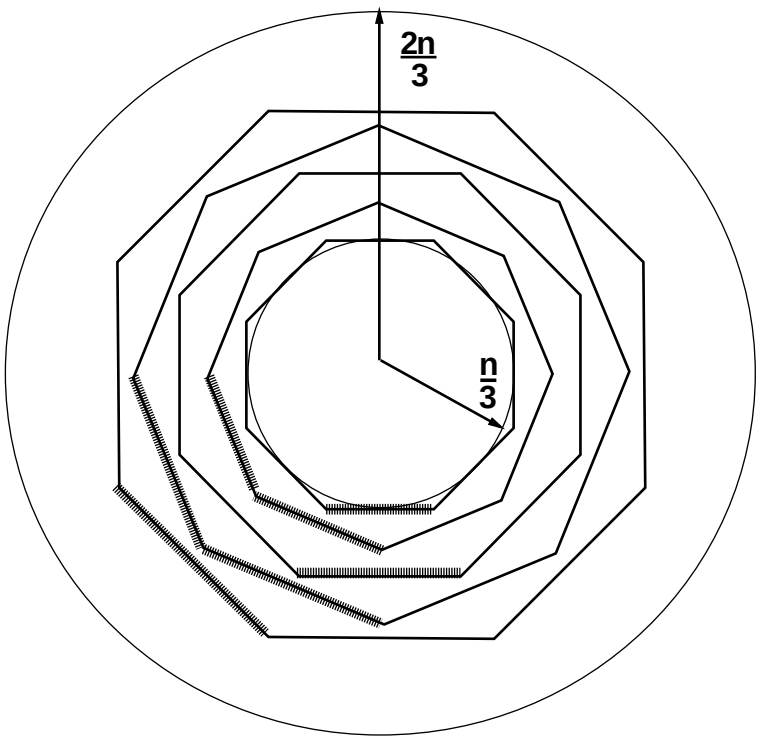


FIG. 4. *The lower bound in Theorem* 6.2. *The obstacles touched by the robot are shaded.*

Proof of Theorem 6.1. Consider $\lfloor n/27 \rfloor + 1$ circles centered at t with radii $\lceil n/3 \rceil + 1 + 9i$, $i = 0, \ldots, \lfloor n/27 \rfloor$. (See Fig. 4.) Inscribe in each a regular $\lfloor \sqrt{n} \rfloor$-gon, aligning all these polygons. Rotate all the polygons inscribed in circles of radii $\lceil n/3 \rceil + 9i$ for even i by an angle $\pi/\sqrt{n}$. Each edge of each polygon can now be replaced by a rectangular obstacle of unit width (in the radial direction) and length very nearly the length of that edge. The length of each obstacle is at least $2\pi\sqrt{n}/3$. Now any obstacle-avoiding path between s and t has to walk a distance of at least $2\pi\sqrt{n}/3$ going from a vertex of the polygon (i.e., gap between the obstacles) on the circle with radius $\lceil n/3 \rceil + 18i$ to a vertex on the circle with radius $\lceil n/3 \rceil + 18i + 18$ for $0 \leq i < n/54$. □

Proof of Theorem 6.2. Consider the scene described in the proof of Theorem 6.1.

We allow a (deterministic) robot to walk from s to t. We now remove from the scene any obstacle not touched by the robot. Let T be the number of obstacles it touches. There is a constant c_1 such that the distance walked by the robot between touching a corner of every *fourth* new obstacle is at least $c_1\sqrt{n}$. This sums to a distance of at least $c_1 T\sqrt{n}/4$. The total area of the obstacles touched by the robot is bounded from above by $2c_1 T\sqrt{n}$. Thus there exists an angle $2i\pi/\sqrt{n}$, $1 \le i \le \sqrt{n}$, such that the path from t to s given by staying on the radius at this angle and going "around" obstacles encountered is of distance at most $c_2 T$ for some constant c_2. This implies that $\rho(R, n) = \Omega(\sqrt{n})$. Notice that the lower bound holds only if the robot is tactile and cannot use any visual information. To make the lower bound work in the case of a robot that uses visual information, we use a slightly different construction together with a technique given in [26].

Again consider $\lfloor n/27 \rfloor + 1$ circles centered at t with radii $\lceil n/3 \rceil + 1 + 9i$, $i = 0, \ldots, \lfloor n/27 \rfloor$, and inscribe in each a regular $\lfloor \sqrt{n} \rfloor$-gon, aligning the polygons. This time, however, rotate all the polygons inscribed in circles of radii $\lceil n/3 \rceil + 18i$ and $\lceil n/3 \rceil + 18i + 9$ for even i by an angle $\pi/\sqrt{n}$. For each i, call the polygons inscribed in circles of radii $\lceil n/3 \rceil + 18i$ and $\lceil n/3 \rceil + 18i + 9$ a *layer*. Each edge of the inner polygon of each layer is replaced by a rectangular obstacle as above, except that it has thin openings spaced at unit distances. The outer polygon has similar obstacles in all edges but one, which has a solid rectangular obstacle with no holes. The holes in each inner polygon are out of alignment with the corresponding holes in the outer polygon; thus the robot cannot see through any layer.

We run the robot algorithm layer by layer, and we make the first obstacle seen by the robot in each outer polygon a solid obstacle. By the same argument as above, it follows that the robot walks $\Omega(\sqrt{n})$ between every fourth layer. □

We now turn to upper bounds. Define the angle of a rectangle to be the angle of its longest edge with the x-axis. We first describe a modification of algorithm Oriented-Room-Find to handle not just obstacles of angles of 0 and $\pi/2$ but obstacles angled in the range $[0, \pi/2]$ as well. Note that we do *not* allow obstacles angled in the remaining range of $(\pi/2, \pi)$. Then we describe how this new algorithm can be modified for scenes $\mathcal{S}$ where there is a fixed known excluded range (d_1, d_2) of angles (for example, $d_1 = \pi/5$ and $d_2 = \pi/4$). Let $\tilde{n} = n/\alpha$, where $\alpha = d_2 - d_1$. Our algorithm achieves $R(\mathcal{S}) = (\tilde{n} \cdot 2^{3\sqrt{\log \tilde{n} \log\log \tilde{n}}})$. The length of the shortest path in such scenes is $O(n)$. We remark that for "practical" cases, it may be enough to consider scenes where there is a known excluded range, especially when the number of different angles is small. Finally, we give a randomized algorithm that achieves $\rho(R, n) = \sqrt{n} \cdot 2^{O(\sqrt{\log n \log\log n})}$ regardless of the angles of the obstacles.

Suppose that the obstacles are angled in the range $[0, \pi/2]$. We describe our algorithm for this case in two steps. We first show that we need only consider the case where obstacles are zero-width line segments angled in the range $[0, \pi/2]$ such that at most some constant number of obstacles cross any line of length one. We then give an algorithm for that special case.

The idea for translating into the zero-width case is to view each obstacle as either two or four line-segment obstacles consisting of its edges that are in the legal angle range. (All four edges are in the legal range only if the obstacle is oriented.) One can easily verify that this implies a constant upper bound on the number of obstacles that cross any given line segment of length one. Let us first assume that when the robot touches an obstacle, it is given its entire description (as has been our standard model). Therefore, the robot can simulate an algorithm for the zero-width case,

and if the simulated algorithm enters a real obstacle, the robot just waits until that algorithm exits and then meets it at the exit point and continues. By going around the obstacle in the shortest way, the (actual) robot travels at most twice as much as the simulated algorithm. To do the simulation with a tactile robot, notice that when an obstacle edge is touched, the robot can determine whether it should be treated as a zero-width obstacle or as empty space based on its angle. If the obstacle is to be treated as empty space, the robot can use the technique of Baeza-Yates et al. to find the point where the simulated algorithm would exit the (actual) obstacle with only a constant factor additional cost. The key point here and in the previous case is that the longer edges of an obstacle are never treated as empty space.

We now describe the algorithm for the zero-width case. The reason for reducing to this case is that since every obstacle edge now has an angle in the range $[0, \pi/2]$, we can perform greedy $\langle +x, +y\rangle$ and $\langle +y, +x\rangle$ paths. The reason we cannot immediately use algorithm Oriented-Room-Find, however, is that we can no longer make the greedy $\langle +x, -y\rangle$ and $\langle +y, -x\rangle$ paths required in Step 3. Instead, we will replace that portion of the algorithm with a less efficient binary-search strategy.

More precisely, let us say our start point is (x_0, y_0) and t is at (n, n). In contrast to Oriented-Room-Find, our invariant will be that we have *two* monotone paths: a $\langle +y, +x\rangle$ path from (x_0, y_0) to some point to the west of t (i.e., a point (x, n), where $x \leq n$) and a $\langle +x, +y\rangle$ path from (x_0, y_0) to some point to the south of t. As in Oriented-Room-Find, we begin by recursively (or using brute-force if the distance to t is sufficiently small) traveling to a temporary point t' defined as in that algorithm at distance $O(m)$ from s. Now in place of Steps 2 and 3 of that algorithm, we will instead use a binary search (described below) to find a point with either the same x-coordinate or the same y-coordinate as t' (and to the northeast of (x_0, y_0)) with the following property P: the greedy $\langle +x, +y\rangle$ and $\langle +y, +x\rangle$ paths from this point pass to the south and west of t, respectively. Thus we will maintain our invariant, increasing either the x- or y-coordinate of the new "start point" as in Oriented-Room-Find.

We now show how to find the desired point. First, if t' is such a point, we are done. Otherwise, suppose that both greedy $\langle +x, +y\rangle$ and greedy $\langle +y, +x\rangle$ paths starting at t' hit points to the south of t. (The other case is analogous.) Let s' be a point with the same y-coordinate as t' on the $\langle +y, +x\rangle$ path from (x_0, y_0) given by our invariant. Since the $\langle +y, +x\rangle$ path from s' hits a point to the west of t, if the $\langle +x, +y\rangle$ path from s' hits a point to the south of t, then we are done as well. Otherwise, we travel to a point t'' halfway between s' and t'—using the same procedure as that used to reach t'—and examine the $\langle +x, +y\rangle$ and $\langle +y, +x\rangle$ paths from t''. (If t'' as defined is inside an obstacle, we examine the two points to the west and east of t'' on that obstacle boundary.) Depending on the outcomes of these greedy paths, we either halt with success or continue the binary search with a new t''' and so on. We stop the binary search when either success is discovered or the interval under consideration has length at most 1. Thus at most $\lceil \log n \rceil$ iterations of the binary search will be made. If the binary search stops because the interval remaining is too short, a point with property P can easily be found by traveling from the west endpoint to the east endpoint of the interval and, each time an obstacle is hit (this can happen at most a constant number of times), testing it for property P before going around the obstacle, which costs only $O(n)$. This strategy succeeds because if two points a and b have the same y-coordinate and there is either no obstacle, between them or both are at the boundary of the same obstacle, then the $\langle +x, +y\rangle$ path from the leftmost point intersects the $\langle +y, +x\rangle$ path from the rightmost point.

We get that the total distance is given by:

$$T(n) \;\le\; \lceil 2n/m \rceil \, \lceil \log n \rceil \, [T(m) + cn] \quad \text{for some constant } c.$$

Substituting $m = \lceil n/2^{\sqrt{\log n \log\log n}} \rceil$ yields

$$T(n) = O\left(n \cdot 2^{3\sqrt{\log n \log\log n}}\right).$$

This strategy can be used for a smaller range (d_1, d_2) of excluded angles by just performing a rotation and a coordinate transformation on the space. Essentially, instead of writing t as $n\vec{x} + n\vec{y}$ for orthogonal unit vectors $\vec{x}$ and $\vec{y}$, we may write t as $(n'\vec{d_1} + n''\vec{d_2})$, where $\vec{d_1}$ and $\vec{d_2}$ are unit vectors in the d_1 and d_2 directions. It is not difficult to see that both n' and n'' are $O(n/\alpha)$, where $\alpha = d_2 - d_1$. Let $\tilde{n} = n/\alpha$. The performance of the previous algorithm after the transformation is $R(\mathcal{S}) = O(\tilde{n} \cdot 2^{3\sqrt{\log \tilde{n} \log\log \tilde{n}}})$ since the lengths are changed by at most a factor of $1/\alpha$.

THEOREM 6.3. *There is a deterministic algorithm for the room problem with an excluded angular range of size α that achieves $R(\mathcal{S}) = O(\tilde{n}2^{3\sqrt{\log \tilde{n} \log\log \tilde{n}}})$. Here $\tilde{n} = n/\alpha$.*

Now consider the general case where the angles of the obstacles may be in any range. A simple pigeonholing argument implies that a constant fraction of the ranges $[i\pi/\sqrt{n}, (i+1)\pi/\sqrt{n}]$ for $0 \le i < \sqrt{n}$ have the property that the total perimeter of the obstacles angled in this range is no more than $2/\sqrt{n}$ of the total perimeter. To bound the total perimeter, note that from our assumption that a unit circle can be inscribed in each obstacle, it follows that the perimeter of an obstacle, is always less than four times its area. Since the total area of all obstacles is at most n^2, the total perimeter of obstacles in such a range is $O(n^{3/2})$.

Consider a randomized algorithm that first guesses such a range. It then applies the above algorithm assuming that there are no obstacles with angles in this range. On actually encountering any obstacle in this range, it just goes around the obstacle at cost at most the perimeter of the obstacle. From the definition of the "forbidden angle range," it follows that on any given greedy path, the robot will go around any such obstacle at most once. Therefore, the expected total distance walked by this algorithm is given by the recursion given above where $\tilde{n} = n^{3/2}$, and a constant times $n^{3/2}$ is added to the $c\tilde{n}$ term (which remains $O(\tilde{n})$). Thus we obtain the solution $T(n) = n^{3/2} \cdot 2^{O(\sqrt{\log n \log\log n})}$.

THEOREM 6.4. *There is a randomized algorithm that achieves a ratio of $\sqrt{n} \cdot 2^{O(\sqrt{\log n \log\log n})}$ for the room problem provided that every obstacle is a rectangle within which a unit circle can be inscribed.*

6.2. Arbitrary convex polygons. We now describe how our randomized algorithm for the room problem can be extended to handle arbitrary convex polygons provided that a unit circle can be inscribed in each obstacle and that the entire description of an obstacle is given to the robot when that obstacle is touched. (The only part of the description of the obstacle that is required is the angle that its longest diagonal makes with the x-axis.) We do not have a solution for the wall problem with arbitrary convex obstacles, and thus we have no solution for point-to-point navigation with convex obstacles.

We define the *angle* of a convex polygonal obstacle to be the angle its longest diagonal makes with the x-axis. The idea for the conversion is that each time the

robot encounters an obstacle, it picks a longest diagonal D and treats that obstacle as a collection of line segments parallel to D. In particular, it imagines a line segment at D and then additional segments (if any) parallel to and at distance 1, 2, 3, etc. from D, each as long as possible to still be contained within the obstacle. It then feeds this collection of line segments to the algorithm for rectangular obstacles. As in the case for unoriented rectangular obstacles, suppose the line-segment algorithm wishes to travel a path along line segments that leads through one of the convex obstacles: say the path is between points a and b on some obstacle's border. The robot then simply travels the shortest path from a to b along the obstacle boundary. Since the line segments are parallel to the longest diagonal of the obstacle and the obstacle is convex, we are guaranteed that the shortest path along the obstacle between a and b is at most a constant multiple of the straight-line path.

We note that in case there is a fixed known excluded range of angles, then the algorithm of Theorem 6.3 can be extended as well.

7. Extensions to three dimensions. This section summarizes extensions of our techniques to three dimensions. We begin by extending our study of the wall problem to three dimensions, and then we extend our optimal algorithm for point-to-point navigation to three dimensions.

7.1. The wall problem in three dimensions. Suppose that t is an infinite plane perpendicular to the x-axis at distance n from the origin s. We begin by extending the lower bound of [26] to three dimensions, showing a lower bound of $\Omega(n^{2/3})$. We then give a generalization of the two-dimensional sweep algorithm that achieves a matching upper bound.

THEOREM 7.1. *For any deterministic robot, there are scenes $\mathcal{S}$ of the three-dimensional wall problem for which $\rho(R, n) = \Omega(n^{2/3})$.*

Proof. To prove the lower bound, it will be convenient to assume a tactile robot. Using the technique of [26] used in section 6.1, this proof can be extended to robots with visual capabilities.

As the robot walks in the direction of t, the adversary places obstacles as follows. Each obstacle is a cuboid whose cross-section parallel to the yz-plane is a square of side $n^{2/3}$ and whose width in the x-direction is one. Whenever the robot first reaches x-coordinate i for each $i \in \{0, 1, \ldots, n-1\}$, a cuboid is placed directly in front of it. Thus the robot must travel a distance at least $\frac{1}{2}n^{2/3}$ perpendicular to the x-axis in order to advance one unit parallel to the x-axis. Thus $R(\mathcal{S}) \geq \frac{1}{2}n^{5/3}$.

We now show that $d(\mathcal{S}) \leq 3n$. Since the cross-sectional area of each cuboid is $n^{4/3}$, by the pigeonhole principle, there is a line ℓ parallel to the x-axis with the following properties: (a) its distance from the x-axis is at most n; (b) it cuts at most $n^{1/3}$ cuboids. Consider a path that starts from s and first goes to the leftmost point of ℓ. It then goes along ℓ parallel to the x-axis, traveling around each cuboid it encounters.

The distance from s to the leftmost point of ℓ is at most n. The distance traveled parallel to the x-axis is also n. The total perpendicular distance traveled in circumventing the cuboids cut by ℓ is at most $n^{1/3} \times n^{2/3} = n$. Therefore, $d(\mathcal{S}) \leq 3n$ and the ratio $\rho(R, n) = \Omega(n^{2/3})$. □

We now give an algorithm that matches this lower bound to within a constant. At a high level, the algorithm can be viewed as an extension of the two-dimensional sweep algorithm. The window, which in the plane was the region between two lines parallel to the x-axis, now becomes a cylinder whose axis is the x-axis. The radius of

this cylinder is initially n and is subsequently increased at certain points. The sweep used in the plane is now replaced by a spiral about the x-axis.

For simplicity, we first describe the algorithm as if every obstacle were a cylinder of circular cross-section with its axis parallel to the x-axis and its center placed directly in front of the robot. The radii and lengths of these obstacles can vary. Following the analysis of this simple case, we outline the extension of the algorithm to more general obstacles.

Consider a point orbiting around a fixed point, with the radius of the orbit increasing linearly with angular position at a rate of D units for every 2π radians of angular position. We call the path of the moving point a *spiral* and D the *spacing* of this spiral. (See Fig. 5.)

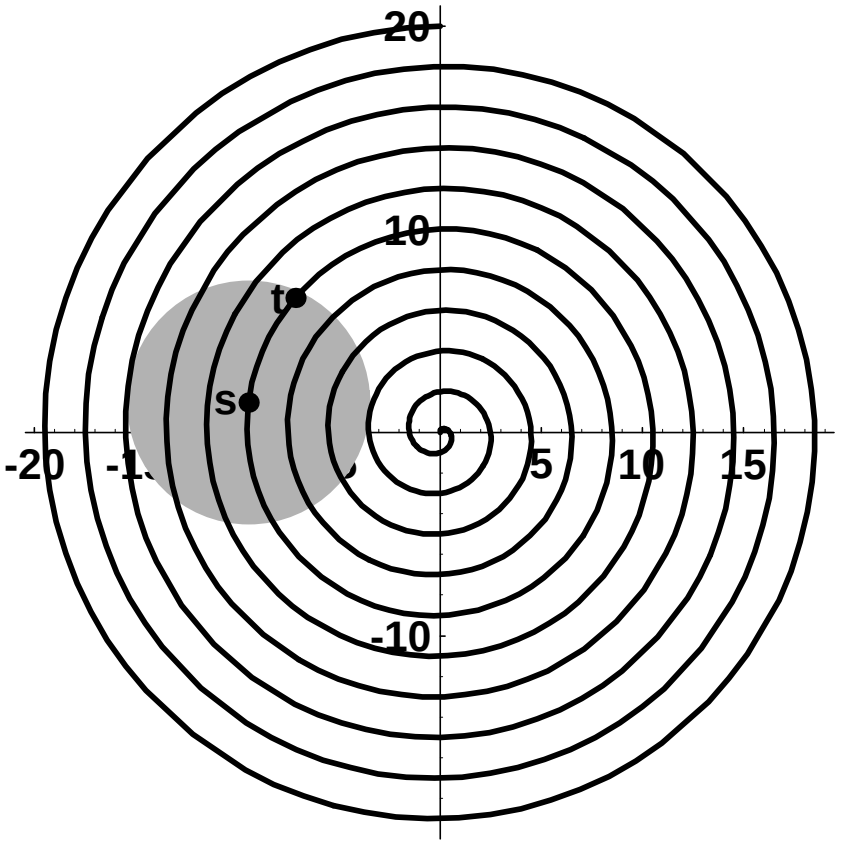


FIG. 5. *A projection of a spiral with radius* 20 *and spacing* 2*, overlaid with a projection of a cylinder of radius* 6*. The point s is the projection of the point where the robot encountered the obstacle, and point t is the projection of the nearest point on the outward spiral that is not covered.*

Our algorithm begins with $W = n$; at all times, $\tau = W/n^{1/3}$. Consider a spiral whose center is on the x-axis and whose orbits lie in a plane perpendicular to the

x-axis. The spacing will be $\tau/3$. Thus there are at most $3n^{1/3}$ orbits in the spiral within the current window.

The y- and z-coordinates of the robot will always lie on such a spiral. Analogous to the sweep direction in the plane (north or south), the robot now maintains a spiral direction that is either "outwards" or "inwards" along the spiral. On encountering a (cylinder) obstacle directly in front of it, the robot first checks if the radius of the cylinder exceeds τ. If not, the robot "goes around" the cylinder, retaining its current yz-coordinates. On the other hand, if the radius does exceed τ, the robot proceeds to the nearest point p on the spiral along its current spiral direction (outwards or inwards) that is not covered by the cylinder, proceeding as far along the x-direction as it can in the process. If this nearest point lies at a distance W' from the x-axis that exceeds W, the robot increases W to $2W$, resets the sweep counter to zero, and proceeds to begin a new spiral inwards from this point. Whenever the robot completes an inward spiral by reaching the x-axis or an outward spiral by reaching a point at distance W from the x-axis, it increments the sweep counter. Whenever the sweep counter reaches $n^{1/3}$, the robot doubles W (and τ), resets the sweep counter to zero, and continues.

THEOREM 7.2. *The spiral algorithm achieves a ratio of $O(n^{2/3})$, provided that every obstacle is a cylinder whose center is directly in front of the robot when it is first encountered.*

Proof. The analysis is essentially identical to that of Theorems 2.1 and 2.2. Let W_f be the final window radius. Since there are at most $3n^{1/3}$ orbits in each spiral, the distance walked by the robot in the last completed spiral is $O(n^{1/3}W_f)$. Since there are at most $\lceil n^{1/3} \rceil$ complete spirals in each window, the total distance walked by the robot is $O(n^{2/3}W_f)$.

We show that the length of the shortest path is $\Omega(W_f)$. First, we may assume that the robot has completed $\lceil n^{1/3} \rceil$ spirals for some window size $W \geq \frac{1}{2}W_f$, or else we are immediately done. (This is by the same reasoning used for the two-dimensional wall problem.) Consider a point on the shortest path that is farthest from the x-axis. If the distance of this point to the x-axis is at least W, then clearly the shortest path has length at least W, so we may assume that this is not the case.

Given a shortest path, for each of the completed spirals, define its first entry point to be the first point on the shortest path whose x-coordinate is the same as the x-coordinate of the starting point of the spiral. Similarly, define its first exit point to be the first point on the shortest path whose x-coordinate is the same as that of the end point of the spiral. As in the two-dimensional problem, the exit point of spiral i appears before the entry point of spiral $i+1$. Therefore, we need only show that for each spiral in some window of size W, the yz-plane component of the shortest path from the entry point to the exit point of the spiral is $\Omega(\tau)$. This will imply that the total yz-plane component of the shortest path is $\Omega(\tau n^{1/3}) = \Omega(W)$. Therefore, imagine projecting the completed spiral onto the yz-plane, projecting all the cylinders encountered in that spiral onto circles in the yz-plane. Observe that every point on the spiral is at distance at most $\tau/2$ from the center of one such circle of radius at least τ. Since the orbits of the spiral are at distance $\tau/3$ from one another, for any point of distance at most W from the origin, there exists a circle such that the distance of this point from the periphery of the circle is $\Omega(\tau)$. In particular, this also holds for the projection of the entry point of the sweep. Thus the yz component of any path from the entry point to the exit point must be $\Omega(\tau)$. □

The extension to the case of general cylindrical obstacles is similar. We define a

general cylindrical obstacle to be one for which there is a simple closed curve C in the yz-plane such that the obstacle's intersection with any plane perpendicular to the x-axis when translated to the yz-plane is either empty or it is C and its interior. As long as the robot moves in the positive x-direction, it will hit an obstacle only at some point of its unique "west face" in the yz-plane. On encountering such an obstacle, the robot measures the shortest distance from its present position to a point p on the spiral not touched by the obstacle. If this quantity is less than τ, it uses this shortest path to circumvent the obstacle and retain its yz-coordinates. Otherwise, it goes to p and proceeds as far along the x direction as it can. The analysis is very similar to the case of unit-height cylinders.

THEOREM 7.3. *For three-dimensional scenes $\mathcal{S}$ with general cylindrical obstacles and in which s and t are points, our spiral algorithm achieves a ratio of $\rho(R, n) = O(n^{2/3})$ for the wall problem.*

7.2. Point-to-point navigation in three dimensions. We now give an upper bound for point-to-point navigation in three dimensions that matches the lower bound to within a constant factor provided that every obstacle is a cuboid whose sides are parallel to the axes. As in two dimensions, our upper bound for point-to-point navigation comes from combining an algorithm for point-to-plane navigation and another for the room problem. However, in the three-dimensional case, it suffices to combine the three-dimensional wall algorithm with the two-dimensional room algorithm to obtain a three-dimensional point-to-point navigation algorithm. For simplicity of analysis, we assume that all obstacles have vertices at integral coordinates. However, our algorithm would still work provided that a unit cube can be inscribed within every cuboid in the scene.

Suppose without loss of generality that the x-, y-, and z-coordinates of s are less than those of t. The algorithm consists of three stages. In the first stage, the robot reaches a point s'' such that at least two of its coordinates are the same as t. This is done as follows. The robot starts by taking a greedy $\langle +x, +y, +z\rangle$ path until one of the three coordinates is the same as t. Call this point s' and without loss of generality say the y-coordinates of s' and t are the same. Next, fixing the y-coordinate (i.e., staying in the xz-plane of point s'), the robot takes a greedy $\langle +x, +z\rangle$ path from s' until one of the other two coordinates is the same as t. The endpoint of this path is the desired point s''. Without loss of generality, assume that the y- and z-coordinates of s'' are the same as t and let $n_x \leq n$ be the distance between s'' and t. The total distance walked in the first stage is $O(n)$.

In the second stage, the robot uses the three-dimensional wall algorithm from s'' to reach a point t' with the same x-coordinate as t. See Fig. 6. The total distance walked in this step is at most $O(n^{2/3})$ times $d(\mathcal{S})$. Let n_w be the distance between t' and t.

Assume that t' is not t (otherwise we are done). Consider the plane that contains the three points s'', t', and t. (See Fig. 6.) In the third stage, the robot will stay in this plane. Notice that since all of the obstacles are cuboids, the intersections of all obstacles with this plane are oriented rectangles. Define w to be a linear combination of y- and z-directions so that points on this plane can be written in (x, w) coordinates, and translate these so that $s'' = (0, n_w)$, $t' = (n_x, 0)$, and $t = (n_x, n_w)$ in this system.

From t' do a greedy $\langle +w, -x\rangle$ path until either the w-coordinate is n_w or else the x-coordinate is 0, whichever comes first. If the first case occurs, then the greedy path is a monotone boundary and we can apply the room-problem algorithm of Theorem 3.2. (The two-dimensional slice may technically violate our conditions for the

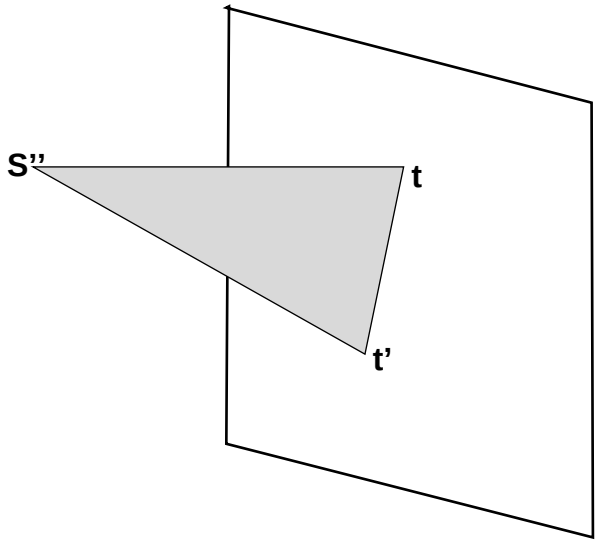


FIG. 6. *The plane that contains the points s'', t', and t.*

room problem by having obstacles that are too "thin." However, because a unit cube can be inscribed in each of the three-dimensional obstacles, there is sufficient separation for the room-problem algorithm to work.) If the second case occurs (we reached a point with x-coordinate 0), then go back to s'', retracing all our steps if we have to, and perform a greedy $\langle +x, -w \rangle$ path from there. This is guaranteed to hit a point with x-coordinate of n_x and w-coordinate at least 0 since it cannot cross our previous greedy path. Therefore, we again have a room and can run the room algorithm. The distance walked in this stage is $O(\sqrt{n} \cdot d(\mathcal{S}))$ since $n_w \leq d(\mathcal{S})$. We therefore have the following theorem.

THEOREM 7.4. *For three-dimensional scenes $\mathcal{S}$ in which s and t are points and every obstacle is a cuboid whose sides are parallel to the axes, our algorithm achieves a ratio of $\rho(R, n) = O(n^{2/3})$.*

To extend this result to a tactile robot, we again use the technique due to Baeza-Yates et al. [1]. Their strategy allows one to start from a face of a cuboid and travel to the nearest edge (at distance d), walking distance $O(d)$ in the process, without prior knowledge of d or the direction to the nearest edge. As in section 5, this allows our algorithm for point-to-point navigation in three dimensions to work for tactile robots with the same asymptotic ratio bounds.

8. The power of randomization. We now consider randomized robots that toss coins as they walk from s to t. The scene $\mathcal{S}$ is fixed in advance by an *oblivious* adversary [3] who knows the randomized algorithm but not the coin tosses made by the robot during a walk. The cost of robot R on scene $\mathcal{S}$ is now a random variable; we thus define the ratio $\rho(R, n)$ to be $\sup_{\mathcal{S} \in \mathcal{S}(n)} E[R(\mathcal{S})]/d(\mathcal{S})$. The main result of this section is a randomized algorithm for the two-dimensional wall problem that achieves a ratio of $2^{O(\sqrt{\log n \log \log n})}$ provided that the obstacles are all vertical line segments with endpoints at integral x-coordinates and the robot is allowed *vision*. Notice that for this situation, the robot can see the entire "column" of obstacles directly in front of it; that is, if the robot is at a point with x-coordinate in the range $(i-1, i)$ for integer

i, it can see all obstacles of x-coordinate i. To keep with our previous conventions on the thickness of obstacles, we could equivalently consider obstacles of width between one and two having their left walls only at even x-coordinates; this would still allow the robot to see an entire "column" at once.

The Papadimitriou–Yannakakis lower bound of $\Omega(\sqrt{n})$ still holds for deterministic algorithms for this restricted class of scenes [26]. Therefore, for such scenes, a randomized algorithm is provably better than a deterministic one. We leave as an open question whether one can achieve similar bounds for the more general wall problem.

The idea for the randomized algorithm is to view the problem as a k-server problem on $(k+1)$ equally spaced points on a line and then use as a subroutine known randomized strategies [4] for that server problem. For the benefit of the reader, we now define the *k-server problem*, first defined in [23]. An on-line algorithm manages k mobile servers located at the vertices of a graph G whose edges have positive real lengths. The algorithm has to satisfy on-line a sequence of requests, each of which is some vertex v of G, by moving a server to v unless it already has a server there. Each time it moves a server, it pays a cost equal to the distance moved by that server. We compare the cost of such an algorithm to the cost of an adversary that, in addition to moving its servers, also generates the sequence of requests. In fact, our problem can be better described as a *metrical task system* of [6], but we will use the language of servers here. In the lower bound direction, a recent result of Karloff et al. for the server problem shows that even for the special case of scenes we consider, no randomized algorithm can achieve a constant ratio [17].

We now present our randomized algorithm. There is a randomized strategy for k servers on $k+1$ equally spaced points on the line that achieves competitiveness $2^{O(\sqrt{\log k \log\log k})}$ against the oblivious adversary [4]. (For completeness, details are given in the appendix.) We map the navigation problem to this k-server problem as follows. Let $k = n-1$ and define the spacing between adjacent points on the line to be W/n, where W is the width of a window of y-coordinates currently considered by the robot; the value of W will be specified below. Each point in the server problem corresponds to a range of W/n y-coordinates for the navigation problem. The "hole" (the point without a server) represents the range currently inhabited by the robot.

We begin with $W = n$ and start the hole at the center of the line. Each time the robot sees a column of obstacles, the robot notes all points in the server problem corresponding to ranges that are completely blocked by obstacles. It then makes enough requests to the server algorithm on those points so that for the server algorithm of [4], the hole no longer resides on such points. Note that this request sequence is determined by the scene and thus obeys the definition of an oblivious adversary. The robot then moves to the range occupied by the hole (if it is not already there) and then moves a vertical distance at most W/n to find a point where it can go forward in the $+x$ direction to the next column. Thus the distance moved by the robot is at most the on-line server cost plus $W/n+1$ for each unit moved in the $+x$ direction. If the off-line server cost reaches W, the robot doubles the window width and restarts the server algorithm; each point now corresponds to a larger range of y-values.

THEOREM 8.1. *The randomized algorithm above achieves a ratio of* $2^{O(\sqrt{\log n \log\log n})}$ *for the wall problem in the plane where the robot uses vision and the obstacles are vertical line segments at integral x-coordinates.*

Proof. For a fixed window width, the off-line server cost in the above transformation is a lower bound on the length of the shortest path for the robot problem (assuming the off-line hole is also started at the center of the line). The off-line server

cost could be a bit lower than the length of the shortest path since we do not make requests to points corresponding to y-value ranges only partially blocked by obstacles. Note that when the off-line cost exceeds W, the shortest path might escape the window, which is why W is doubled.

As mentioned previously, the on-line cost for the robot is a most the on-line cost for the server problem plus $W/n + 1$ for each unit advance in the x-direction. Therefore, if W_f is the final window width used, the total distance traveled by the robot is at most $(W_f + n) + W_f 2^{O(\sqrt{\log n \log\log n})} = d(\mathcal{S})2^{O(\sqrt{\log n \log\log n})}$. □

9. Nonconvex obstacles and mazes. When the obstacles are nonconvex, the scene can be a maze. In this case, it is easy to see that $\rho(R, n)$ cannot be bounded by any function of n (the Euclidean distance between s and t). Instead, we prove a ratio between $R(\mathcal{S})$ and $d(\mathcal{S})$ as a function of the total number of vertices in all the obstacles, $|V|$.

THEOREM 9.1. *No randomized algorithm achieves a ratio better than* $(|V| - 10)/6$.

Proof. Consider the maze in Fig. 7 and its obvious generalization.

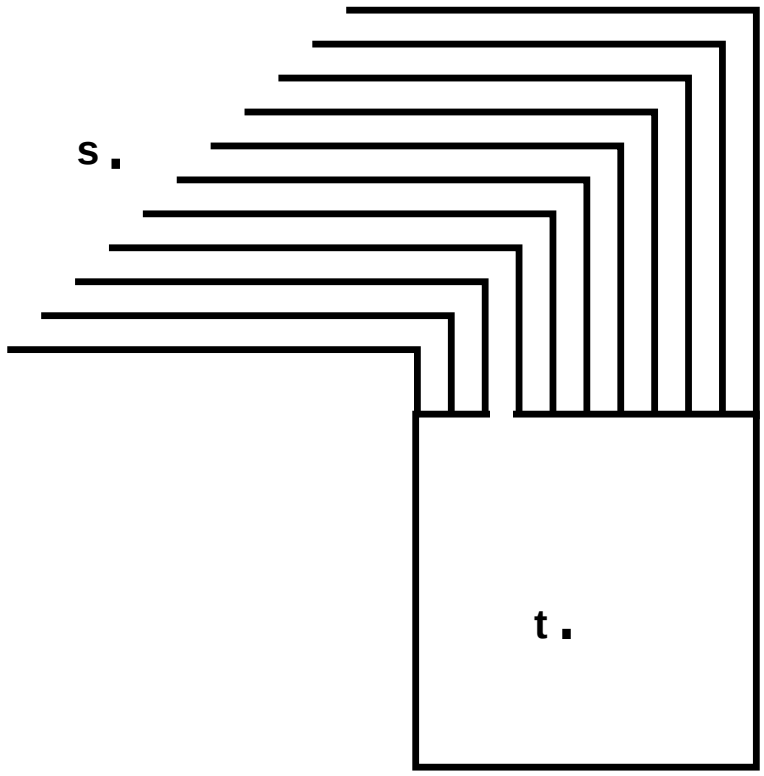


FIG. 7. *A maze achieving the lower bound. Each line segment corresponds to an obstacle with four vertices.*

The maze has $(|V| - 10)/6$ passages that could lead from s to t. An algorithm attempts various passages in turn until it finds the sole passage open to t. For any randomized algorithm, there is one passage whose expected "time to attempt" is at

least (number of passages − 1)/2; this passage is left open to t. The robot walks $2d(\mathcal{S})$ on every failure before that attempt, and $d(\mathcal{S})$ on that attempt. □

The bound applies a fortiori to deterministic algorithms. Rao et al. [28] give a deterministic algorithm that explores a maze by building a map of the scene, proceeding at each step to that unexplored vertex of the maze nearest to the vertices that have already been visited. It is easy to show that this algorithm achieves a ratio of at most $2|V|$, matching the above lower bound to within a constant. This algorithm is memory intensive, and this may be a handicap when space is limited or the scene changes quickly enough that a map is not worth building. We now give a simple, memoryless, randomized alternative based on a random walk that works for scenes in the plane. We first define a graph $G(\mathcal{S})$ on the vertices of the polygons in $\mathcal{S}$ and prove a simple geometric property of this graph. We then describe how the robot can perform a random walk on this graph, and we invoke a result on random walks to prove that the robot's ratio is $O(|V|)$.

The graph $G(\mathcal{S})$ is defined as follows: each vertex of a polygon in $\mathcal{S}$ is a node in the graph. A node v in $G(\mathcal{S})$ chooses up to twelve neighbors, defined as follows. Consider the twelve cones defined by angular intervals $[\pi i/6, \pi(i+1)/6)$, $i = 1, 2, \ldots, 12$ about v. There is an edge joining v to the nearest visible vertex (if any) in each cone. Thus $G(\mathcal{S})$ has at most $12|V|$ edges. A construction similar to $G(\mathcal{S})$ appears in Clarkson [11], where a result similar to the following lemma was given:

LEMMA 9.2. *Let $d_{st}(\mathcal{S})$ be the distance between two vertices s and t in the scene $\mathcal{S}$. There is a path in $G(\mathcal{S})$ between s and t of length at most $2.1d_{st}(\mathcal{S})$.*

Proof. For two vertices u and v in the scene that are mutually visible, denote by d_{uv} the distance between them. The shortest path in $\mathcal{S}$ between s and t is a path in the *visibility graph* of $\mathcal{S}$ [32]: a graph whose nodes are the vertices of obstacles in $\mathcal{S}$, with two nodes being joined by an edge if they are visible from each other. We now show that given this shortest path (of length $d_{st}(\mathcal{S})$) in the visibility graph, we can find a path in $G(\mathcal{S})$ between s and t whose length is at most $2.1d_{st}(\mathcal{S})$. Note that we can afford to find this path "off-line": we only wish to exhibit the existence of a short path in $G(\mathcal{S})$ from s to t.

We use an iterative strategy: we take the first edge of the visibility graph on the shortest path, say (s, a). If (s, a) is an edge in $G(\mathcal{S})$, we proceed to a and continue from there. Otherwise, we show that there is an edge (s, b) in $G(\mathcal{S})$ with the following property: $d_{bt}(\mathcal{S}) \leq d_{st}(\mathcal{S}) - 0.48d_{sb}$. (Note that s and b are mutually visible.) We therefore go from s to b in the first step of our path in $G(\mathcal{S})$ from s to t, having ensured that (1) we move to a node whose distance to t in $\mathcal{S}$ is less than from s and (2) the distance we have walked is proportional to the reduction in the remaining distance. We now continue the iteration from b, with b playing the role of s. In fact, since our distance to t diminishes at each iteration, we will have at most $|V| - 1$ iterations before arriving at t.

It remains for us to bound the first step (s, b) when (s, a) is not an edge in $G(\mathcal{S})$. Since s and a are mutually visible, the only reason that segment sa is not an edge in $G(\mathcal{S})$ is that the cone containing the line segment sa has a node b in it such that (s, b) is an edge in $G(\mathcal{S})$, i.e., $d_{sb} \leq d_{sa}$. Consider the sector of the circle with center s and radius d_{sb} that lies in the cone containing the segment sa. Let c be the point where this sector cuts segment sa. Since b is the closest vertex to s in the cone, no obstacle vertex lies in this sector. Further, since both a and b are visible from s, no portion of any obstacle lies in this sector. Therefore, b and c are mutually visible, as

are a and c. Figure 8 illustrates these facts. Thus

$$d_{bt}(\mathcal{S}) \leq d_{bc} + d_{ca} + d_{at}(\mathcal{S}).$$

On the other hand,

$$d_{st}(\mathcal{S}) = d_{sc} + d_{ca} + d_{at}(\mathcal{S}).$$

Using some elementary trigonometry and the fact that the angle between segments sa and sb is at most $\pi/6$, we have

$$d_{bt}(\mathcal{S}) \leq d_{st}(\mathcal{S}) - d_{sc} + d_{bc} \leq d_{st}(\mathcal{S}) - 0.48 d_{sb}. \qquad \square$$

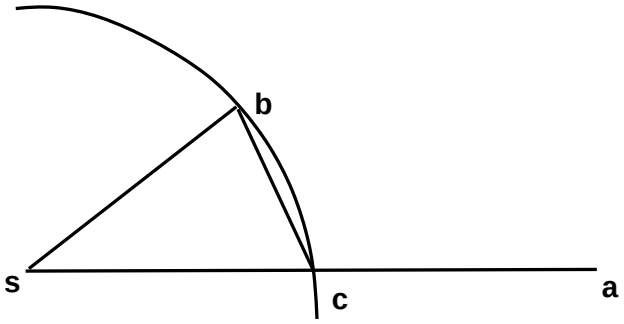


FIG. 8. $d_{bc} < 0.52 d_{sc}$ *and* $d_{sb} = d_{sc}$.

We now describe the random walk that the robot executes in going from s to t. At each node, it looks out to see the nearest node in each of the twelve cones, if such a node exists; let them be $v_1, v_2, \ldots, v_k$ at distances $d_1, d_2, \ldots, d_k$, respectively. Then it chooses to go to v_i with probability

$$\frac{1/d_i}{\sum_{j=1}^{k} 1/d_j}.$$

Note that this probabilistic decision is a local choice that does not need knowledge of $G(\mathcal{S})$ in advance; at each vertex, the robot measures the distance to the nearest visible node in each cone and chooses each with probability inversely proportional to its distance. The robot stops the process on arriving at t.

THEOREM 9.3. *The expected distance traveled by the robot is at most* $50.4|V| d_{st}(\mathcal{S})$. *Thus it achieves a ratio of at most* $50.4|V|$.

Proof. Chandra et al. [8] have studied the following general walk in a graph with positive real edge lengths: at each node, the walk chooses the next edge to walk along with probability inversely proportional to its length. They show that the expected distance traversed by the walk in going from a node a to a node b is at most $2m\ell_{ab}$, where m is the number of edges and ℓ_{ab} is the length of the shortest path in the graph between a and b.

In our case, $m \leq 12|V|$, and $\ell_{st} \leq 2.1 d_{st}(\mathcal{S})$ by Lemma 9.2; combining these facts with the result in [8] yields the theorem. $\square$

Clearly, there is a tradeoff between the number of cones in the graph $G(\mathcal{S})$ that we define and the factor 2.1 in Lemma 9.2; had we used 36 cones each of angle $\pi/18$, we could have gotten a tighter factor there but the number of edges in the graph (which figures in the ratio achieved by the random walk) would have gone up. Our choice of twelve cones optimizes this tradeoff.

10. Open problems. We conclude with some open problems.

- What are the tight bounds (deterministic as well as randomized) for the room problem with general obstacles?
- Can a randomized algorithm for the room problem beat deterministic algorithms?
- Extend the sweep algorithm for the wall problem to handle arbitrary polygonal obstacles and hence or otherwise obtain an algorithm for point-to-point navigation with such obstacles.
- Extend all of the above to three dimensions.
- Give an algorithm that achieves a provably good ratio for three-dimensional scenes with nonconvex obstacles (three-dimensional mazes).
- Blum and Kozen [5] show that a planar maze can be traversed in a number of steps polynomial in the number of vertices in the maze by a deterministic automaton using two pebbles. We have seen that the deterministic algorithm of Rao et al. achieves an optimal ratio but is memory intensive, whereas the random walk achieves a similar ratio without using memory to build a map. Is there a deterministic automaton using few pebbles (small memory) that achieves a good ratio? It seems reasonable to expect that the automaton would need a distance counter as well.

Appendix. We outline the randomized k-server algorithm invoked in section 8 and the proof of its competitiveness.

THEOREM A.1 (see [4]). *There is a randomized algorithm for $k = n - 1$ servers on n equally spaced points on a line that achieves an expected competitiveness ratio of $2^{O(\sqrt{\log n \log\log n})}$ against an oblivious adversary.*

Proof. Without loss of generality, assume that the points are spaced at unit distance. For convenience, we call the point without any server the "hole" and think of the algorithm as being on the hole position and having to move when it is "hit" by a request. The idea of the algorithm is to break the line into a collection of equal-sized intervals and then to stay within some interval until the adversary has made "enough" requests inside it. Once the adversary has made enough requests, the algorithm moves the hole to a different interval, choosing at random from those intervals into which "not too many" requests have been made.

More specifically, the algorithm proceeds as follows. Let $m = \lceil n/2^{\sqrt{\log n \log\log n}} \rceil$.

ALGORITHM RANDOMIZED-LINE

Step 1. Break the line into $\lceil n/m \rceil$ intervals, each of m points except possibly the last. We label these intervals $I_1, \ldots, I_{\lceil n/m \rceil}$. Initialize each interval to be "unmarked."

Step 2. For each point i $(1 \le i \le n)$, initialize a variable $C(i)$ to zero. Each $C(i)$ represents the minimum possible off-line cost of ending at point i given the sequence of requests seen since the last initialization (and assuming the off-line server may start at any point). Updating $C(i)$ is easy: after a request is made at point i, $C(i) \leftarrow \min\{C(i-1)+1, C(i+1)+1\}$. To handle the endpoints, initialize $C(0)$ and $C(n+1)$ to infinity.

Step 3. Randomly choose one unmarked interval I_j. Stay inside the larger interval $I_{j-1} \cup I_j \cup I_{j+1}$, running Randomized-Line recursively within that region, until the

minimum cost $C(i)$ for $i \in I_j$ has risen to be greater than $m/2$. As a base case, for a small enough interval, any deterministic algorithm will do (e.g., the deterministic algorithm given in [10]). For consistency at the endpoints, define I_0 and $I_{\lceil n/m \rceil + 1}$ to be empty and always marked.

Remark. The reason for staying within a larger interval of size $3m$ is a technical one to handle the "edge effects" that occur at the boundaries of the intervals I_j, as discussed in the analysis.

Step 4. Mark all intervals I_j such that the minimum cost $C(i)$ for all $i \in I_j$ is at least $m/2$. If there is some unmarked interval left, then go back to Step 3.

Step 5. All intervals are now marked, so the off-line cost since the last initialization of the $C(i)$'s is at least $m/2$. Go back to Step 2 and reinitialize.

Analysis. Let $T(n)$ be the expected cost of algorithm Randomized-Line for $n-1$ servers on a line of n unit-spaced points for a sequence of requests yielding a minimum off-line cost of $\lceil n/6 \rceil$.

Each application of Step 3 costs the algorithm at most an expected $n + T(3m)$: n for moving to the chosen interval and $T(3m)$ for the cost inside that interval. Notice that the cost function $C(i)$ for $i \in I_{j-1} \cup I_j \cup I_{j+1}$ may not yield values as high as those computed by the recursive application. The reason is the "edge effects": in the recursive application, the off-line costs of the endpoints of the range are not constrained by the costs of points outside it. However, the range is large enough so that *for all i in the middle region I_j*, the cost $C(i)$ is at least that computed by the recursive application as long as $C(i) \le \lceil m/2 \rceil$.

Each application of Step 3 results in half of the unmarked intervals becoming marked in Step 4 on average since the central interval I_j inhabited by the hole is chosen randomly from the unmarked intervals. Thus after $O(\log n)$ applications of Step 3, with high probability all intervals have been marked. Once we repeat Steps 2–5 $\lceil n/(3m) \rceil$ times, the off-line cost has increased by at least $n/6$.

Therefore, we get the following recurrence:

$$\begin{aligned} T(n) &\le \lceil n/3m \rceil \left[O(\log n)[T(3m) + n] \right] \quad \text{for, say, } n \ge 16 \\ &\le \frac{c(n \log n)}{m} [T(3m) + n] \quad \text{for some constant } c. \end{aligned}$$

Substituting $m = \lceil n/2^{\sqrt{\log n \log\log n}} \rceil$ yields

$$T(n) = O\left(n \cdot 2^{3\sqrt{\log n \log\log n}} \right). \qquad \square$$

Acknowledgments. We thank Alok Aggarwal, Allan Borodin, Don Coppersmith, Leo Guibas, Sandy Irani, Ming Kao, Howard Karloff, Samir Khuller, and Yishay Mansour for comments and suggestions. We also thank the referees for many valuable comments.

REFERENCES

[1] R. A. Baeza-Yates, J. C. Culberson, and G. J. E. Rawlins, *Searching in the plane*, Inform. and Comput., 106 (1993), pp. 234–252.

[2] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan, *On-line navigation in a room*, J. Algorithms, 17 (1994), pp. 319–341.

[3] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, *On the power of randomization in on-line algorithms*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, ACM, New York, 1990, pp. 379–388.

[4] A. Blum, A. Borodin, D. Foster, H. J. Karloff, Y. Mansour, P. Raghavan, M. Saks, and B. Schieber, *Randomized on-line algorithms for graph closures*, unpublished manuscript, 1990.

[5] M. Blum and D. Kozen, *On the power of the compass (or, why mazes are easier to search than graphs)*, in Proc. 19th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1978, pp. 132–142.

[6] A. Borodin, N. Linial, and M. Saks, *An optimal online algorithm for metrical task systems*, in Proc. 19th Annual ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 373–382.

[7] S. J. Buckley, *Planning compliant motion strategies*, Internat. J. Robotics Res., 8 (1989), pp. 28–44.

[8] A. K. Chandra, P. Raghavan, W. L. Ruzzo, R. Smolensky, and P. Tiwari, *The electrical resistance of a graph captures its commute and cover times*, in Proc. 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 574–586.

[9] L. Cheng and J. D. McKendrick, *Autonomous knowledge based navigation in an unknown two dimensional environment with convex polygonal obstacles*, Proc. Internat. Soc. Opt. Engrg., 1095 (1989), pp. 752–759.

[10] M. Chrobak, H. J. Karloff, T. Payne, and S. Vishwanathan, *New results on server problems*, in Proc. 1st ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1990, pp. 291–300.

[11] K. L. Clarkson, *Approximation algorithms for shortest path motion planning*, in Proc. 19th ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 56–65.

[12] E. G. Coffman and E. N. Gilbert, *Paths through a maze of rectangles*, Networks, 22 (1992), pp. 349–367.

[13] M. Daily, J. Harris, D. Keirsey, D. Olin, D. Payton, K. Reiser, J. Rosenblatt, D. Tseng, and V. Wong, *Autonomous cross-country navigation with the ALV*, in Proc. IEEE International Conference on Robotics and Automation, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 718–726.

[14] P. Eades, X. Lin, and N. C. Wormald, *Performance guarantees for motion planning with temporal uncertainty*, Austral. Comput. J., 25 (1993), pp. 21–28.

[15] J. Hallam, P. Forster, and J. Howe, *Map free localization in a partially moving* 3*-D world: The Edinburgh feature based navigator*, in Proc. International Conference on Intelligent Autonomous Systems, Vol. 2, IOS Press, Burke, VA, 1989, pp. 726–736.

[16] B. Kalyanasundaram and K. Pruhs, *A competitive analysis of algorithms for searching unknown scenes*, Comput. Geom., 3 (1993), pp. 139–155.

[17] H. J. Karloff, Y. Rabani, and Y. Ravid, *Lower bounds for randomized server algorithms*, in Proc. 23rd ACM Symposium on Theory of Computing, ACM, New York, 1991, pp. 278–288.

[18] R. Klein, *Walking an unknown street with bounded detour*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 303–313.

[19] V. Lumelsky, *Algorithmic issues of sensor-based robot motion planning*, in Proc. 26th IEEE Conference on Decision and Control, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 1796–1801.

[20] V. Lumelsky, *Algorithmic and complexity issues of robot motion in an uncertain environment*, J. Complexity, 3 (1987), pp. 146–182.

[21] V. J. Lumelsky and A. A. Stepanov, *Dynamic path planning for a mobile automaton with limited information on the environment*, IEEE Trans. Automatic Control, AC-31 (1986), pp. 1058–1063.

[22] A. Mei and Y. Igarashi, *An efficient strategy for robot navigation in unknown environment*, Inform. Process. Lett., 52 (1994), pp. 51–56.

[23] M. S. Manasse, L. A. McGeoch, and D. D. Sleator, *Competitive algorithms for on-line problems*, J. Algorithms, 11 (1990), pp. 208–230.

[24] H. Moravec, *The Stanford cart and the CMU rover*, Proc. IEEE, 71 (1983), pp. 872–874.

[25] B. J. Oomen, S. S. Iyengar, N. S. V. Rao, and R. L. Kashyap, *Robot navigation in unknown terrains using learned visibility graphs, part* I*: The disjoint convex obstacle case*, IEEE J. Robotics Automation, 3 (1987), pp. 672–681.

[26] C. H. Papadimitriou and M. Yannakakis, *Shortest paths without a map*, in Proc. 16th International Colloquium on Automata, Languages, and Programming, Springer-Verlag, Berlin, 1989, pp. 610–620.

[27] N. S. V. Rao, *Algorithmic framework for learned robot navigation in unknown terrains*, IEEE Trans. Comput., 22 (1989), pp. 37–43.

[28] N. S. V. Rao, S. S. Iyengar, and G. deSaussure, *The visit problem: visibility graph based solution*, in Proc. IEEE International Conference on Robotics and Automation, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 1650–1655.

[29] N. S. V. Rao, S. S. Iyengar, B. J. Oomen, and R. L. Kashyap, *On terrain model acquisition by a point robot amid polyhedral obstacles*, IEEE J. Robotics Automation, 4 (1988), pp. 450–455.

[30] C. N. Shen and G. Nagy, *Autonomous navigation to provide long distance surface traverses for Mars rover sample return mission*, in Proc. IEEE International Symposium on Intelligent Control, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 362–367.

[31] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. Assoc. Comput. Mach., 28 (1985), pp. 202–208.

[32] C.-K. Yap, *Algorithmic motion planning*, in Advances in Robotics, J. T. Schwartz and C. K. Yap, eds., Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 95–144.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 138–152, February 1997

008

FINITE MONOIDS: FROM WORD TO CIRCUIT EVALUATION*

MARTIN BEAUDRY†, PIERRE MCKENZIE‡, PIERRE PÉLADEAU§, AND DENIS THÉRIEN¶

Abstract. The problem of evaluating a circuit whose wires carry values from a finite monoid M and whose gates perform the monoid operation provides a meaningful generalization to the well-studied problem of evaluating a *word* over M. Evaluating words over monoids is closely tied to the fine structure of the complexity class NC^1, and in this paper analogous ties between evaluating *circuits* over monoids and the structure of the complexity class P are exhibited. It is shown that circuit evaluation in the case of any nonsolvable monoid is P complete, while circuits over solvable monoids can be evaluated in $DET \subseteq NC^2$. Then the case of aperiodic monoids is completely elucidated: their circuit evaluation problems are either in AC^0 or L- or NL-complete, depending on the precise algebraic properties of the monoids. Finally, it is shown that the evaluation of circuits over the cyclic group $\mathbb{Z}_q$ for fixed $q \geq 2$ is complete for the logspace counting class *co*-MOD_qL, that the problem for p-groups (p a prime) is complete for MOD_pL, and that the more general case of nilpotent groups of exponent q belongs to the Boolean closure of MOD_qL.

Key words. complexity theory, automata and formal languages, monoids

AMS subject classifications. 68Q15, 68Q25, 20M35

PII. S0097539793249530

1. Introduction.

Fix a finite monoid M, that is, a finite set with an associative binary operation $*$ for which an element of the set acts as an identity. Define a *circuit over monoid* M as a circuit whose inputs are elements of M and whose gates perform the operation $*$.

The circuit evaluation problem over the monoid M, denoted CEP(M), is that of determining the monoid element computed at a designated output gate in a circuit over M prescribed on input.

CEP(M) can be thought of as a generalization of the *word problem* over M, in which a sequence $m_1, m_2, \ldots, m_k$ of elements of M is given and the task is to evaluate $m_1 * m_2 * \cdots * m_k$. CEP(M) is thus particularly interesting in view of the role played by word problems over monoids in the algebraic characterization of NC^1 and its subclasses [4, 7, 6, 14, 28]. In this paper, we investigate the complexity of CEP(M) for various M. Our results suggest that in a strong sense, circuit evaluation problems

*Received by the editors May 24, 1993; accepted for publication (in revised form) April 21, 1995. This paper revises and extends the abstract "Circuits with monoidal gates" that was presented by the first, second, and third authors at the 1993 Symposium on Theoretical Aspects of Computer Science and appeared in *Proc. 10th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Comput. Sci. 665, Springer-Verlag, Berlin, New York, Heidelberg, 1993, pp. 555–565 [11].

http://www.siam.org/journals/sicomp/26-1/24953.html

†Département de Mathématiques et d'Informatique, Université de Sherbrooke, Sherbrooke, PQ J1K 2R1, Canada (beaudry@dmi.usherb.ca). The research of this author was supported by NSERC grant OGP0089786 and FCAR grants 92-NC-0608 and 91-ER-0642.

‡Département d'Informatique et Recherche Opérationnelle, Université de Montréal, C. P. 6128, Succursale Centre Ville, Montréal, PQ H3C 3J7, Canada (mckenzie@iro.umontreal.ca). The research of this author was supported by NSERC grant OGP0009979 and FCAR grant 91-ER-0642.

§LITP, Université de Paris IV, 4 Place Jussieu, 75253 Paris, France. Current address: Booz, Allen, and Hamilton, 112 Avenue Kléber, 75116 Paris, France.

¶School of Computer Science, McGill University, 3480 University Street, Montréal, PQ H3A 2A7, Canada (denis@cs.mcgill.ca). The research of this author was supported by the NSERC and the FCAR.

over monoids are to the complexity class P what word problems over monoids are to the complexity class NC^1.

Boolean circuit evaluation over the basis $\{\vee, \wedge, \neg\}$ or over the singleton basis {NAND} is a well-known P-complete problem [27]. Viewing CEP(M) as a circuit evaluation over a "basis" consisting of a single *associative* operator, one might expect CEP(M) for any M to be markedly easier than circuit evaluation in the Boolean case. This is *not* so, however. We prove that whenever M is a nonsolvable monoid, i.e., includes a subset which forms a nonsolvable group, CEP(M) is P-complete. Such an increase in complexity associated with nonsolvability of the monoid has already been encountered in another context: the *word* problem over a fixed monoid M is complete for the complexity class NC^1 if M is nonsolvable [4] and believed to be contained in a proper subclass of NC^1 otherwise [7].

What happens, then, when M is solvable, i.e., contains no subset which forms a nonsolvable group? The complexity of CEP(M) then apparently drops significantly. We prove that CEP(M) then belongs to DET, a subclass of NC^2 defined as the closure of the integer determinant problem under NC^1 (Turing) reducibility [20]. This proof relies on a nondeterministic algorithm, which is an interesting application of language-theoretical concepts described in section 3, and on the fact, discussed and proved in section 2, that the logspace counting class MOD_qL appropriately relativized to a DET oracle is contained in DET. (Here MOD_qL is the set of languages Y for which some nondeterministic logspace Turing machine N satisfies the property that $x \in Y$ iff the number of accepting paths of N on input x is not divisible by q [18].)

In the absence of groups altogether, we are able to completely elucidate the complexity of CEP(M), assuming that the classes L and NL are distinct. Indeed let M be an aperiodic monoid, i.e., a monoid none of whose subsets forms a nontrivial group. If M is commutative and idempotent, then CEP(M) $\in AC^0$. Otherwise, if M contains at least one nonidempotent element and every idempotent of M commutes with each element of M, then CEP(M) is L-complete. Otherwise, CEP(M) is NL-complete. The different cases coincide with *varieties of monoids*, namely $\mathbf{J_1}$, the variety of all commutative and idempotent monoids, $\mathbf{M_{nil}}$, the idempotent central monoids, and the variety $\mathbf{A}$ of all aperiodic monoids. (Terms used in this paragraph are defined in a later section.)

Further, write $\mathbb{Z}_q$ for the cyclic group of order $q \geq 2$. We prove that CEP($\mathbb{Z}_q$) is *co*-MOD_qL complete. Moreover, if G is any p-group (i.e., a group of order a power of the prime p), then CEP(G) is MOD_pL-complete. Finally, in the general case of a nilpotent group G of exponent q, CEP(G) belongs to the Boolean closure of MOD_qL.

Table 1 summarizes our results; some of the notation and terminology used in this table is explained in sections 2 and 3.

Section 2 gives preliminaries and proves that MOD_qL relativized to DET is contained in DET. Section 3 presents the minimal background on monoids required for what follows. The main section is section 4, which discusses the complexity of CEP(M). Section 5 concludes with a discussion and suggestions for further work.

2. Preliminaries and definitions. Fix a finite monoid M and let "product" refer to its associative binary operation. We define the unconnected circuit evaluation problem UCEP(M) and its connected variant CEP(M).

DEFINITION. *Problem UCEP(M):*

Given *an element $x \in M$ and a directed acyclic graph of arbitrary indegree and arbitrary outdegree in which each in-degree-0 (input) node is labeled with an element*

TABLE 1.1
Our results on the complexities of CEP(M) and UCEP(M) as the algebraic properties of M vary. $\langle MOD_qL\rangle$ denotes the Boolean closure of the class MOD_qL.

Monoid M	CEP(M)	UCEP(M)
Nonsolvable Solvable	P-complete $\in DET$	P-complete $\in DET$
APERIODIC CASE: $M \in \mathbf{J_1}$ $M \in \mathbf{M_{nil}} - \mathbf{J_1}$ $M \notin \mathbf{M_{nil}}$	 $\in AC^0$ L-complete NL-complete	 NL-complete NL-complete NL-complete
SELECT GROUP CASES: Nilpotent, exponent q Cyclic group Z_q p-group, p-prime	 $\in \langle MOD_qL\rangle$ co-MOD_qL-complete MOD_pL-complete	 $\in \langle MOD_qL\rangle$ co-MOD_qL-complete MOD_pL-complete

from M and some out-degree-0 node is designated as the output node.

Determine *whether x is the value computed at the output node, where computation proceeds by having each internal node broadcast along its "output wires," the product of the sequence of monoid elements received along its "input wires."*

DEFINITION. *Problem* CEP(M)*: restricted variant of* UCEP(M) *in which there is a unique out-degree-0 node, accessible from each and every other node in the graph.*

Clearly an upper bound on the complexity of UCEP(M) applies to CEP(M) as well, and a hardness result on the complexity of CEP(M) applies to UCEP(M).

A circuit is prescribed on input by a variant of its direct connection language (see [19]). Since the operation performed at a gate is in general noncommutative, we add the condition that the tuples encoding the connections between a node and its inputs be numbered in a manner consistent with the order of evaluation at that node.

We assume familiarity with NP, P, NL, and L and with AC^0 and NC^k in their uniform settings. The precise choice of uniformity will not matter, but for definiteness we adopt $DLOGTIME$ uniformity for AC^0 [5] and U_{E^*} uniformity for NC^k [30, 20]. The class DET is defined as the closure of the integer determinant problem under NC^1 (Turing) reducibility [20]:

$$AC^0 \subset NC^1 \subseteq L \subseteq NL \subseteq DET \subseteq NC^2 \subseteq P$$

In general, we let the context distinguish between classes of languages and classes of functions, for example, between L and the class FL of functions computed in logspace. Throughout this paper, we will say that language A "NC^1 reduces" to language B, written $A \leq_m^{NC^1} B$, iff A "many–one" reduces to B via an NC^1-computable function.

Let $t \geq 0$ and $q \geq 1$ be natural numbers. Define the $\theta_{t,q}$ relation on $\mathbb{N}$ as follows:

$$i\theta_{t,q}j \iff [(i = j) \vee ((i \geq t) \wedge (j \geq t))] \wedge [i \equiv j (\mathrm{mod}\ q)].$$

The $\theta_{t,q}$ relations have algebraic significance because they are the only equivalence relations on $\mathbb{N}$ which are in fact finite index congruences (see [35]). Let $MOD_{t,q}L$ denote the set of languages Y such that for some $i \in \mathbb{N}$ and some nondeterministic logspace Turing machine M the following holds for each input x:

$$x \notin Y \iff i\theta_{t,q}|\mathrm{ACCEPT}(M, x)|,$$

where ACCEPT(M, x) denotes the set of accepting paths of M on input x. Observe that $MOD_{0,q}L$ is exactly the class MOD_qL as defined in [18]; this is because not just zero but any other single congruence class (mod q) could have been used in the definition of MOD_qL [18] without affecting the complexity class (see [13]). Observe further that $MOD_{1,1}L = NL \cup co\text{-}NL = NL$ [34, 24].

In order to analyze the complexity of problems CEP(M) in the case of solvable monoids, it will be convenient to first obtain a structural complexity result which is of independent interest (Theorem 2.1 below). This result deals with relativized counting classes and requires a suitable definition of relativized space. Here we borrow the definition proposed by Ruzzo, Simon, and Tompa [31].

DEFINITION (space-bounded oracle machine model). *Define M^A as the language recognized by a (possibly nondeterministic) machine M using oracle A in the following controlled manner: M has a write-only query tape not subject to a space bound and operates deterministically from the time some symbol is written onto the tape until the time the next oracle query is made, after which the query tape is erased.*

As pointed out in [31, Lemma 7], a language Y is equal to M^A for a logspace bounded machine M iff Y is logspace Turing reducible to some B (using queries which *are* subject to the space bound) and B is many–one reducible to A via a logspace transducer also having access to M's input.

When C is a complexity class and D is a set of languages, C^D represents the set of languages M^A such that M is a machine obeying the resource bounds associated with C and $A \in D$.

THEOREM 2.1. *$(MOD_{t,q}L)^{DET} \subseteq DET$ and $(co\text{-}MOD_{t,q}L)^{DET} \subseteq DET$ for any $t \geq 0$ and any $q \geq 1$.*

Proof. We do the proof for $(MOD_{t,q}L)^{DET}$. A desirable consequence of the above definition of relativized space is that (even) a nondeterministic logspace machine cannot make more than a polynomial number of distinct oracle queries in the course of its computation on any given input. This is because each query is the image, under transduction, of a logspace bounded "prequery." The simulation of a $MOD_{t,q}L$ machine M with oracle $A \in DET$ on input x will thus

1. list all possible queries,
2. compute all query answers and insert them into the list,
3. compute $|\text{ACCEPT}(M^A, x)|$, and
4. reject iff $i\theta_{t,q}|\text{ACCEPT}(M^A, x)|$, where i is the target number specified with M.

The first step is a logspace computation, which an NC^1 circuit with DET oracle gates can simulate because $L \subseteq DET$ [20]. The second step involves another layer of NC^1 circuitry with DET oracle gates because $A \in DET$. The fourth step is a pure NC^1 computation involving binary subtraction and division by a constant.

Now consider step three. It is easy to simulate machine M, which uses A as oracle, with a logspace bounded machine M' with no oracle, provided we input to M' the list computed in step two together with x. This simulation is easily made to preserve the number of accepting paths. But then computing the number of accepting paths of M' from its input is a $\#L$ computation. Since $\#L \subseteq DET$ [1], step three can also be carried out by an NC^1 circuit layer with DET oracle gates. □

An argument similar to the above is used in [18] to prove the (probably) weaker statement that $L^{\#L} \subseteq DET$.

In section 4, we will freely apply Theorem 2.1 to the situation in which a $MOD_{t,q}L$ machine queries *several* oracles $L_0, L_1, \ldots, L_s \in DET$. The standard technical justifi-

cation for this is the obvious fact that if $A \in DET$ and $B \in DET$ then $A \uplus B \in DET$, where $A \uplus B$ is the *disjoint union* of A and B defined as $\{w0 : w \in A\} \cup \{w1 : w \in B\}$ (see [3]). Thus, instead of querying L_i directly, a $MOD_{t,q}L$ machine can query the oracle $(\cdots((L_0 \uplus L_1) \uplus L_2) \cdots \uplus L_s)$ and obtain the desired answer. Since this holds for each $i, 0 \leq i \leq s$, the single oracle $(\cdots((L_0 \uplus L_1) \uplus L_2) \cdots \uplus L_s) \in DET$ suffices to answer all L_i queries.

3. Background on monoids, languages, and varieties. Recall that a monoid is a set equipped with an associative binary operation and an identity for this operation. We will use M to denote both the monoid and its underlying set and represent the operation as a concatenation (i.e., the product of $a \in M$ and $b \in M$ will be denoted ab).

Let A be a finite set or *alphabet*. We write A^* for the *free monoid* over A with concatenation as operation and the empty word as identity. Given a monoid M, let $\phi : M^* \to M$ be the canonical morphism. We will call a *word problem* of M any set or *language* of the form $Q\phi^{-1}$, where $Q \subseteq M$.

We write $\mathbb{Z}_q$ for the cyclic group modulo q and $C_{t,q}$ for the monoid performing addition modulo $\theta_{t,q}$ on the set $\{0, 1, \ldots, t + q - 1\}$. In particular, the operation of monoid $C_{1,1} = \{0, 1\}$ corresponds to the Boolean OR.

The *reverse* of a monoid M has the same underlying set as M but its operation is read in reverse order, i.e., evaluating abc in the reverse of M is equivalent to evaluating cba in M.

The usual notations for the monoid identity element and the universally absorbing element, when it is present, are 1 and 0, respectively. We depart from this convention and use 0 for the identity, however, when we speak of the $\mathbb{Z}_q$ and $C_{t,q}$, as above.

It is extremely useful to classify monoids according to their algebraic complexity, and the tool to do this is the *variety*. A variety of monoids is a set of monoids which is closed under *division* (the operation of taking a homomorphic image of a submonoid) and finite direct products. See [22] and [29] for references on this subject. Notice that the natural ordering of monoids induced by division has bearing on the relative complexities of the CEP problems: indeed if N divides M, then clearly CEP(N) NC^1 reduces to CEP(M).

Varieties of monoids explicitly mentioned in this paper are the set of all nilpotent groups (i.e., direct products of groups of prime power order); the set of all *solvable* monoids (i.e., from which the only simple groups that can be obtained are cyclic); the set of all *aperiodic* (i.e., group-free) monoids, which is denoted by $\mathbf{A}$; the smallest nontrivial aperiodic variety, denoted $\mathbf{J_1}$, which consists of all idempotent (i.e., satisfying $x^2 = x$ for each x) and commutative monoids; its superset the variety $\mathbf{J}$ of *J-trivial* monoids, consisting of all monoids M such that for each $m, n \in M$ if $MmM = MnM$ then $m = n$ (see [29]); the variety $\mathbf{M_{nil}}$ of the *idempotent central* monoids, which contains $\mathbf{J_1}$ and is contained in $\mathbf{J}$ and consists of those monoids M such that for all $e, m \in M$ such that $e^2 = e$, $em = me$ [33].

Each of the varieties $\mathbf{J_1}$, $\mathbf{J}$, and $\mathbf{M_{nil}}$ has the property that there is a finite set of *minimal* aperiodic monoids outside it, that is, a set of aperiodic monoids such that any monoid outside of the variety is divided by at least one of the monoids in the minimal set. For example, $T_2 = \{1, a, a^2\}$ is the smallest J-trivial monoid outside of $\mathbf{J_1}$. The monoid $R_1 = \{1, a, b\}$ with operation satisfying $aa = ab = a$ and $ba = bb = b$ and its reverse L_1, monoid $BA_2 = \{1, a, b, ab, ba, 0\}$ with operation satisfying $aa = bb = 0$, $aba = a$, and $bab = b$, and monoid $BH_2 = \{1, a, b, ab, ba, 0\}$ with operation satisfying

$aa = aba = a$, $bb = 0$, and $bab = b$ (each of BA_2 and BH_2 is equal to its reverse), are the minimal aperiodic monoids which are not J-trivial. (Also, $C_{1,1}$ is the minimal nontrivial aperiodic monoid.) To our knowledge the minimal monoids outside of $\mathbf{M_{nil}}$ had not been identified until now.

Let M_{ba^*} be the syntactic monoid of the language ba^* over the alphabet $\{a, b\}$. This monoid has four elements $\{1, a, b, 0\}$ with products $aa = a$, $ba = b$, and $ab = bb = 0$.

LEMMA 3.1. *R_1, M_{ba^*}, and their reverses, along with BA_2 and BH_2, are the minimal aperiodic monoids outside of $\mathbf{M_{nil}}$.*

Proof. If M is not J-trivial, then it is divided by R_1 or its reverse, by BA_2, or by BH_2. Thus we only need to show that M_{ba^*} and its reverse are the minimal J-trivial monoids outside of $\mathbf{M_{nil}}$.

Let M be a J-trivial monoid outside of $\mathbf{M_{nil}}$. There are therefore e, $m \in M$ such that $e^2 = e$ and $em \neq me$. Let $\alpha = em$ and $\beta = me$. We first show that monoid M satisfies the property

$$(^*) \qquad (e\beta \neq \beta e \wedge \forall k \geq 2 : \beta^k \neq \beta) \vee (e\alpha \neq \alpha e \wedge \forall k \geq 2 : \alpha^k \neq \alpha),$$

from which the lemma will be proved. Denoting by $f \leq_J g$ the relation $MfM \subseteq MgM$, observe that if $(e\beta \neq \beta e \wedge \forall k \geq 2 : \beta^k \neq \beta)$ does not hold, then $\beta \leq_J \alpha$, and similarly $\alpha \leq_J \beta$ if $(e\alpha \neq \alpha e \wedge \forall k \geq 2 : \alpha^k \neq \alpha)$ is false. Since M is J-trivial and $\alpha \neq \beta$, both $\beta \leq_J \alpha$ and $\alpha \leq_J \beta$ cannot be true. This proves $(^*)$.

To conclude the lemma from $(^*)$, we distinguish two cases. If $(e\beta \neq \beta e \wedge \forall k \geq 2 : \beta^k \neq \beta)$ holds, then let M' be the monoid generated by e and β. It is easy to see that $M' = \{1, e, \beta\} \cup \{\beta^k : k \geq 2\} \cup \{e\beta^k : k \geq 1\}$. Let the function $\phi : M' \to M_{ba^*}$ be defined by $\phi(1) = 1$, $\phi(e) = a$, $\phi(\beta) = b$, and $\phi(\beta^{k+1}) = \phi(e\beta^k) = 0$ for any $k \geq 1$. The definition of ϕ is unambiguous. Indeed, note that 1, e, and β are distinct. Next, since M is J-trivial, $\beta \leq_J e$ implies $e \not\leq_J \beta$; hence for $k \geq 1, \beta^{k+1}$ and $e\beta^k$ are different from 1 and e. Meanwhile, $\beta^{k+1} \neq \beta$ by assumption. Finally, assuming that $e\beta^k = \beta$, we have $e\beta = e\beta^k = e\beta^k e = \beta e$, which contradicts the assumption. Verifying that ϕ is a surjective morphism is a standard exercise. Therefore, monoid M_{ba^*} divides M.

The other case where $(e\alpha \neq \alpha e \wedge \forall k \geq 2 : \alpha^k \neq \alpha)$ is treated dually and yields the conclusion that the reverse of M_{ba^*} divides M. □

Another way to look at monoids is through the languages they recognize (via morphisms). A language $Y \subseteq A^*$ is *recognized* by a monoid M iff Y is the inverse image of a subset of M under some homomorphism from A^* to M. The *syntactic monoid* of a language $Y \subseteq A^*$ is the smallest monoid (in the ordering induced by monoid division) which recognizes Y. Straubing [33] identified the languages recognized by the idempotent central monoids.

THEOREM 3.2. *A language $Y \subseteq A^*$ is recognized by a monoid in $\mathbf{M_{nil}}$ iff it is a Boolean combination of languages of the form $B^*a_1B^*a_2 \cdots B^*a_kB^*$ where $a_1, \ldots, a_k \in A$ and $B = A - \{a_1, \ldots, a_k\}$.*

Note that in Theorem 3.2 the a_i are not assumed to be distinct.

At the other end of the spectrum, Straubing [32] and Thérien [35] have developed a useful parametrization of the languages recognized by solvable monoids. Recall the $\theta_{t,q}$ congruence defined in section 2. If $L_0, \ldots, L_s$ are languages over A^* and $a_1, \ldots, a_s \in A$, we denote by $[L_0, a_1, L_1, \ldots, a_s, L_s]_{i;t,q}$ the set of those words $w \in A^*$ for which the number of factorizations $w = u_0a_1u_1 \cdots a_su_s$, $u_j \in L_j$, $0 \leq j \leq s$ is congruent to i with threshold t and period q. Also, if $\mathcal{L}$ denotes a class of languages, let

$\langle\mathcal{L}\rangle$ denote its Boolean closure. We take from [35] the following two parametrizations of the languages recognized by solvable monoids.

THEOREM 3.3. *A language $L \subseteq A^*$ has a syntactic monoid which is finite and solvable iff $L \in \mathcal{M}_{t,q}^k$, where $\mathcal{M}_{t,q}^0 = \langle A^* \rangle$ and for $k \geq 1$:*

$$\mathcal{M}_{t,q}^k = \langle \{ [L_0, a, L_1]_{i;t,q} : L_0, L_1 \in \mathcal{M}_{t,q}^{k-1}, i \in \mathbb{N} \} \rangle.$$

THEOREM 3.4. *A language $L \subseteq A^*$ has a syntactic monoid which is finite and solvable iff $L \in \mathcal{N}_{t,q}^k$, where $\mathcal{N}_{t,q}^0 = \langle A^* \rangle$ and for $k \geq 1$:*

$$\mathcal{N}_{t,q}^k = \langle \{ [L_0, a_1, L_1, \ldots, a_s, L_s]_{i;t,q} : L_0, \ldots, L_s \in \mathcal{N}_{t,q}^{k-1}, i, s \in \mathbb{N} \} \rangle.$$

If we choose to work in the $\mathcal{M}_{t,q}^k$ hierarchies of languages, membership of x in language L is determined by counting (with respect to $\theta_{t,q}$) those occurrences of the character a that occur in the context of a prefix in L_0 and a suffix in L_1. In the $\mathcal{N}_{t,q}^k$ hierarchies, we would instead count occurrences of subwords $a_1, \ldots, a_s$ within contexts $L_0, \ldots, L_s$. Contrary to [7], where the former hierarchies were used, we find here that the $\mathcal{N}_{t,q}^k$ hierarchies seem more appropriate to the study of problem CEP.

The following results are taken from [35] and [36]. They describe some of the main special cases of solvable monoids.

THEOREM 3.5. *A language Y is recognizable by a finite aperiodic monoid iff Y is in $\mathcal{N}_{t,1}^k$ for some $k, t \geq 0$ iff Y is in $\mathcal{N}_{1,1}^k$ for some $k \geq 0$.*

THEOREM 3.6. *A language Y is recognizable by a finite solvable group iff Y is in $\mathcal{N}_{0,q}^k$ for some $k \geq 0$, $q \geq 1$.*

THEOREM 3.7. *A language Y is recognizable by a finite nilpotent group iff Y is in $\mathcal{N}_{0,q}^1$ for some $q \geq 1$.*

THEOREM 3.8. *A language Y is recognizable by a finite solvable monoid iff Y is in $\mathcal{N}_{t,q}^k$ for some $k, t \geq 0, q \geq 1$ iff Y is in $\mathcal{N}_{1,q}^k$ for some $k \geq 0, q \geq 1$.*

A byproduct of these theorems is that given a finite solvable monoid M, one can find parameters k, t, and q such that every word problem of M lies in $\mathcal{N}_{t,q}^k$. (Recall that word problems and the canonical morphism ϕ were defined at the beginning of this section.) In fact, given any $Q \subseteq M$, one can actually compute an expression that represents $Q\phi^{-1}$ involving M^*, the characters of M, the Boolean operations, and the operation $L_0, a_1, L_1, \ldots, a_s, L_s \to [L_0, a_1, L_1, \ldots, a_s, L_1]_{i;t,q}$. Notice that parameter q is a multiple of the exponents of all maximal subgroups of M (of M itself if M is a group).

4. Complexity of circuit problems. This section contains our upper bounds and hardness results on the complexities of UCEP(M) and CEP(M). Except when M is aperiodic (see subsection 4.2), the two problems have identical complexities.

4.1. The impact of solvability. In this subsection we exhibit a significant gap in the complexity of CEP(M), depending on whether M is solvable or not (within the hypothesis that $NC^2 \neq P$). We prove that CEP(M) and UCEP(M) are P-complete under $\leq_m^{NC^1}$ reducibility if M is nonsolvable and belong to DET, hence to NC^2, otherwise. Then we investigate the special cases of cyclic groups, p-groups, and nilpotent groups, and we prove an NL upper bound in the aperiodic case.

Throughout this subsection, each result obtained applies equally well to CEP(M) and to UCEP(M), even when this is not stated explicitly.

LEMMA 4.1. *If M contains a nonsolvable group, then* CEP(M) *is P-complete.*

Proof. Membership in P is obvious. To prove hardness, as shown below, we merely adapt Barrington's simulation of the Boolean operations AND, OR, and NOT

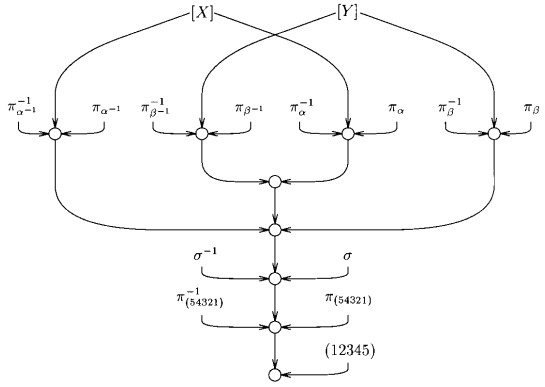


FIG. 4.1. *Circuit over $\mathcal{A}_5$ simulating X* NAND *Y.*

by products within the alternating group $\mathcal{A}_5$ [4, Theorem 1]. The generalization to the case of any nonsolvable group follows as in [4, Theorem 5]. (Alternatively, Zalcstein pointed out that a construction related to Barrington's had been investigated independently by Bergman [15] and others.)

It suffices to explain how to simulate a NAND gate, because evaluating a Boolean circuit with NAND gates is clearly P-hard [27]. Let $\alpha, \beta \in \mathcal{A}_5$ be 5 cycles such that $\alpha^{-1}\beta^{-1}\alpha\beta$ is also a 5 cycle. Let $\sigma \in \mathcal{A}_5$ satisfy

$$\sigma^{-1}\alpha^{-1}\beta^{-1}\alpha\beta\sigma = (12345).$$

Furthermore, for any 5 cycle $\gamma \in \mathcal{A}_5$, write π_γ for the unique element of $\mathcal{A}_5$ satisfying $\pi_\gamma^{-1}(12345)\pi_\gamma = \gamma$. Then we simulate a NAND gate with Boolean inputs X and Y using the constant-fan-in, constant-depth subcircuit given by Figure 1.

Let $[X]$ denote the subcircuit over $\mathcal{A}_5$ which encodes the work of gate X. Assuming inductively that $[X]$ evaluates to the identity of $\mathcal{A}_5$ (resp., (12345)) iff X outputs the Boolean value 0 (resp., 1), and similarly for $[Y]$, then the subcircuit on Figure 1 evaluates to the identity (resp., (12345)) iff X NAND $Y = 0$ (resp., X NAND $Y = 1$). □

Remark. The circuit over $\mathcal{A}_5$ built in the proof of hardness has gates of bounded fan-in. Recall however that the definition of problem CEP(M) allows gates with arbitrary fan-in and that associativity can be used to collapse a depth d instance of CEP(M) into a depth 1 instance consisting of the inputs and of one gate with fan-in $2^{O(d)}$. Thus gate fan-in can be traded for circuit depth. Using this idea, it can be shown that problem CEP(M) restricted to depth-$O(\log^k n)$ circuits is complete for NC^{k+1} under logspace reducibility. The proof repeats that of Lemma 4.1; in the NC^{k+1} hardness part, the depth $O(\log^{k+1} n)$, constant fan-in instance of CEP($\mathcal{A}_5$) obtained as an intermediate result, is divided into depth d subcircuits, $d \in O(\log n)$, which are then collapsed into gates of polynomial fanin, reducing the depth to $O(\log^k n)$. For further details we refer the reader to [10], where a technique for an analogous deterministic logspace reduction is described.

THEOREM 4.2. *If M is a solvable monoid then* UCEP(M) *belongs to DET.*

Proof. Consider the circuit in a UCEP(M) instance. The idea behind the algorithm developed below is that a word of M^* can be associated with each node in the circuit: a one-character word for the input gates, otherwise the concatenation of

the words associated with those nodes from which the gate receives its inputs. Thus determining the element of M computed at the output gate g is equivalent to applying M's canonical homomorphism ϕ to the word $W(g)$ associated with g, which in turn amounts to testing whether $W(g)$ belongs to a language recognized by M. This enables us to exploit the hierarchical parametrizations of the languages recognized by a solvable monoid.

Let M be a solvable monoid and $\phi : M^* \to M$ denote its canonical homomorphism. By Theorem 3.8, for each $m \in M$, $m\phi^{-1}$ belongs to $\mathcal{N}_{t_m,q_m}^{k_m}$ for some $k_m, t_m \geq 0$, and $q_m \geq 1$. Setting $k = \max\{k_m\}$, $t = \max\{t_m\}$, and $q = \operatorname{lcm}\{q_m\}$ yields $m\phi^{-1} \in \mathcal{N}_{t,q}^{k}$ for each $m \in M$.

Let g be the output gate specified in the instance of UCEP(M); testing for output g amounts to testing for membership of $W(g)$ in a language $[L_0, a_1, L_1, \ldots, a_s, L_s]_{i;t,q}$ recognized by M. Let M_r with operation $*_r$ be the syntactic monoid of L_r for $0 \leq r \leq s$. Let $B_r \subseteq M_r$ be the image of L_r under an appropriate morphism $\phi_r : M^* \to M_r$. The testing will be done by counting the successful computations of the following nondeterministic algorithm, that is, counting the accepting computations of a nondeterministic logspace Turing machine with oracles for UCEP(M_r), $0 \leq r \leq s$. From Theorem 2.1 and the closure of DET under Boolean operations, it will follow inductively that UCEP(M) belongs to DET. (Remark that the algorithm works on a circuit with in-degree-2 gates, which can be obtained from an arbitrary circuit through a logspace transduction.)

- Let $u_1, \ldots, u_s$ be pointers to nodes of the input circuit over M, each initialized to g.
- Let $b_0, \ldots, b_s$ be Boolean values initialized to false, except for b_0 and for b_s, which are true.
- Let $x_1 = \cdots = x_s = 1$, where each x_r is regarded as an element of monoid M_{r-1}.
- Let $y_1 = \cdots = y_s = 1$, where each y_r is regarded as an element of monoid M_r.
- *Repeat* until each u_r is an input node, or the instance is rejected:
  - nondeterministically pick a child v_r for each u_r (let $v_r = u_r$ if u_r is a leaf);
  - *if* there is an $r < s$ such that $b_r =$ false and v_r and v_{r+1} are the right and left child, respectively, of node $u_r = u_{r+1}$, *then* reject the instance; (*comment: subword characters a_r and a_{r+1} would then be accessed in the wrong order*)
  - *for* $r = 1 \ldots s$ *do*
    - *if* $b_{r-1} =$ true and v_r is the right child of u_r, *then* evaluate the subcircuit rooted at the left child of node u_r as if it were on monoid M_{r-1}, that is, for each $m \in M_{r-1}$, test whether the subcircuit, having each of its input gate labels appropriately translated through morphism ϕ_{r-1} into the corresponding element of M_{r-1}, evaluates to m. Denote by z the value thus obtained. Then let $x_r = x_r *_{r-1} z$.
    - *if* $b_r =$ true and v_r is the left child of u_r, *then* evaluate the subcircuit rooted at the right child of node u_r as if it were on monoid M_r; that is, for each $m \in M_r$, test whether the subcircuit, having each of its input gate labels appropriately translated through morphism ϕ_r into the corresponding element of M_r, evaluates to m. Denote by z the value thus obtained. Then let $y_r = z *_r y_r$.
- *for* $r = 1 \ldots s$ *do*: *If* $b_r =$ false and $v_r \neq v_{r+1}$, *then* let $b_r =$ true.
- *for* $r = 1 \ldots s$ *do*: Let $u_r = v_r$.

- *If* each u_r is an input node carrying value a_r *and* $x_1 \in B_1$ *and* $y_r *_r x_{r+1} \in B_r$ for $1 \le r \le s-1$ *and* $y_s \in B_s$ *and* $b_r = \mathsf{true}$ for $0 \le r \le s$, *then* accept; *else* reject.

The algorithm is a nondeterministic logspace computation with oracle. Here the u_r's and v_r's are pointers used for top-down traversal of the circuit; each path from g to an input node coincides with a character of $W(g)$, and the orderings of the paths and of the characters are consistent. Each Boolean variable b_r indicates whether the paths to characters a_r and a_{r+1} still coincide ($b_r = \mathsf{false}$) or have split apart. From the observation that the number α of accepting computations of the algorithm is the number of ways in which circuit inputs labeled $a_1, \ldots, a_s$ can appear in the word $W(g)$ within the appropriate contexts in $L_0, \ldots, L_s$, we obtain that $W(g) \in [L_0, a_1, L_1, \ldots, a_s, L_s]_{i;t,q}$ iff $i \,\theta_{t,q}\, \alpha$.

The analysis goes by induction on k. The base case $k = 0$ is clear because then $x\phi^{-1} = M^*$ or $x\phi^{-1} = \emptyset$, where x is the target element, and the algorithm can output an answer without looking at its input. Now for the induction step. In general, $x\phi^{-1}$ is a Boolean combination of languages of the kind $[L_0, a_1, L_1, \ldots, a_s, L_s]_{i;t,q}$. Testing for membership in such a language is done by counting accepting computations of our algorithm, which runs in nondeterministic logspace with calls to an oracle for UCEP$(M_0), \ldots,$ UCEP(M_s), but then we are done by applying Theorem 2.1 inductively. □

Intuition might suggest that a preliminary step to the above algorithm would be to compute that portion of the (unconnected) circuit which is connected to the output node. However this is not necessary, for the evaluation problem amounts to counting (context-dependent) paths from the root to certain input nodes, and this is precisely what the algorithm does through nondeterministic descents in the circuit. A good way to see this is to apply the algorithm to the case in which the monoid is a cyclic group of order q: problem UCEP(M) is then equivalent to the mod-q accessibility problem in an acyclic-directed graph (see Lemma 4.6).

We now apply the algorithm to some special cases of solvable monoids.

COROLLARY 4.3. *If M is an aperiodic monoid then* CEP(M) *belongs to NL.*

Proof. By Theorem 3.5, the algorithm deals with a language $x\phi^{-1} \in \mathcal{N}_{1,1}^k$. Hence the counting involved amounts to determining whether the number of accepting computations is zero or at least one. The analysis therefore gives an inductive step complexity in NL^{NL}. The result follows from the fact that $NL^{NL} = NL$ [24, 34]. □

COROLLARY 4.4. *Let M be the cyclic group $\mathbb{Z}_q$. Then problem* CEP(M) *belongs to co-MOD_qL.*

Proof. This is a special case in which the above algorithm deals with a language in $\mathcal{N}_{0,q}^1$, so that the recursive calls require no computation at all. Indeed, an instance of CEP$(\mathbb{Z}_q)$ can be thought of as a circuit with two leaves, carrying inputs 0 and 1; then the value of the output equals the number of paths from the 1 leaf to the output node, counted modulo q. Thus testing whether the circuit evaluates to a given target value reduces to testing whether the number of paths between two given nodes in an acyclic-directed graph is congruent to 0 modulo q; this "zero mod-q accessibility problem" is *co-MOD_qL* complete [18]. □

COROLLARY 4.5. *Let M be a nilpotent group of exponent q for $q \ge 2$. Then problem* CEP(M) *belongs to the Boolean closure of MOD_qL.*

Proof. We reason as in Corollary 4.3, noting that every language recognized by a nilpotent group of exponent q belongs to $\mathcal{N}_{0,q}^1$ by Theorem 3.7. □

The following hardness result completes the picture afforded by the last two corollaries.

LEMMA 4.6. *Let $q \geq 2$. If M contains an element of period q, then* CEP(M) *is hard for co-MOD_qL under $\leq_m^{NC^1}$ reducibility.*

Proof. It suffices to show that CEP($\mathbb{Z}_q$) is hard for *co-MOD_qL*. We reduce from the problem of determining whether the number of paths from u (of in-degree 0) to v (of out-degree 0) in an acyclic-directed graph is a multiple of q [18]. Assign 1 to u and 0 to any other in-degree-0 node. Then for each node $w \neq v$ of out-degree 0, add a gadget connecting w to node v through exactly q distinct paths. The resulting graph viewed as a circuit over $\mathbb{Z}_q$ evaluates to 0 at the unique output gate v iff the number of paths from u to v in the original graph was a multiple of q. □

THEOREM 4.7. *For $q \geq 2$ a fixed integer, problem* CEP($\mathbb{Z}_q$) *is co-MOD_qL-complete under $\leq_m^{NC^1}$ reducibility.* □

In the special case where q is prime, Corollary 4.5 and the fact that $MOD_qL = MOD_{q^n}L$, $n \geq 1$ is closed under Boolean operations [18] combine with Theorem 4.7 to yield a stronger statement.

COROLLARY 4.8. *For $p \geq 2$ a fixed prime integer and any p-group M, problem* CEP(M) *is MOD_pL-complete under $\leq_m^{NC^1}$ reducibility.* □

4.2. The aperiodic case. In this section, we discuss the computational complexity of problems CEP(M) and UCEP(M) when M is aperiodic. In the aperiodic case the complexities of the two variants differ. Indeed, counting in an aperiodic monoid implies distinguishing between "zero" and "at least one," which in the context of circuit evaluation means deciding whether at least one input node carrying a nontrivial value is connected with the output node. Consequently, UCEP(M) is NL-hard for any nontrivial aperiodic monoid M (Proposition 4.14). However the main result in this section is the following striking characterization of the complexities of CEP(M) as M ranges over the set of all finite aperiodic monoids.

THEOREM 4.9. *When M is an aperiodic monoid, exactly three cases arise:*

- *If M belongs to variety $\mathbf{J_1}$, then* CEP(M) *belongs to AC^0 (Lemma* 4.10*).*
- *If M belongs to variety $\mathbf{M_{nil}}$ and is nonidempotent, then* CEP(M) *is L-complete (Lemmas* 4.11 *and* 4.12*).*
- *If M does not belong to $\mathbf{M_{nil}}$, then* CEP(M) *is NL-complete. (Lemma* 4.13 *and Corollary* 4.3*).*

Corollary 4.3 was proved in subsection 4.1; we complete the proof of the theorem with the following four lemmas.

LEMMA 4.10. *If M belongs to $\mathbf{J_1}$, then* CEP(M) *can be solved in AC^0.*

Proof. When the expression for an element of an idempotent and commutative monoid is evaluated, each character involved contributes exactly once. To see this, take an arbitrary expression and first apply commutativity to regroup together all occurrences of the same character. Then, since $a^2 = a$ for all $a \in M$, replace them with a single occurrence. Thus by looking only at the content of the leaves and ignoring the rest of the circuit (thus relying heavily on the hypothesis that the circuit is connected), we obtain for the output a fixed-length expression which can be evaluated using table lookup. □

LEMMA 4.11. *If M is a nonidempotent aperiodic monoid, then* CEP(M) *is L-hard under $\leq_m^{NC^1}$ reducibility.*

Proof. This follows by reduction from the accessibility problem in a directed forest [21] to problem CEP(T_2), where $T_2 = \{1, a, a^2\}$ divides all nonidempotent aperiodic monoids (see section 3). Let an instance of the accessibility problem consist of asking

whether leaf u belongs to the tree rooted at v; let all edges in a tree be oriented toward the root. The reduction consists of first creating two new nodes named v' and w and adding an edge from each child of v to v' and then an edge from each out-degree-0 node to w. We obtain a graph with one root w and with nodes of out-degree at most 1, except for the children of v which have out-degree 2. Next, we label all interior nodes with the operation of T_2, leaf u with element a, and all other leaves with 1. If v can be reached from u, then there are exactly two paths from u to w (through v and v'), so that the circuit evaluates to a^2. Else there is exactly one path, and the output is a. □

LEMMA 4.12. *If M belongs to variety $\mathbf{M_{nil}}$ then problem* CEP(M) *belongs to L.*

Proof. Let $M \in \mathbf{M_{nil}}$. From Theorem 3.2, for each $x \in M$ the language $x\phi^{-1}$ is a Boolean combination of languages $B^*a_1B^*a_2 \cdots B^*a_kB^*$, with $a_1, \ldots, a_k \in M$ and $B = M - \{a_1, \ldots, a_k\}$ (the a_i not necessarily distinct). Using the closure of L under Boolean operations, we evaluate the circuit by checking, for each such language in the Boolean combination for $x\phi^{-1}$, whether the ordered sequence of all paths from the leaves to the root yields a word in the language. Given one such language $B^*a_1B^*a_2 \cdots B^*a_kB^*$, let $A' = \{a'_1, \ldots, a'_{k'}\}$ be the set of characters in $S = \{a_1, \ldots, a_k\}$. For each character a'_i, count the number of paths from the leaves labeled a'_i to the root; if it differs from the number of occurrences of a'_i in S, reject the instance. Otherwise, there are m_i paths from the leaves labeled a'_i to the root, and we must next verify that all $\sum m_i = k$ paths are combined in the appropriate order (unless M is commutative, in which case the algorithm stops here). Create a fixed-length register for each pair (r, s) of those paths, with possible values first (path r comes before s in the lexicographic order of root-leaf paths), second (r comes after s), and neither (ordering unknown). Initialize all registers to neither. Then, start every path at the appropriate leaf and advance along them toward the root, synchronously, one node at a time. If the paths are of different lengths, then start the shorter ones with an appropriate delay in such a way that all paths reach the root at the same time. Whenever two paths meet at a node, we obtain information on their relative ordering, depending on the edges used to reach the node; we store this information in the appropriate register. The paths may later part (the register is then reset to neither), but since all paths eventually reach the same node at the same time, we are certain to obtain their ordering, i.e., a setting of the registers which tells whether the inputs combine to give an element of $B^*a_1B^* \cdots B^*a_kB^*$.

Since the number of relevant paths involved in a positive CEP(M) instance depends only on the monoid and therefore is a constant, the above algorithm can be executed in deterministic logspace. In particular, making critical use of the CEP(M) assumption that a path exists in the circuit from each node to the output node, the breadth-first-search procedure (from leaf to output) invoked to count the number of paths from a leaf to the output node can be aborted when the number of stacked nodes exceeds the constant. □

LEMMA 4.13. *If monoid M does not belong to $\mathbf{M_{nil}}$, then problem* CEP(M) *is NL hard under $\leq_m^{NC^1}$ reducibility.*

Proof. By Lemma 3.1, at least one of monoids R_1, M_ba^* or their reverses, BA_2, or BH_2 divides M. Thus it suffices to prove hardness for these five minimal monoids. In all cases, we reduce from the accessibility problem in directed acyclic graphs. Let an instance consist in asking whether root v and leaf u are connected. We describe the reduction for the case of monoid R_1. First, we arbitrarily order the roots, placing v in the leftmost position. Then we create two new nodes v' and w, and we add an

edge from every root to w, plus one from v' to w, in such a way that the ordering of the roots is respected and that v' is inserted in this ordering immediately to the right of v. Next, we label all interior nodes with the operation of R_1, leaf u with element a, leaf v' with element b, and every other leaf with 1. The circuit evaluates to a at the output gate w iff the edge from v to w carries value a; that is, iff node v can be reached from u. Otherwise, the circuit evaluates to b.

The reduction for monoid M_{ba^*} is identical; there the modified circuit evaluates to $ab = 0$ if u and v are connected and to b otherwise. The reduction for the reverse of one of the above two monoids is obtained by simply reversing the order of the children of the output node w.

Finally, the reductions for monoids BA_2 and BH_2 are also similar; set $u = ab$, $v' = b$, and all the other leaves to 1, then if u is connected to v the circuit evaluates to 0, and if u is not connected to v the circuit will evaluate to b. □

We conclude this section with the case of UCEP(M).

PROPOSITION 4.14. *If M is a nontrivial aperiodic monoid, then problem* UCEP(M) *is NL-complete.*

Proof. Corollary 4.3 provides the upper bound. For *NL*-hardness, consider the minimal nontrivial aperiodic monoid, $C_{1,1}$. A reduction from the accessibility problem in a directed acyclic graph to UCEP(M) is obtained by labeling the source node with value 1 and all other in-degree-0 nodes with value 0. □

5. Conclusion. Finite monoids play a crucial role in the algebraic theory of finite automata. In particular, the algebraic classification of finite monoids maps to a classification of regular languages, and vice versa. In more general complexity theory, two recent streams of results have linked the classification of finite monoids to fundamental complexity classes. The first characterized the fine structure of the complexity class NC^1 in terms of word problems over monoids (see [4, 7, 28]). The second linked the classification of finite monoids to the discrete jumps in complexity, from membership in AC^0 to $PSPACE$ completeness, of the membership problem in transformation monoids (see [8, 9, 12, 26] and also [2]). In this paper, a third link between the structure of monoids and fundamental complexity classes was established, via the circuit evaluation problem.

When M is nonsolvable, CEP(M) is P-complete. When M is aperiodic, CEP(M) is either *NL*-complete, *L*-complete, or in AC^0, depending on further basic algebraic properties of M. Assuming $L \neq NL \neq P$, the complexity of CEP(M) is therefore completely determined in these cases. In the remaining solvable cases, CEP(M) $\in DET \subseteq NC^2$.

As an open question, our analysis in the solvable case needs refinement. For example, in the study of *word* problems over solvable monoids, the derived length[1] is closely related to the precise circuit depth of the corresponding NC^1 subclasses. What parameter associated with natural DET subclasses corresponds to the derived length in the context of CEP(M)?

Along those lines, we suspect a close connection between problem CEP(M) and the concept of *leaf languages* studied in [16, 17, 23]. Leaf languages provide a way to use regular languages as a basis for defining more general languages, as follows. Let each final configuration of a nondeterministic polynomial time Turing machine M

[1]For a solvable group G, the *derived length* of G is the length of the series $G_0 = \{1\} \triangleleft G_1 \triangleleft \cdots \triangleleft G_k = G$ in which G_i/G_{i-1} is an Abelian quotient of G_i, $1 \leq i \leq k$: it is also the smallest k such that all languages recognized by G belong to $\mathcal{M}^k_{0,q}$. By analogy, we define the derived length of a solvable monoid M to be the smallest k such that all languages recognized by M belong to $\mathcal{M}^k_{1,q}$.

emit a character from a fixed alphabet A. Ordering nondeterministic choices, let the *leaf string of M on input x* be the sequence of characters emitted at the leaves of M's computation tree on input x. Then, via M, any "leaf language" $Y \subseteq A^*$ defines a language, namely the set of inputs x such that the leaf string of M on x belongs to Y. A wealth of complexity classes, including $PSPACE$ and the polynomial hierarchy, were characterized as sets of languages definable via appropriately chosen regular leaf languages [16, 17, 23].

Now write $\mathcal{R}(\mathbf{V})$ for the class of languages definable via a regular leaf language recognized by a monoid from variety $\mathbf{V}$ (using nondeterministic *logspace* Turing machines). The hierarchy of solvable monoid varieties induces a (perhaps collapsing) hierarchy of classes $\mathcal{R}(\mathbf{V})$. What is the exact relationship between this hierarchy and problem CEP(M)? (This question has been partially investigated in [25].) Can such a relationship be exploited in order to define a useful parametrization of complexity classes like DET? In particular, adapting the several equivalent definitions of the polynomial hierarchy yields several possible meaningful definitions for complexity hierarchies built above MOD_qL and contained in DET. It would seem useful to know how these definitions relate to one another and whether the levels of these hierarchies have complete CEP(M) problems.

Acknowledgments. We thank the anonymous referees for their very thoughtful reading of our manuscript and for a more streamlined proof of Lemma 3.1.

REFERENCES

[1] C. Àlvarez and B. Jenner, *A very hard* log *space counting class*, in Theoret. Comput. Sci., 107 (1993), pp. 3–30.

[2] L. Babai, E. Luks, and A. Seress, *Permutation groups in NC*, in Proc. 19th ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 409–420.

[3] J. L. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity* I and II, Springer-Verlag, Berlin, New York, Heidelberg, 1988 and 1990.

[4] D. A. Barrington, *Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1*, J. Comput. System Sci., 38 (1989), pp. 150–164.

[5] D. A. M. Barrington, N. Immerman, and H. Straubing, *On uniformity within NC^1*, J. Comput. System Sci., 41 (1990), pp. 274–306.

[6] D. A. Barrington, H. Straubing, and D. Thérien, *Non-uniform automata over groups*, Inform. Comput., 89 (1990), pp. 109–132.

[7] D. A. Barrington and D. Thérien, *Finite monoids and the fine structure of NC^1*, J. Assoc. Comput. Mach., 35 (1988), pp. 941–952.

[8] M. Beaudry, *Membership testing in commutative transformation semigroups*, Inform. and Comput., 79 (1988), pp. 84–93.

[9] M. Beaudry, *Membership testing in threshold one transformation monoids*, Inform. and Comput., 113 (1994), pp. 1–25.

[10] M. Beaudry and P. McKenzie, *Circuits, matrices, and nonassociative computation*, in J. Comput. System Sci., 50 (1995), pp. 441–455.

[11] M. Beaudry, P. McKenzie, and P. Péladeau, *Circuits with monoidal gates*, in Proc. 10th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 665, Springer-Verlag, Berlin, New York, Heidelberg, 1993, pp. 555–565.

[12] M. Beaudry, P. McKenzie, and D. Thérien, *The membership problem in aperiodic transformation monoids*, J. Assoc., Comput. Mach., 39 (1992), pp. 599–616.

[13] R. Beigel, J. Gill, and U. Hertrampf, *Counting classes: Thresholds, parity, mods, and fewness*, in Proc. 7th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 415, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 49–57.

[14] F. Bédard, F. Lemieux, and P. McKenzie, *Extensions to Barrington's M-program model*, Theoret. Comput. Sci., 107 (1993), pp. 31–61.

[15] G. Bergman, *Embedding arbitrary algebras into groups*, Algebra Universalis, 25 (1988), pp. 107–120.
[16] D. Bovet, P. Crescenzi, and R. Silvestri, *Complexity classes and sparse oracles*, in Proc. 6th IEEE Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 102–108.
[17] D. Bovet, P. Crescenzi, and R. Silvestri, *A uniform approach to define complexity classes*, Theoret. Comput. Sci., 104 (1992), pp. 263–283.
[18] G. Buntrock, C. Damm, U. Hertrampf, and C. Meinel, *Structure and importance of logspace* MOD-*classes*, Math. Systems Theory, 25 (1992), pp. 223–237.
[19] S. R. Buss, S. Cook, A. Gupta, and V. Ramachandran, *An optimal parallel algorithm for formula evaluation*, SIAM J. Comput., 21 (1992), pp. 755–780.
[20] S. A. Cook, *A taxonomy of problems with fast parallel solutions*, Inform. and Comput., 64 (1985), pp. 2–22.
[21] S. A. Cook, and P. McKenzie, *Problems complete for deterministic logarithmic space*, J. Algorithms, 8 (1987), pp. 385–394.
[22] S. Eilenberg, *Automata, Languages and Machines*, vol. B, Academic Press, New York, 1976.
[23] U. Hertrampf, C. Lautermann, T. Schwentick, H. Vollmer, and K. Wagner, *On the power of polynomial time bit-reductions*, in Proc. 8th IEEE Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 200–207.
[24] N. Immerman, *Nondeterministic space is closed under complementation*, SIAM J. Comput., 17, 5 (1988), pp. 935–938.
[25] B. Jenner, P. McKenzie, and D. Thérien, *Logspace and logtime leaf languages*, Inform. and Comput., 129 (1996), pp. 21–33.
[26] D. Kozen, *Lower bounds for natural proof systems*, in Proc. 18th ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1977, pp. 254–266.
[27] R. E. Ladner, *The circuit value problem is* log-*space complete for* P, ACM SIGACT Newslett. 7 (1975), pp. 18–20.
[28] P. McKenzie, P. Péladeau, and D. Thérien, NC^1: *The automata-theoretic viewpoint*, Comput. Complexity, 1 (1991), pp. 330–359.
[29] J.-E. Pin, *Variétés de langages formels*, Masson, Paris, 1984 (in French); *Varieties of Formal Languages*, Plenum Press, New York, 1986 (in English).
[30] W. Ruzzo, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.
[31] W. Ruzzo, J. Simon, and M. Tompa, *Space-bounded hierarchies and probabilistic computations*, J. Comput. System Sci., 28 (1984), pp. 216–230.
[32] H. Straubing, *Varieties of recognizable sets whose syntactic monoids contain solvable groups*, Ph.D. thesis, University of California at Berkeley, Berkeley, CA, 1978.
[33] H. Straubing, *The variety generated by finite nilpotent monoids*, Semigroup Forum, 24 (1982), pp. 25–38.
[34] R. Szelepcsényi, *The method of forcing for nondeterministic automata*, Bull. European Assoc. Theoret. Comput. Sci., 33 (1987), pp. 96–100.
[35] D. Thérien, *Classification of finite monoids: The language approach*, Theoretical Comput. Sci., 14 (1981), pp. 195–208.
[36] D. Thérien, *Subword counting and nilpotent groups*, in Combinatorics on Words: Progress and Perspectives, L. J. Cummings, ed., Academic Press, New York, 1983, pp. 297–305.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 153–172, February 1997

009

PARALLELISM ALWAYS HELPS*

LOUIS MAK†

Abstract. It is shown that every unit-cost random-access machine (RAM) that runs in time T can be simulated by a concurrent-read exclusive-write parallel random-access machine (CREW PRAM) in time $O(T^{1/2} \log T)$. The proof is constructive; thus it gives a mechanical way to translate any sequential algorithm designed to run on a unit-cost RAM into a parallel algorithm that runs on a CREW PRAM and obtain a nearly quadratic speedup. One implication is that there does not exist any recursive function that is "inherently not parallelizable."

Key words. computational complexity, time complexity, random-access machine, parallel random-access machine, simulation, speedup

AMS subject classifications. 68Q05, 68Q10, 68Q15, 03D10, 03D15

PII. S0097539794265402

1. Introduction.

1.1. Motivation.

For some problems, the direct parallelization of a sequential algorithm gives a faster parallel algorithm. An example is matrix multiplication. The brute-force sequential algorithm for matrix multiplication runs in $O(n^3)$ time for $n \times n$ matrices. It is straightforward to parallelize this sequential algorithm to get an $O(\log n)$-time parallel algorithm using $O(n^3/\log n)$ processors. On the other hand, some problems are very difficult to parallelize. For example, depth-first search does not seem to admit itself to parallelization [7]. In this paper, we address the following question: Are all sequential algorithms parallelizable?

Cook and Reckhow [3] defined the unit-cost random-access machine (RAM). Fortune and Wyllie [6] introduced the parallel random-access machine (PRAM). These two models are, respectively, the most commonly used machine models for analyzing sequential and parallel algorithms. Thus the above question can be rephrased as follows: Given any unit-cost RAM R that runs in time T, is it always possible to construct a PRAM that simulates R in time $T' = o(T)$? We answer this question affirmatively by exhibiting such a construction with $T' = O(T^{1/2} \log T)$. Several variants of the PRAM have appeared in the literature since it was first introduced. The original model of Fortune and Wyllie has become known as the concurrent-read exclusive-write (CREW) PRAM, which is the model we use in our construction.

Parberry and Schnitger [15] considered the WRAM, a powerful variant of the PRAM. The WRAM differs from the CREW PRAM in three respects:

1. The WRAM is a concurrent-read concurrent-write (CRCW) priority PRAM [5].

2. The WRAM has a richer instruction set for arithmetic operations. The CREW PRAM supports only addition and subtraction, whereas the WRAM also allows unit-time unrestricted right shifts and modulus operations.

3. The WRAM and the CREW PRAM differ in the manner in which the processors are activated. In the WRAM, an arbitrary number of processors are self-activated at the beginning of the computation. In the CREW PRAM, only one processor is

* Received by the editors March 30, 1994; accepted for publication (in revised form) April 24, 1995. This research was supported by National Science Foundation grants CCR-8922008 and CCR-9315696.

http://www.siam.org/journals/sicomp/26-1/26540.html

† Coordinated Science Laboratory and Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, IL 61801 (mak@grinch.csl.uiuc.edu).

active initially. An active processor activates an idle processor explicitly by executing a FORK instruction. Consequently, in t steps, a PRAM can activate at most 2^t processors.

Parberry and Schnitger showed that every Turing machine that runs in time T can be simulated in constant time by a WRAM with $2^{O(T)}$ processors. The best-known simulation of unit-cost RAMs by Turing machines incurs a cubic overhead in the running time [3]. It follows that every unit-cost RAM with time complexity T can be simulated in constant time by a WRAM with $2^{O(T^3)}$ processors. It is desirable to reduce this huge number of processors used in the simulation for two reasons. The first reason is, obviously, to reduce the hardware requirement.

The second and more important reason is that the ability of the WRAM to use an arbitrary number of processors renders this model unreasonably powerful. The above result of Parberry and Schnitger essentially says that every decidable problem can be decided in constant time by a WRAM. This anomaly arises mainly from allowing self-activated processors. The *parallel-computation thesis* [8, 14] asserts that the class of languages accepted by any reasonable parallel-machine model in polynomial time is equivalent to PSPACE, where PSPACE, as usual, denotes the class of languages that can be accepted by deterministic Turing machines in polynomial space. The WRAM violates the parallel-computation thesis and is considered unreasonably powerful [13]. In contrast, the PRAM is considered reasonable because it obeys the parallel-computation thesis [6]. So the challenge is to speed up a unit-cost RAM by a PRAM with a reasonable number of processors; the number of processors should be small enough so that all processors can be activated explicitly within the simulation time.

1.2. Comparison with previous results. Dymond and Tompa [4] showed that every deterministic Turing machine running in time T can be simulated by a CREW PRAM in time $O(T^{1/2})$. However, the random-access memory of the PRAM is much more flexible than the linear tapes of the Turing machine, which forbid random access into individual tape cells. It was unclear whether it is the parallelism, the more flexible storage structure, or the combination of both that realizes such a quadratic speedup. Our result demonstrates that parallelism alone suffices to achieve an almost quadratic speedup.

To the best of our knowledge, in all previous speedup results [4, 10, 13, 15, 16, 18, 20], the machine being simulated is limited to the Turing machine. All these results depend on the fact that the changes in the configuration of a Turing machine in t steps are localized to the $2t-1$ cells around each tape head. In contrast, the random-access memory of a unit-cost RAM allows the RAM to change the contents of registers with widely different addresses in consecutive steps. The versatility of the random-access memory of a unit-cost RAM has defied all prior attempts to speed up a unit-cost RAM by a PRAM. This paper presents the first speedup theorem of unit-cost RAMs by PRAMs.

Reif [17] demonstrated that every probabilistic unit-cost RAM that runs in time T can be simulated by a probabilistic CREW PRAM in time $t(T, L) = O((T \log T \log(LT))^{1/2})$, where L is the largest integer manipulated by the probabilistic RAM during its computation. It is straightforward to modify Reif's proof to show that every unit-cost RAM running in time T can be simulated by a CREW PRAM in time $t(T, L)$. With unit-time addition, however, a RAM can generate integers as large as $2^{O(T)}$ in time T. Reif's result does not guarantee a speedup since $t(L, T) = O(T(\log T)^{1/2})$ when $L = 2^{O(T)}$. Our result gives a definite speedup of unit-cost RAMs by PRAMs, regardless of the value of L. It is routine to generalize our

proof to establish a speedup theorem of probabilistic unit-cost RAMs by probabilistic CREW PRAMs. This paper subsumes the above result of Reif. Thus all algorithms (deterministic and probabilistic) are parallelizable.

In summary, all previous simulation results suffer from one or more of the following drawbacks:

1. No definite speedup is guaranteed (Reif [17]).
2. The machine being sped up is limited to the Turing machine [4, 10, 13, 15, 16, 18, 20].
3. The speedup result fails to isolate the effect of parallelism; that is, apart from the parallelism, the simulator enjoys some additional advantage over the machine being simulated—for example, a more flexible storage structure (Dymond and Tompa [4]).
4. The simulator is too strong to be called reasonable because it violates the parallel computation thesis (Parberry and Schnitger [15]).

Our result does not suffer from any of the above drawbacks.

The rest of this paper is organized as follows. Section 2 defines the RAM and the PRAM models precisely. In section 3, we build up a repertoire of techniques for programming a PRAM efficiently. We use these techniques in section 4 to establish our main result: for every unit-cost RAM R with time complexity T, we construct a CREW PRAM that simulates R in time $O(T^{1/2} \log T)$. We conclude with a few comments in section 5. All logarithms are taken to base 2.

2. Definitions.

2.1. The unit-cost RAM. A RAM R consists of a *memory*, and a *program*. The memory is an infinite sequence of registers $(r(i))$, $i = 0, 1, \ldots$. The *address* of $r(i)$ is the integer i. Each register can hold an integer. Let $\langle r(i) \rangle$ denote the content of $r(i)$ and $|\langle r(i) \rangle|$ denote the absolute value of $\langle r(i) \rangle$. The program consists of a finite number of *statements*, numbered $1, 2, \ldots, Q$. Each statement contains one instruction. The allowed instructions are shown in Table 1. The input of R is a binary number $\alpha = \alpha_0 \alpha_1 \ldots \alpha_{n-1}$, where each $\alpha_i \in \{0, 1\}$. Initially, $r(0), r(1), \ldots, r(K-1)$ hold some constant values required in the computation of R, where K is a constant that depends on R; $r(K+i)$ holds α_i for $0 \leq i < n$, and $r(K+n)$ holds -1 to mark the end of the input. All other registers contain 0. A unit-cost RAM executes each instruction in one step. Each step takes unit time. Thus step t takes a unit-cost RAM from time $t-1$ to time t. The running time of a unit-cost RAM is the number of steps performed.

TABLE 1
Instructions of a RAM.

Instruction	Meaning
$r(i) \leftarrow r(j)$	$r(i)$ gets $\langle r(j) \rangle$.
$r(i) \leftarrow (r(0))$	$r(i)$ gets $\langle r(j) \rangle$, where $j = \lvert\langle r(0) \rangle\rvert$.
$(r(0)) \leftarrow r(j)$	$r(i)$ gets $\langle r(j) \rangle$, where $i = \lvert\langle r(0) \rangle\rvert$.
$r(i) \leftarrow r(j) + r(k)$	$r(i)$ gets $\langle r(j) \rangle + \langle r(k) \rangle$.
$r(i) \leftarrow r(j) - r(k)$	$r(i)$ gets $\langle r(j) \rangle - \langle r(k) \rangle$.
JUMP q	If $\langle r(0) \rangle \leq \langle r(1) \rangle$, then jump to statement q.
ACCEPT	Accept and halt.
REJECT	Reject and halt.

2.2. The PRAM. A PRAM P comprises a collection of processors $P(0)$, $P(1)$, ..., which communicate via a global memory $(g(i))$. The initial contents of the global memory are as follows: the first K' global registers hold some constants, where K'

is another constant that depends on P; the next $n+1$ global registers hold the n input bits, followed by the end-of-input marker, and all other global registers contain 0. Every processor is a unit-cost RAM. Each $P(p)$ has its own local memory $(r_p(i))$ and can use every global memory register in the same manner as it uses a local memory register. In addition, each processor has an extra FORK q instruction for processor activation. Initially, only $P(0)$ is active. Whenever a processor executes a FORK q instruction, a new processor is activated and starts running at statement q. When $P(p)$ executes the FORK instruction the tth time, processor $P(2^{t-1}(2p+1))$ is activated. The *processor id* (PID) of $P(p)$ is the integer p. When $P(p)$ is activated, its local register $r_p(0)$ is initialized with its PID p, and all other local registers of $P(p)$ contain 0. The PRAM P accepts if and only if $P(0)$ executes an ACCEPT instruction.

In a PRAM, several processors may attempt to access the same memory cell at the same time. A PRAM may allow concurrent-read and concurrent-write (CRCW) operations, concurrent-read and exclusive-write (CREW) operations, or exclusive-read and exclusive-write (EREW) operations [2, 22, 25]. In a CRCW PRAM, some mechanism is necessary to resolve the simultaneous write conflicts [2, 8, 21]. Fich et al. [5] studied the relationships between CRCW PRAMs with different conflict-resolution mechanisms.

In what follows, we restrict our attention to CREW PRAMs. Unless otherwise stated, our results also hold for CRCW PRAMs.

3. Techniques for programming PRAMs. In this section, we present several techniques for programming the PRAM. First, we show how to perform the following operations quickly on a PRAM: logical AND, summation, and multiplication of "small" integers. Second, we describe a fast implementation of multidimensional memory on a PRAM. Third, we explain how every processor can extract useful information from its PID efficiently.

3.1. Logical AND, summation, and multiple memories. It is convenient to interpret integers as logical values. We interpret a nonzero integer as true and 0 as false.

LEMMA 3.1. [folklore] *Suppose in a PRAM P, the global memory registers $g(1)$, $g(2), \ldots, g(n)$ store n integers $k_1, k_2, \ldots, k_n$. Then P can find the sum and the logical AND of these n integers in $O(\log n)$ time.*

By interleaving memory registers, Cook and Reckhow [3] demonstrated that a unit-cost RAM with a single memory can simulate a unit-cost RAM with multiple memories with merely a constant factor overhead in the running time. By applying the same technique to the PRAM, it is easy to prove the following lemma.

LEMMA 3.2. [folklore] *Let $\gamma > 1$. Every PRAM with time complexity T and γ global memories $(g_1(i)), (g_2(i)), \ldots, (g_\gamma(i))$ can be simulated in time $O(T)$ by a PRAM with one global memory.*

3.2. Multiplication of small integers. Trahan et al. [24] studied PRAMs with unit-time multiplication. By the following lemma, we may assume that ordinary PRAMs can perform unit-time multiplication of "small" integers.

LEMMA 3.3. *Let P be a PRAM that* (i) *runs in time T and* (ii) *can perform unit-time multiplication on T-bit integers. Then P can be simulated by an ordinary PRAM in time $O(T)$.*

Proof. We use a PRAM P' with multiple memories to simulate P. Lemma 3.3 then follows from Lemma 3.2. P' simulates P step by step. We only need to show that P' can multiply two T-bit integers in $O(1)$ time. Multiplication reduces to squaring and halving via the identity $xy = ((x+y)^2 - x^2 - y^2)/2$. For two T-bit integers x

and y, $x+y$ and $2xy$ are, respectively, at most $T+1$ and $2T+1$ bits long. It suffices to show that P' can perform in $O(1)$ time (i) squaring on $(T+1)$-bit positive integers and (ii) halving (right shift) on $(2T+1)$-bit positive integers. Before simulating P, P' precomputes a Square Table of size 2^{T+1} and a Right-Shift Table of size 2^{2T+1}. Then during the simulation, P' can perform squaring and halving in $O(1)$ time by table lookup. It remains to demonstrate that the Square Table and the Right-Shift Table can be precomputed in $O(T)$ time.

P' uses four global memories $(ls(i))$, $(rs(i))$, $(lsb(i))$, and $(sq(i))$ to implement four tables:

1. Left-Shift Table: $\langle ls(i)\rangle = 2i$;
2. Right-Shift Table: $\langle rs(i)\rangle = \lfloor i/2 \rfloor$;
3. Least-Significant-Bit Table: $\langle lsb(i)\rangle =$ least significant bit of i;
4. Square Table: $\langle sq(i)\rangle = i^2$.

P' initializes the first three tables for $0 \le i < 2^{2T+1}$ as follows. In $O(T)$ time, P' activates processors $P(0), P(1), \ldots, P(2^{2T+1}-1)$. Each processor does the following:

1. Store PID + PID in ls(PID).
2. Store PID in rs(PID + PID) and rs(PID + PID + 1).
3. Store PID $-$ $ls(rs(\text{PID}))$ in lsb(PID).

Obviously, steps 1, 2, and 3 take $O(1)$ time. After the Left-Shift Table, the Right-Shift Table, and the Least-Significant-Bit Table are initialized, then for $0 \le i < 2^{T+1}$, each $P(i)$ computes the square of its PID using the paper–pencil multiplication method (repeated shift and add) and stores the result in $sq(i)$. This takes $O(T)$ time. Hence all four tables can be precomputed in $O(T)$ time.

We have assumed that P' knows the value of T a priori. This assumption can be removed easily; P' just tries successive powers of two as an estimate of T. This modification does not increase the asymptotic running time of P'. □

3.3. Multidimensional memory. A d-dimensional RAM is one with memory $(r(i_1, i_2, \ldots, i_d))$, where $i_1, i_2, \ldots, i_d \ge 0$. A d-dimensional PRAM is one with global memory $(g(i_1, i_2, \ldots, i_d))$; each processor of a d-dimensional PRAM is a d-dimensional RAM. Robson [19] showed that ordinary RAMs can simulate multidimensional RAMs with only a constant-factor overhead in the running time. However, the proof of Robson cannot be adapted directly to prove the analogous result for PRAMs. Briefly, the reason is as follows. To simulate a RAM R with two-dimensional memory $(r(i,j))$ by an ordinary RAM R' with memory $(r'(i))$, Robson devised a mapping from the $r(i,j)$'s to the $r'(i)$'s. This mapping depends on the sequence of $r(i,j)$'s accessed during the computation of R, and R' constructs this mapping incrementally as it simulates R step by step. Consider applying the same idea to simulate a PRAM P with two-dimensional global memory $(g(i,j))$ by an ordinary PRAM P' with global memory $(g'(i))$. If we simulate each processor of P by a corresponding processor of P' as in the proof of Robson, then different processors of P may access the $g(i,j)$'s in different ways, and hence different processors of P' may have different mappings. Thus some processor of P' may think that the value of $g(0,0)$ is stored in $g'(0)$, whereas another processor of P' thinks that the same value is stored in $g'(1)$. Obviously, such a simulation of P by P' does not work.

All in all, the analogous result for PRAMs does hold, as shown by the next lemma.

LEMMA 3.4. *Every d-dimensional PRAM P running in time T can be simulated by an ordinary PRAM P' in time $O(T)$.*

Proof. P' uses processor $P'(i)$ to simulate the corresponding processor $P(i)$ of P. Every $P'(i)$ simulates $P(i)$ step by step. It suffices to explain how to emulate d-dimensional memories by one-dimensional memories. We demonstrate how $P'(i)$

emulates an access of $P(i)$ to the d-dimensional global memory of P by an access to the one-dimensional global memory of P'. $P'(i)$ uses its one-dimensional local memory to emulate the d-dimensional local memory of $P(i)$ in a similar fashion.

P has global memory $(g(i_1, i_2, \ldots, i_d))$; P' has global memory $(g'(i))$. In time T, $P(i)$ can produce integers no longer than CT bits for some constant C. Define $b = 2^{CT}$ and $\eta(i_1, i_2, \ldots, i_d) = \sum_{j=1}^{d} i_j b^{j-1}$. We map $g(i_1, i_2, \ldots, i_d)$ of P to $g'(\eta(i_1, i_2, \ldots, i_d))$ of P'. It is easy to verify that the $\eta(i_1, i_2, \ldots, i_d)$'s are distinct for $0 \leq i_1, i_2, \ldots, i_d < 2^{CT}$. To emulate an access to $g(i_1, i_2, \ldots, i_d)$ by $P(i)$, $P'(i)$ computes $\eta(i_1, i_2, \ldots, i_d)$ and accesses $g'(\eta(i_1, i_2, \ldots, i_d))$.

It remains to show that computing η takes $O(1)$ time. By repeated doubling, $b = 2^{CT}$ can be precomputed in $O(T)$ time. For $i_1, i_2, \ldots, i_d < 2^{CT}$, η is at most dCT bits long. By Lemma 3.3, we may assume that $P'(i)$ can perform multiplication on dCT-bit integers in $O(1)$ time. Thus $P'(i)$ can compute η in $O(1)$ time.

Again, we have presumed that the value of T is available. This assumption can be removed in the same way as in the proof of Lemma 3.3. □

Lemma 3.4 shows that without loss of generality, we may assume that CREW PRAMs have multidimensional memories. Apparently, some authors have used this fact without proof [4, 17].

3.4. Extracting information from the PID. The advantage of a PRAM over a RAM is that in a PRAM, many processors can work together in parallel. Clearly, this advantage is defeated if all processors just do the same thing on the same data, in which case one processor is as good as many. To take advantage of the parallelism, therefore, different processors have to operate differently. This is easily achieved by exploiting the distinctness of the PIDs; each processor consults its PID to determine its operation. For our later purpose, we require each processor to be able to look at successive single bits and successive $O(\log T)$ bits of its PID in order to determine its operation. Next, we demonstrate that every PRAM can be modified to fulfill this requirement.

Let P be a PRAM with time complexity T. In time T, P can activate at most 2^T processors. The PID of every processor is at most T bits long. We modify P as follows.

1. P activates all 2^T processors before any actual computation.

2. P starts its computation by initializing in $O(1)$ time a Least-Significant-Bit Table, a Right-Shift Table, and a Left-Shift Table, all of size 2^T, as described in the proof of Lemma 3.3. Using the first two tables, each processor can extract successive single bits of its PID, spending $O(1)$ time per bit.

3. P implements two additional tables with global memories $(lsb'(i))$ and $(rs'(i))$:

(i) $lsb'(i)$ = the least significant $\lfloor \log T \rfloor$ bits of i.

(ii) $rs'(i) = \lfloor i/2^{\lfloor \log T \rfloor} \rfloor$, i.e., i right-shifted $\lfloor \log T \rfloor$ bits.

These two tables can be precomputed in $O(\log T)$ time as follows. We presume the availability of the three tables mentioned in modification 2. For $0 \leq i < 2^T$, processor $P(i)$ does the following:

(i) Right shift its PID $\lfloor \log T \rfloor$ times and store the result in $rs'(i)$.

(ii) Left shift $rs'(i)$ $\lfloor \log T \rfloor$ times, subtract the result from its PID, and store the difference in $lsb'(i)$.

Then each processor can extract successive $\lfloor \log T \rfloor$ bits of its PID by table lookup, spending $O(1)$ time per $\lfloor \log T \rfloor$ bits. We have assumed that P knows a priori the values of T and $\lfloor \log T \rfloor$. The knowledge of T is justifiable, as argued in the proof of Lemma 3.3, and $\lfloor \log T \rfloor$ is simply the number of bits in the binary representation

of T.

These modifications increase the running time of P by at most a constant factor.

4. Speedup of RAMs by PRAMs. We now prove that the PRAM is always faster than the RAM.

THEOREM 4.1. *Every unit-cost RAM running in time T can be simulated by a CREW PRAM in time $O(T^{1/2}\log T)$ with $T^{O(T^{1/2})}$ processors.*

Let R be a unit-cost RAM with memory $(r(i))$ and time complexity $T = T(n)$. We devise a CREW PRAM P with multiple multidimensional memories that simulates R in time $O(T^{1/2}\log T)$. Theorem 4.1 then follows from Lemmas 3.2 and 3.4. Let A be a large enough constant so that every address in the program of R can be encoded in A bits; we choose A to be at least $3\log 3 + 1$ to suit our later purpose. As the input length n tends to infinity, so does T since $T(n) \geq n$. Consequently, if n exceeds some constant n_0, then $AT^{1/2} > \log(2(T+K+n+1))$, where K is a constant that depends on R as explained in section 2.1. It suffices to argue that P runs in $O(T^{1/2}\log T)$ time for $n > n_0$ since we can modify P to handle inputs of length less than n_0 by table lookup. We assume that P knows the value of $T^{1/2}$ in advance. Otherwise, P tries successive powers of two as an estimate of $T^{1/2}$.

4.1. Overview of simulation. Dymond and Tompa [4] proved that every Turing machine with time complexity t can be sped up by a CREW PRAM in time $O(t^{1/2})$; briefly, their proof is as follows. In $O(t^{1/2})$ time, the PRAM precomputes a transition table that represents the $t^{1/2}$-step transition function of the Turing machine. Then the PRAM can simulate a block of $t^{1/2}$ steps of the Turing machine in $O(1)$ time by consulting this table. The PRAM accesses this table $t^{1/2}$ times to simulate the Turing machine for t steps. We would like to point out that this standard technique of precomputing a transition table, though useful for simulating the Turing machine, cannot be applied to speed up the unit-cost RAM by the CREW PRAM. Below we explain why this technique works for the Turing machine but not the unit-cost RAM.

In $t^{1/2}$ steps, the changes in the configuration of a Turing machine M depend only on the *local configuration* of M, i.e., the state of M and the contents of $O(t^{1/2})$ cells around each tape head. Consider a transition table that maps the current local configuration of M to the local configuration $t^{1/2}$ steps afterwards. Such a table contains $2^{O(t^{1/2})}$ entries since there are $2^{O(t^{1/2})}$ different local configurations. A CREW PRAM can build this table in $O(t^{1/2}) = o(t)$ time. Thus an asymptotic speedup is possible.

Now consider the unit-cost RAM R (with time complexity T). In $T^{1/2}$ steps, the changes in the configuration of R are not localized to $O(T^{1/2})$ consecutive registers; the PRAM P cannot build a transition table for local configurations as in the case of the Turing machine. In time T, R can construct integers as large as $2^{\Theta(T)}$. With indirect addressing, R may use these integers as addresses and assigns to register $r(i)$ an integer j, where $0 \leq i, j \leq 2^{\Theta(T)}$. In T steps, R can write to $\Theta(T)$ different registers. Hence there are at least $(2^{\Theta(T)})^{\Theta(T)} = 2^{\Omega(T^2)}$ different configurations of R. A transition table that maps the current configuration of R to the configuration $T^{1/2}$ steps afterwards will have $2^{\Omega(T^2)}$ entries. To generate such a huge table, P requires $\Omega(T^2) = \omega(T)$ time; this results in an asymptotic slowdown. Therefore, the technique of precomputing a transition table does not work for the unit-cost RAM. The novel idea in this paper is to simulate the unit-cost RAM without building its transition table.

Fix an input $\alpha = \alpha_0\alpha_1 \ldots \alpha_{n-1}$ and consider the computation of R on α. Formally, the configuration of R at time t consists of the statement number of R at time

t and the contents of all registers at time t. Denote the configuration of R at time t by *config*(t).

Our simulation comprises two phases. In phase I, P uses $O(T^{1/2}\log T)$ time to activate $T^{1/2}$ groups of $T^{O(T^{1/2})}$ processors. For $1 \le m \le T^{1/2}$, the processors in group m perform some preprocessing such that after the preprocessing, *config*$(mT^{1/2})$ can be computed from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time. All groups do the preprocessing simultaneously. In phase II, P finds *config*(T) as follows. The initial configuration of R, *config*(0), can be determined trivially. For $m = 1, 2, \ldots, T^{1/2}$, P computes *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time. Let q^* be the statement number in *config*(T). P accepts if and only if statement q^* contains an ACCEPT instruction. Both phases take $O(T^{1/2}\log T)$ time. Next, we present an efficient representation of the configuration of R and then provide the details of phases I and II.

4.2. Representing the configuration of *R*. The PRAM P uses a data structure CONFIG to represent the configuration of R. One difficulty is that P cannot use a single register to store the content of a corresponding register of R. This is because R can generate integers as large as $2^{O(T)}$ in time T, but P can produce integers no larger than $T^{O(T^{1/2})}$ within the intended $O(T^{1/2}\log T)$ time bound. To overcome this difficulty, P divides every $O(T)$-bit integer into $N = O(T^{1/2})$ blocks, each $B = AT^{1/2}$ bits long. Without loss of generality, we assume that R represents negative integers using sign-and-magnitude representation; thus R works with nonnegative integers exclusively. With this simplifying assumption, every block in our blockwise representation is a nonnegative integer.

Initially, all registers of R contain 0, except for $r(0), r(1), \ldots, r(K+n)$. By convention, the first step of R is step 1, and $r(0), r(1), \ldots, r(K+n)$ are first written to in step 0 (i.e., they are initialized at time 0). For $0 \le i \le K+n$, let u_i denote $r(i)$. For $i > K+n$, if a new register of R is written to in step $i-K-n$, then let u_i denote this register; otherwise, u_i is undefined. To describe the configuration of R at time t, it suffices to specify the statement number of R at time t and the address and content at time t of u_i for $0 \le i \le t+K+n$.

CONFIG consists of three global memories $(a(i,j))$, $(c(i,j))$, and $(b(i))$. To represent the configuration of R at time t, register $b(0)$ holds the statement number of R at time t. If $0 \le i \le t+K+n$ and u_i is defined, then $c(i,j)$ holds the jth block of B bits in $\langle u_i \rangle$. Thus $\langle u_i \rangle = \sum_{j=0}^{N-1} \langle c(i,j) \rangle 2^{jB}$. For brevity, we say that $c(i)$ holds $\langle u_i \rangle$, or $\langle c(i) \rangle = \langle u_i \rangle$, implying the blockwise representation. Register $a(i)$ holds the address of u_i in the same blockwise format. If $t < i-K-n \le T$ or u_i is undefined, then $a(i)$ holds -1, and $c(i)$ is not used. The number of registers that CONFIG uses is therefore $O(NT) = O(T^{3/2})$.

Henceforth, when we mention *config*(t), we imply the above representation.

4.3. Phase I.

4.3.1. Static, dynamic, and effective instructions. Due to conditional jumps, a program statement may be executed more than once. For clarity, we distinguish between a *static instruction* and a *dynamic instruction*. The former is a static entity in a program statement of R. The latter is an executed instruction—an instance of a static instruction during the computation of R. A static instruction may correspond to none or many dynamic instructions.

Divide the dynamic instructions of R into three types:

1. ACCEPT, REJECT, and JUMP;

2. direct and indirect load and store instructions: $r(i) \leftarrow r(j)$, $r(i) \leftarrow (r(0))$, and $(r(0)) \leftarrow r(j)$;

3. arithmetic operations: $r(i) \leftarrow r(j) + r(k)$ and $r(i) \leftarrow r(j) - r(k)$.

Consider the effect of each dynamic instruction on the memory of R. A type-1 instruction does not change the content of any register. As far as the effect on the memory is concerned, a type-1 instruction is equivalent to a $u_0 \leftarrow u_0$ instruction. An instruction of type-2 copies the content of one register to another. Without loss of generality, we assume that $r(K-1)$ always holds 0. Reading from an uninitialized register is the same as reading from $r(K-1) = u_{K-1}$ since all uninitialized registers contain 0. The effect of a type-2 instruction is thus $u_{i'} \leftarrow u_{j'}$ for some $0 \le i', j' \le T+K+n$. A type-3 instruction finds the sum or difference of the contents of two registers and stores the result in a third register. The effect of a type-3 instruction is either $u_{i'} \leftarrow u_{j'} + u_{k'}$ or $u_{i'} \leftarrow u_{j'} - u_{k'}$ for some $0 \le i', j', k' \le T + K + n$. Therefore, each dynamic instruction is, in effect, of the form $u_{i'} \leftarrow u_{j'}$, $u_{i'} \leftarrow u_{j'} + u_{k'}$, or $u_{i'} \leftarrow u_{j'} - u_{k'}$ for some $0 \le i', j', k' \le T + K + n$. Since $T = T(n) \ge n$, there are $O(T^3)$ *effective instructions* of the above forms. Note that one static instruction may correspond to several dynamic instructions, each equivalent to a different effective instruction.

4.3.2. The preprocessing. We fix m and describe the processors in group m. Let π_m be the triple (q_m, β_m, σ_m), where q_m is the statement number of R at time $(m-1)T^{1/2}$, σ_m is the sequence of $T^{1/2}$ effective instructions from time $(m-1)T^{1/2}$ to $mT^{1/2}$, and β_m is a binary string that encodes the outcomes of all conditional jumps between time $(m-1)T^{1/2}$ and $mT^{1/2}$. For uniformity, we view every static and dynamic instruction as a conditional jump. An ACCEPT instruction, for example, may be viewed as a conditional jump where the condition is always false, and the destination of the jump is statement 1. In this way, β_m is always of length $T^{1/2}$. The triple π_m specifies the behavior of R between time $(m-1)T^{1/2}$ and $mT^{1/2}$. Using the information contained in π_m, group m performs some preprocessing that enables *config*$(mT^{1/2})$ to be computed from *config*$((m-1)T^{1/2})$ quickly. One problem is that group m does not know π_m in advance. To surmount this problem, group m uses enough processors to try all possible triples. Let Q be the number of statements in the program of R. The number of possible triples is thus $Q \times 2^{T^{1/2}} \times O(T^3)^{T^{1/2}} = T^{O(T^{1/2})}$. Group m uses $T^{O(T^{1/2})}$ processors, which can be activated in $O(T^{1/2} \log T)$ time. Each processor is responsible for a distinct triple, which is encoded in the processor's PID. All processors in all groups carry out their preprocessing simultaneously in parallel.

We focus on one specific processor P_π of group m, which is responsible for one particular triple $\pi = (q, \beta, \sigma)$. Notice the difference in notation: the PID of $P(p)$ is p, whereas the PID of P_π is not π, but the triple π is encoded in the PID of P_π. P_π decodes its PID to obtain q, β, and σ. To obtain the sequence of $T^{1/2}$ effective instructions σ, P_π extracts the least significant $O(T^{1/2} \log T)$ bits from its PID, $O(\log T)$ bits at a time. Every effective instruction can be encoded in $O(\log T)$ bits since there are $O(T^3)$ different effective instructions. To recover the individual bits of β, P_π extracts the next $T^{1/2}$ bits from its PID, one bit at a time. The next $\lfloor \log Q \rfloor + 1$ bits of the PID constitute q. Using the techniques prescribed in section 3.4, P_π can decode its PID in $O(T^{1/2})$ time. P_π saves all decoded information in tables so that it can access each bit of β and each effective instruction in $O(1)$ time by table lookup.

To facilitate our discussion, we say the triple π "happens" if the actual behavior of R conforms with the information contained in π. Now π may or may not happen. In phase I, P_π performs some preprocessing so that in phase II, once *config*$((m-1)T^{1/2})$ has been computed, P_π is able to decide in $O(\log T)$ time whether π actually happens

and, if so, computes *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time. Because of the way we represent the configuration of R (section 4.2), to compute *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$, it suffices to determine the following:

1. the statement number of R at time $mT^{1/2}$;
2. for $(m-1)T^{1/2} < i - K - n \le mT^{1/2}$, the address of u_i if u_i is defined;
3. for $0 \le i \le mT^{1/2} + K + n$, the content of u_i at time $mT^{1/2}$ if u_i is defined.

Below we explain the preprocessing that enables P_π to determine each of the above three items efficiently in phase II, assuming π actually happens.

4.3.3. The statement number. Starting from statement q, P_π steps through the program of R statement by statement, following the flow of control defined by β. Meanwhile, P_π keeps track of the statement number of R. After $T^{1/2}$ steps, P_π obtains the statement number of R at time $mT^{1/2}$. This preprocessing takes $O(T^{1/2})$ time.

4.3.4. The addresses of the u_i's. In $O(\log T)$ time, P_π activates $T^{1/2}$ processors P_i, where $(m-1)T^{1/2} < i - K - n \le mT^{1/2}$. Each P_i is responsible for finding the address of u_i.

We fix i and describe P_i. P_i considers the effective instruction in step $s = i-K-n$ given by σ. Suppose this effective instruction is of the form $u_{i'} \leftarrow u_{j'}$. Other cases are handled similarly. If $i' \ne i$, then no new register is written to in step s, and u_i is undefined. Otherwise, u_i is the register first written to in step s. P_i steps through the program of R in the same manner as described in section 4.3.3 and finds the dynamic instruction in step s. If this dynamic instruction is of the form $r(j) \leftarrow r(k)$ or $r(j) \leftarrow (r(0))$, then the address of u_i is j. The blockwise representation of j is readily obtained since all addresses in the program of R are at most $A \le B$ bits long; the least significant B-bit block of j is just j itself, and all other blocks are 0. If the dynamic instruction in step s is of the form $(r(0)) \leftarrow r(k)$, then the address of u_i is the content of $r(0)$ at time $s-1$. Denote by $\langle u_i, t\rangle$ the content of u_i at time t. In section 4.3.5, we explain the preprocessing for finding $\langle u_i, mT^{1/2}\rangle$. P_i performs the preprocessing for finding $\langle r(0), s-1\rangle = \langle u_0, s-1\rangle$ in a similar fashion.

4.3.5. The contents of the u_i's. Since the sole arithmetic operations permitted are addition and subtraction, it follows that for a fixed π, $\langle u_i, mT^{1/2}\rangle$ is a linear combination of the $\langle u_i, (m-1)T^{1/2}\rangle$'s. Let

$$\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle u_j, (m-1)T^{1/2}\rangle,$$

where the C_{ij}'s are integer coefficients which depend only on π. In $O(\log T)$ time, P_π deploys $O(T^2)$ processors P_{ij}, where $0 \le i, j \le T+K+n$. Each P_{ij} finds C_{ij} in phase I.

We fix i and j and describe P_{ij}. P_{ij} creates an empty directed multigraph G and then processes the effective instructions specified by σ one by one. As P_{ij} considers each effective instruction, it inserts nodes and edges into G. P_{ij} marks each edge either "positive" or "negative." Node w is a positive child of node v if the edge (v, w) is positive. A negative child is defined analogously. Let v^+ and v^-, respectively, be the set of positive and negative children of v.

P_{ij} maintains a counter τ to keep track of the step corresponding to the effective instruction currently under consideration. P_{ij} initializes τ to $(m-1)T^{1/2}+1$ and increments τ after every effective instruction. P_{ij} names each node either $[u_k, (m-1)T^{1/2}]$ or $[u_k, \tau]$ for some $0 \le k \le T+K+n$. Intuitively, node $[u_k, t]$ represents

$\langle u_k, t\rangle$. For a node $v = [u_k, t]$, we write $\langle v\rangle$ for $\langle u_k, t\rangle$. The edges are marked such that

$$\langle v\rangle = \sum_{w\in v^+} \langle w\rangle - \sum_{w\in v^-} \langle w\rangle \quad \text{for each node } v. \tag{1}$$

This will become clear after we explain how P_{ij} constructs G. After processing all $T^{1/2}$ effective instructions, P_{ij} uses G to obtain C_{ij}.

4.3.6. Constructing G. P_{ij} considers the $T^{1/2}$ effective instructions specified by σ one by one and constructs G as follows. For a $u_{i'} \leftarrow u_{j'}$ instruction, P_{ij} does the following. Create node $[u_{i'}, \tau]$. If G does not contain a node $[u_{j'}, \tau']$ for some $\tau' < \tau$, then create node $[u_{j'}, (m-1)T^{1/2}]$. Let $\tau_{j'} < \tau$ be maximum such that G contains node $[u_{j'}, \tau_{j'}]$. Insert edge $([u_{i'}, \tau], [u_{j'}, \tau_{j'}])$ and mark it positive.

P_{ij} processes a $u_{i'} \leftarrow u_{j'} - u_{k'}$ instruction as follows. Create node $[u_{i'}, \tau]$. If G does not contain a node $[u_{j'}, \tau']$ for some $\tau' < \tau$, then create node $[u_{j'}, (m-1)T^{1/2}]$. Similarly, if G does not contain a node $[u_{k'}, \tau']$ for some $\tau' < \tau$, then create node $[u_{k'}, (m-1)T^{1/2}]$. Let $\tau_{j'}, \tau_{k'} < \tau$ be maximum such that G contains nodes $[u_{j'}, \tau_{j'}]$ and $[u_{k'}, \tau_{k'}]$. Insert a positive edge $([u_{i'}, \tau], [u_{j'}, \tau_{j'}])$ and a negative edge $([u_{i'}, \tau], [u_{k'}, \tau_{k'}])$. A $u_{i'} \leftarrow u_{j'} + u_{k'}$ instruction is processed in the same way except that both of the inserted edges are positive.

It is mechanical to verify that the above construction yields a graph which satisfies (1), and every node of the graph has out-degree at most two. We illustrate the above construction with an example for P_π of group $m = 1$ with $T^{1/2} = 5$. Figure 1 shows the effective instructions specified by π. The graph constructed by P_{ij} appears in Fig. 2.

Step	Instruction
1	$u_0 \leftarrow u_1$
2	$u_0 \leftarrow u_0 + u_0$
3	$u_0 \leftarrow u_0 - u_2$
4	$u_1 \leftarrow u_1 + u_0$
5	$u_0 \leftarrow u_0 + u_1$

Effect on memory: $\langle u_0, 5\rangle = 5\langle u_1, 0\rangle - 2\langle u_2, 0\rangle$,
$\langle u_1, 5\rangle = 3\langle u_1, 0\rangle - \langle u_2, 0\rangle$.

In this example, $C_{01} = 5$, $C_{02} = -2$, $C_{11} = 3$, and $C_{12} = -1$.

FIG. 1. *The $T^{1/2} = 5$ effective instructions specified by π and their effect on the memory (example).*

4.3.7. Computing C_{ij}. We explain how P_{ij} uses G to compute C_{ij}. P_{ij} checks whether G contains a node $[u_i, \tau']$ for some $\tau' > (m-1)T^{1/2}$.

Case I (no such node exists). By construction of G, P_{ij} will create node $[u_i, \tau']$ if R writes to u_i in step τ'. The hypothesis thus implies that R does not write to u_i between time $(m-1)T^{1/2}$ and $mT^{1/2}$. It follows that $\langle u_i, mT^{1/2}\rangle = \langle u_i, (m-1)T^{1/2}\rangle$. Ergo, $C_{ij} = 0$ for $j \neq i$, and $C_{ii} = 1$.

Case II (otherwise). Let τ_i be maximum such that G contains node $[u_i, \tau_i]$. Similar arguments as in Case I give $\langle u_i, mT^{1/2}\rangle = \langle u_i, \tau_i\rangle$.

Consider the subgraph H of G induced by node $[u_i, \tau_i]$ and all its descendants. By construction, G (and hence H) is a directed acyclic multigraph. P_{ij} sorts the nodes in H topologically and labels each edge in H with an integer as follows. P_{ij} considers the nodes in H in topological order. For each node v, P_{ij} labels the outgoing edges of v. When P_{ij} considers node v, all incoming edges of v are labeled since P_{ij} considers the

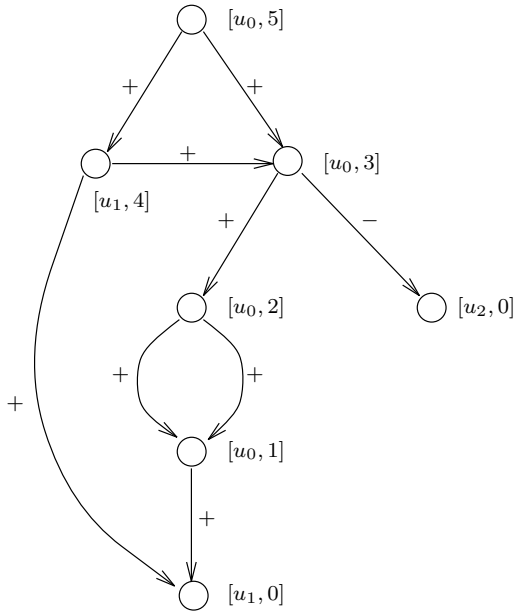


FIG. 2. *The graph G constructed by P_{ij} after processing the effective instructions in Fig.* 1.

nodes in topological order. Evidently, the first node considered is $[u_i, \tau_i]$. P_{ij} labels every positive and negative outgoing edge of $[u_i, \tau_i]$ with 1 and -1, respectively. For each remaining node v in topological order, let $\lambda(H, v)$ be the sum of the labels on the incoming edges of v in H. P_{ij} labels each positive and negative outgoing edge of v with $\lambda(H, v)$ and $-\lambda(H, v)$, respectively. Figure 3 shows the result of applying the above labeling algorithm to the graph in Fig. 2.

For $s > 0$, let H_s be the subgraph of H induced by all labeled edges after s nodes are considered. The *leaves* of H_s, denoted by $L(H_s)$, are the nodes in H_s with no outgoing edges. The following invariant is a consequence of (1): After s nodes are considered, $\langle u_i, \tau_i \rangle = \sum_{v \in L(H_s)} \lambda(H_s, v)\langle v \rangle$. Therefore, after all edges in H are labeled, $\langle u_i, mT^{1/2} \rangle = \sum_{v \in L(H)} \lambda(H, v)\langle v \rangle$. By construction, every leaf of H is named $[u_j, (m-1)T^{1/2}]$ for some j. Hence $C_{ij} = \lambda(H, [u_j, (m-1)T^{1/2}])$ if $[u_j, (m-1)T^{1/2}]$ is a leaf of H; otherwise, $C_{ij} = 0$. For the example in Fig. 3, P_{01} determines that $C_{01} = 4 + 1 = 5$, and P_{02} concludes that $C_{02} = -2$.

In the above labeling algorithm, the sum of the absolute values of all the labels on the edges of H_s is at most triple that of H_{s-1} since every node in H has out-degree at most two. The number of nodes in H is $|H| \leq |G| \leq 3T^{1/2}$ because at most three nodes are created for each of the $T^{1/2}$ effective instructions. Therefore, $|C_{ij}| \leq 3^{3T^{1/2}}$ for all i and j. Each C_{ij} is at most $B = AT^{1/2}$ bits long, since $A \geq 3 \log 3 + 1$.

The number of edges in G is $O(|G|)$, since each node has bounded out-degree. Constructing G and H, topologically sorting H, labeling the edges of H, and computing $\lambda(H, [u_j, (m-1)T^{1/2}])$ all take $O(|G|) = O(T^{1/2})$ time. Thus the bottleneck in phase I is the activation of enough processors to try all possible π, which takes $O(T^{1/2} \log T)$ time.

4.3.8. Table precomputation. In phase II, P has to extract efficiently the most and least significant B bits of a $2B$-bit integer. In phase I, P precomputes two tables $(h1(i))$ and $(h2(i))$ so that the first and second half of i can be ex-

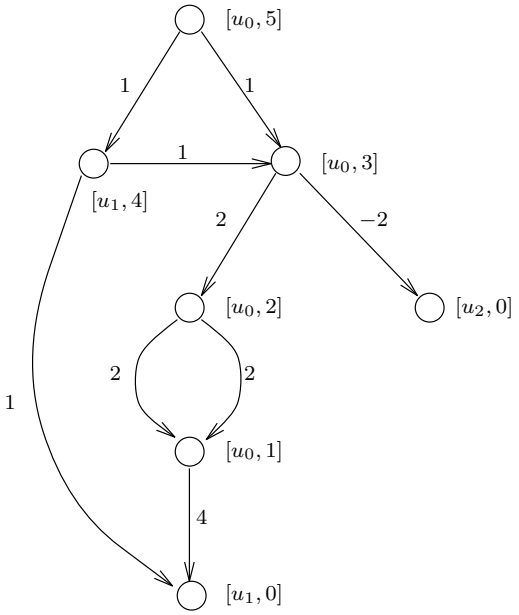


FIG. 3. *The result of applying the labeling algorithm of section* 4.3.7 *to the graph in Fig.* 2.

tracted in $O(1)$ time by table lookup. P uses $O(T^{1/2})$ time to activate processors $P(0), P(1), \ldots, P(2^{2B}-1)$ and builds up a Left-Shift Table and a Right-Shift Table of size 2^{2B} as in the proof of Lemma 3.3. Next, for $0 \le i < 2^{2B}$, each $P(i)$ extracts in $O(T^{1/2})$ time the first and second halves of its PID as follows. The first half is obtained by shifting the PID right B times using the Right-Shift Table. The second half is obtained by shifting the first half left B times and subtracting the (shifted) first half from the PID. $P(i)$ stores the first and second halves in $h1(i)$ and $h2(i)$, respectively. Hence the two tables $(h1(i))$ and $(h2(i))$ can be precomputed in $O(B) = O(T^{1/2})$ time.

4.4. Phase II. The data structure CONFIG has $O(T^{3/2})$ registers. In phase II, P initializes CONFIG in parallel using $O(\log T)$ time so that CONFIG contains *config*(0). For $m = 1, 2, \ldots, T^{1/2}$, the $T^{O(T^{1/2})}$ processors in group m do the following:

1. Each processor P_π in group m checks in $O(\log T)$ time whether π actually happens.

2. If so, compute *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ (stored in CONFIG) in $O(\log T)$ time and update CONFIG accordingly.

After $T^{1/2}$ updates, CONFIG contains *config*(T). P accepts if and only if statement q^* contains an ACCEPT instruction, where q^* is the statement number in *config*(T). Notice that in step 1, exactly one P_π determines that π happens. So in step 2, no write conflicts arise when updating CONFIG.

Next, we demonstrate that P_π can compute *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time, provided that π actually happens. In section 4.5, we prove that $O(\log T)$ time suffices to verify whether π actually happens. The preprocessing of section 4.3.3 yields the statement number at time $mT^{1/2}$ directly. For $(m-1)T^{1/2} < i - K - n \le mT^{1/2}$, the preprocessing of section 4.3.4 either gives the address of u_i directly or reduces the problem of finding the address of u_i to that of finding the content of u_0. It remains to explain how to determine the contents of the u_i's.

4.4.1. Computing contents of registers. We now explain how P_π computes the contents of the u_i's at time $mT^{1/2}$ from the contents of the u_i's at time $(m-1)T^{1/2}$. Suppose *config*$((m-1)T^{1/2})$ is available in CONFIG as described in section 4.2. Then $c(j,k)$ holds the kth B-bit block of $\langle u_j, (m-1)T^{1/2}\rangle$. Recall that

1. $\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle u_j, (m-1)T^{1/2}\rangle$;
2. in phase I, P_π dispatches processor P_{ij} to calculate C_{ij};
3. C_{ij} is a B-bit integer.

The product $C_{ij}\langle c(j,k)\rangle$ is thus a $2B$-bit integer. In phase II, the P_{ij}'s cooperate to compute $\langle u_i, mT^{1/2}\rangle$ in $O(\log T)$ time as follows. The P_{ij}'s use four multidimensional global memories $(g(i_1,i_2,i_3,i_4))$, $(g'(i_1,i_2,i_3,i_4))$, $(h(i_1,i_2,i_3))$, and $(h'(i_1,i_2,i_3))$. Let p be the PID of P_π. In $O(\log T)$ time, every P_{ij} activates $O(T)$ processors P'_k, where $0 \le k \le T+K+n$. Each P'_k multiplies C_{ij} with $\langle c(j,k)\rangle$ and puts the most and least significant B bits of the product in $g'(p,i,j,(k+1))$ and $g(p,i,j,k)$, respectively. By Lemma 3.3, we may assume that the multiplication requires $O(1)$ time. Extracting the most and least significant B bits also takes $O(1)$ time as discussed in section 4.3.8. Then

$$\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle c(j)\rangle, \tag{2}$$

$$\langle c(j)\rangle = \sum_{k=0}^{N-1} \langle c(j,k)\rangle 2^{kB}, \tag{3}$$

$$C_{ij}\langle c(j,k)\rangle = \langle g'(p,i,j,(k+1))\rangle 2^B + \langle g(p,i,j,k)\rangle. \tag{4}$$

From (2), (3), and (4),

$$\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n}\sum_{k=0}^{N} (\langle g'(p,i,j,k)\rangle + \langle g(p,i,j,k)\rangle)2^{kB}. \tag{5}$$

Next, P_π uses $O(\log T)$ time to deploy $O(T^{3/2})$ processors P'_{ik}, where $0 \le i \le T+K+n$ and $0 \le k \le N$. Each P'_{ik} computes the sum

$$\phi_{ik} = \sum_{j=0}^{T+K+n} (\langle g'(p,i,j,k)\rangle + \langle g(p,i,j,k)\rangle)$$

in $O(\log T)$ time (Lemma 3.1). The sum of $2(T+K+n+1)$ integers, each B bits long, is at most $B + \log(2(T+K+n+1)) \le 2B$ bits long. P'_{ik} extracts the most and least significant B bits of ϕ_{ik} and places them in $h'(p,i,(k+1))$ and $h(p,i,k)$, respectively. Therefore,

$$\sum_{j=0}^{T+K+n} (\langle g'(p,i,j,k)\rangle + \langle g(p,i,j,k)\rangle) = h'(p,i,(k+1))2^B + h(p,i,k). \tag{6}$$

Let $\psi_i = \sum_{k=0}^{N+1}\langle h(p,i,k)\rangle 2^{kB}$ and $\psi'_i = \sum_{k=0}^{N+1}\langle h'(p,i,k)\rangle 2^{kB}$. From (5) and (6),

$$\langle u_i, mT^{1/2}\rangle = \sum_{k=0}^{N+1} (\langle h'(p,i,k)\rangle + \langle h(p,i,k)\rangle)2^{kB} = \psi'_i + \psi_i. \tag{7}$$

Consider the carries into and out of the kth B-bit block when we add ψ and ψ' together. By (7), the kth block of $\langle u_i, mT^{1/2}\rangle$ is (roughly) $\langle h'(p,i,k)\rangle + \langle h(p,i,k)\rangle$,

except that we have to adjust for the carries into and out of the kth block. A carry into the block amounts to an increment by 1, whereas a carry out of the block is offset by subtracting 2^B. The value 2^B is precomputed during phase I in $O(B) = O(T^{1/2})$ time by repeated doubling. In section 4.4.2, we show that all block-to-block carries can be determined in $O(\log T)$ time. To update $c(i)$ with $\langle u_i, mT^{1/2}\rangle$, every P'_{ik} finds the kth block of $\langle u_i, mT^{1/2}\rangle$ (by adding $\langle h'(p,i,k)\rangle$ and $\langle h(p,i,k)\rangle$ and adjusting for the carries) and updates $c(i,k)$ accordingly. Hence P_π is able to compute *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time during phase II.

In the above discussion, we have presumed that all C_{ij}'s are positive. Strictly speaking, to calculate $\langle u_i, mT^{1/2}\rangle = \sum_{j=0}^{T+K+n} C_{ij}\langle c(j)\rangle$, we have to sum up the positive and the negative components separately using the above method, do a blockwise subtraction, and adjust for the block-to-block borrows. The calculation of the borrows is analogous to that of the carries.

4.4.2. Computing the carries. Consider adding two $O(T)$-bit integers together. By parallel-prefix computation [1, 11], it is possible to determine all the bit-to-bit carries in $O(\log T)$ time, provided that the individual bits of the integers are immediately accessible. In our case, however, the integers are represented in a blockwise instead of bitwise format. To apply the parallel-prefix technique, we formulate the computation of the block-to-block carries as a prefix-sum problem in a way slightly different from that in the bitwise case. The idea is to let a block take the place of a bit. Define a binary operation $\otimes$ on $\{\bar{g}, \bar{s}, \bar{p}\}$ as follows:

$$x \otimes y = \begin{cases} y & \text{if } y \neq \bar{p}, \\ x & \text{otherwise.} \end{cases} \tag{8}$$

It is routine to check that $\otimes$ is associative. For $0 \le k \le N+1$, let

$$x_k = \begin{cases} \bar{g} & \text{if } \langle h'(p,i,k)\rangle + \langle h(p,i,k)\rangle \ge 2^B, \\ \bar{p} & \text{if } \langle h'(p,i,k)\rangle + \langle h(p,i,k)\rangle = 2^B - 1, \\ \bar{s} & \text{otherwise.} \end{cases}$$

Intuitively, $x_k = \bar{g}$ if a carry is "generated" in the kth block; $x_k = \bar{p}$ if a carry is "propagated" through the kth block (i.e., there is a carry out of the kth block if and only if there is a carry into the kth block); and $x_k = \bar{s}$ if a carry is "stopped" in the kth block (i.e., no carry out of the kth block regardless of whether there is a carry into the kth block). Let $x_{-1} = \bar{s}$, and for $-1 \le k \le N+1$, let $y_k = x_{-1} \otimes x_0 \otimes x_1 \cdots \otimes x_k$. By (8), $y_k = x_{k'}$, where $k' \le k$ is maximum such that $x_{k'} \neq \bar{p}$. This implies $y_k = \bar{g}$ if and only if there is a carry out of the kth block. By parallel-prefix computation, we can determine all the y_k's, and hence all block-to-block carries, in $O(\log N) = O(\log T)$ time.

4.5. Verifying π. During phase I, P_π performs some additional preprocessing so that during phase II, P_π can decide in $O(\log T)$ time whether π actually happens. We first outline the verification process and then supply the details.

4.5.1. The outline. Now π specifies the behavior of R between time $(m-1)T^{1/2}$ and $mT^{1/2}$. We say that π "happens up to time t" if the behavior of R from time $(m-1)T^{1/2}$ to time t agrees with π. Similarly, we say that π "happens in step t" if the behavior of R from time $t-1$ to time t agrees with π. P_π uses $T^{1/2}$ processors P_t^*, where $(m-1)T^{1/2} \le t < mT^{1/2}$. Each P_t^* checks whether π happens in step $t+1$, assuming that π happens up to time t. Each P_t^* obtains a true or false answer. Clearly, π happens if and only if all these answers are true. P_π calculates the logical AND

of these $T^{1/2}$ answers in $O(\log T)$ time (Lemma 3.1) and decides whether π actually happens.

4.5.2. Preprocessing for verification. P_π activates all P_t^*'s in phase I using $O(\log T)$ time. Recall that in phase I, P_π performs some preprocessing based on the triple $\pi = (q, \beta, \sigma)$; if π happens, then this preprocessing enables P_π to compute *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time. During phase I, every P_t^* performs the analogous preprocessing using the triple $(q, \beta(t), \sigma(t))$, where $\beta(t)$ and $\sigma(t)$ are prefixes of β and σ respectively that define the behavior of R between time $(m-1)T^{1/2}$ and t. If π happens up to time t, then this preprocessing enables P_t^* to compute *config*(t) from *config*$((m-1)T^{1/2})$ in $O(\log T)$ time. As argued in section 4.3, this preprocessing takes $O(T^{1/2})$ time.

4.5.3. The actual verification. In section 4.4, we discussed how P_π computes *config*$(mT^{1/2})$ from *config*$((m-1)T^{1/2})$, provided that π actually happens. In an analogous manner, each P_t^* computes *config*(t) from *config*$((m-1)T^{1/2})$ during phase II, assuming that π actually happens up to time t. Using *config*(t), P_t^* verifies whether π happens in step $t+1$ in $O(\log T)$ time.

If π actually happens, then every P_t^* obtains a positive answer (true), and P_π deduces that π actually happens. Otherwise, let t' be maximum such that π happens up to time t'. Then $P_{t'}^*$ determines *config*(t') correctly and discovers that π does not happen in step $t'+1$. $P_{t'}^*$ answers false, and P_π infers that π does not happen. Note that for $t > t'$, the preprocessing of P_t^* yields nothing useful, and P_t^* cannot compute *config*(t) correctly. This does not concern us, however, since the negative answer of $P_{t'}^*$ renders other answers immaterial. We merely need to guarantee that P_t^* finishes its preprocessing within $O(T^{1/2})$ time and produces some answer within $O(\log T)$ time. This is readily accomplished by having each P_t^* count the number of steps it executes.

4.5.4. Verifying a single step. It remains to explain how P_t^* uses *config*(t) to check whether $\pi = (q, \beta, \sigma)$ happens in step $t+1$. Recall the following facts:

1. The integer q specifies the statement number of R at time $(m-1)T^{1/2}$.
2. We treat every static and dynamic instruction as a conditional jump, and β is a binary string of length $T^{1/2}$. For each dynamic instruction from step $(m-1)T^{1/2}+1$ to step $mT^{1/2}$, β stipulates whether the condition of the jump is true or false.
3. Every dynamic instruction is equivalent to an effective instruction, and σ gives the sequence of effective instruction from step $(m-1)T^{1/2}+1$ to step $mT^{1/2}$.
4. Every uninitialized register of R contains 0, and $r(K-1) = u_{K-1}$ contains 0 throughout the computation of R.

To decide whether π happens in step $t+1$, P_t^* performs the following checks:

1. *Check for* q. Let q' be the statement number in *config*(t). If $t = (m-1)T^{1/2}$, then P_t^* checks that $q = q'$.
2. *Check for* β. Let $s = t - (m-1)T^{1/2} + 1$. The sth bit of β specifies whether the condition is true or false for the dynamic instruction in step $t+1$. This dynamic instruction corresponds to the static instruction in statement q'. P_t^* checks that the sth bit of β is 1 if and only if statement q' indeed contains a JUMP instruction, and the condition of the jump is true, i.e., $\langle u_0, t\rangle \le \langle u_1, t\rangle$ according to *config*(t).
3. *Check for* σ. P_t^* checks that the effective instruction in step $t+1$ specified by σ is equivalent to the dynamic instruction in step $t+1$.

The first two checks need no further explanation. We supply the details of the third check below. In section 4.3.1, we showed that every dynamic instruction is equivalent to an effective instruction of the form $u_{i'} \leftarrow u_{j'}$, $u_{i'} \leftarrow u_{j'} + u_{k'}$, or $u_{i'} \leftarrow u_{j'} -$

$u_{k'}$ for some $0 \le i', j', k' \le T + K + n$. The dynamic instruction in step $t+1$ corresponds to the static instruction in statement q'. Consider the effective instruction in step $t+1$ specified by σ. P_t^* checks that the form of this effective instruction is "compatible" with the static instruction in statement q'. Table 2 shows the four categories of compatible instruction pairs. P_t^* performs some further checks according to the category of the compatible pair.

TABLE 2
Compatible effective and static instruction pairs.

Category	Effective Instruction	Static Instruction
1	$u_0 \leftarrow u_0$	ACCEPT, REJECT, and JUMP
2	$u_{i'} \leftarrow u_{j'}$	$r(i) \leftarrow r(j)$, $r(i) \leftarrow (r(0))$, and $(r(0)) \leftarrow r(j)$
3	$u_{i'} \leftarrow u_{j'} + u_{k'}$	$r(i) \leftarrow r(j) + r(k)$
4	$u_{i'} \leftarrow u_{j'} - u_{k'}$	$r(i) \leftarrow r(j) - r(k)$

Category 1. No further check is necessary.

Category 2. The static instruction in statement q' is either $r(i) \leftarrow r(j)$, $r(i) \leftarrow (r(0))$, or $(r(0)) \leftarrow r(j)$; σ stipulates that the effective instruction in step $t+1$ is $u_{i'} \leftarrow u_{j'}$. Let a_{w} be the address of the register that is written to in step $t+1$, and let a_{r} be the address of the register that is read from in step $t+1$. More precisely,

$$a_{\mathrm{w}} = \begin{cases} i & \text{if the static instruction in statement } q' \text{ is } r(i) \leftarrow r(j) \text{ or } r(i) \leftarrow (r(0)), \\ \langle r(0), t\rangle & \text{if the static instruction in statement } q' \text{ is } (r(0)) \leftarrow r(j), \end{cases}$$

$$a_{\mathrm{r}} = \begin{cases} j & \text{if the static instruction in statement } q' \text{ is } r(i) \leftarrow r(j) \text{ or } (r(0)) \leftarrow r(j), \\ \langle r(0), t\rangle & \text{if the static instruction in statement } q' \text{ is } r(i) \leftarrow (r(0)). \end{cases}$$

According to σ, R reads from $u_{j'}$ and writes to $u_{i'}$ in step $t+1$. P_t^* compares a_{r} with the addresses of all u_k's in *config*(t) in parallel. By definition, the addresses of all u_k's are distinct. If a_{r} equals the address of u_k for some k, then P_t^* checks that $j' = k$. Otherwise, R reads from an uninitialized register in step $t+1$; P_t^* checks that $j' = K - 1$.

If $i' = t + 1 + K + n$, then according to σ, a new register is written to in step $t+1$; P_t^* checks that a_{w} is different from the addresses of all u_k's in *config*(t). If $i' < t + 1 + K + n$, then according to σ, R writes to $u_{i'}$ in step $t+1$, but not for the first time; P_t^* checks that a_{w} is the address of $u_{i'}$ in *config*(t). If $i' > t + 1 + K + n$, then σ stipulates that in step $t+1$, R writes to $u_{i'}$, which by definition is the register first written to in step $i' - K - n > t+1$. The information contained in σ contradicts itself; P_t^* simply answers false.

P_t^* uses N processors (comparators) to compare two addresses in the blockwise format for equality. Each comparator checks for equality in a corresponding block and obtains a true or false answer; P_t^* calculates the logical AND of these answers in $O(\log N) = O(\log T)$ time (Lemma 3.1). To compare a_{r} and a_{w} against the addresses of all u_k's in parallel, P_t^* requires $O(NT) = O(T^{3/2})$ comparators. Observe that although these comparators are used in phase II, all comparators can be preactivated in phase I using $O(\log T)$ time. We will need this observation in section 4.6.

Categories 3 *and* 4. These cases are similar to those in Category 2.

Hence P_π can verify whether π actually happens in $O(\log T)$ time. This concludes the proof of Theorem 4.1.

4.6. Time–processor tradeoffs. In this section, we discuss how to reduce the number of processors used in the simulation at the expense of increasing the simulation time. The following theorem is a generalization of Theorem 4.1.

THEOREM 4.2. *Let $\rho > 1$. Every unit-cost RAM that runs in time T can be simulated by a CREW PRAM in time $O(\rho \log T + (T \log \rho)/\rho)$ with $T^{O(\rho)}$ processors.*

Proof. The proof of Theorem 4.2 is similar to that of Theorem 4.1. We explain how to modify the proof of Theorem 4.1 to establish Theorem 4.2. Instead of dividing each integer into $N = O(T^{1/2})$ blocks, we divide every $O(T)$-bit integer into $N' = O(\rho)$ blocks, each $O(T/\rho)$ bits long. We use T/ρ groups of processors. During phase I, group m performs the preprocessing based on the triple $(q'_m, \beta'_m, \sigma'_m)$, where q'_m is the statement number at time $(m-1)\rho$, β'_m is a binary string that encodes the outcomes of all condition jumps from time $(m-1)\rho$ to time $m\rho$, and σ'_m is the sequence of ρ effective instructions between time $(m-1)\rho$ and $m\rho$. In phase I, group m uses $O(\rho \log T)$ time to activate $T^{O(\rho)}$ processors to try all possible triples. Similar analysis as in section 4.3 reveals that the preprocessing takes $O(\rho)$ time. Again, the bottleneck in phase I is the activation of enough processors to try all triples, which takes $O(\rho \log T)$ time.

The content of each u_i at time $m\rho$ is a linear combination of the $\langle u_i, (m-1)\rho\rangle$'s. Let $\langle u_i, m\rho\rangle = \sum_j C'_{ij} \langle u_j, (m-1)\rho\rangle$. Observe that such a linear combination has at most $O(\rho)$ nonzero coefficients since all arithmetic operations between time $(m-1)\rho$ and $m\rho$ involve at most $O(\rho)$ registers. Let $J = \{j \mid C'_{ij} \neq 0\}$. In section 4.4.1, we described how to compute $\sum_{j=0}^{T+K+n} C_{ij} \langle u_j, (m-1)T^{1/2}\rangle$ in $O(\log N + \log(T + K + n)) = O(\log T)$ time. Using the same method, we can compute $\langle u_i, m\rho\rangle = \sum_{j \in J} C'_{ij} \langle u_j, (m-1)\rho\rangle$ in $O(\log N' + \log |J|) = O(\log \rho)$ time. Verification of the triple also takes $O(\log \rho)$ time. Note that the verification of the triple requires $O(T\rho)$ comparators, which can be preactivated in $O(\log T + \log \rho)$ time during phase I. The preprocessing thus enables group m to compute *config*$(m\rho)$ from *config*$((m-1)\rho)$ in $O(\log \rho)$ time during phase II.

In phase II, the PRAM P computes *config*(T) in $O((T \log \rho)/\rho)$ time as follows. For $m = 1, 2, \ldots, T/\rho$, group m computes *config*$(m\rho)$ from *config*$((m-1)\rho)$ in $O(\log \rho)$ time. Let q^* be the statement number in *config*(T). P accepts if and only if statement q^* contains an ACCEPT instruction. This simulation takes $O(\rho \log T + (T \log \rho)/\rho)$ time and uses $T^{O(\rho)}$ processors. ∎

5. Discussion.

5.1. Parallelism always helps. We have shown that we can always speed up a sequential computation on a unit-cost RAM by a CREW PRAM. We mentioned in section 1.1 that the unit-cost RAM is the most commonly used machine model for analyzing sequential algorithms. There are, however, other machine models of sequential computation, for example, the Turing machine, tree Turing machine, multidimensional Turing machine, and log-cost RAM. In a separate paper [12], we show that a sequential computation on each of these other models can also be sped up by a corresponding parallel machine model:

1. Every tree Turing machine that runs in time T can be simulated by an alternating Turing machine in time $O(T/\log T)$.

2. Every d-dimensional Turing machine that runs in time T can be simulated by an alternating Turing machine in time $O(T5^{d \log^* T}/\log T)$.

3. Every log-cost RAM that runs in time T can be simulated by an alternating log-cost RAM in time $O(T \log \log T/\log T)$.

We conclude that parallelism always helps us speed up a sequential computation.

5.2. Speedup using a polynomial number of processors. It is well known that the Turing machine enjoys the constant speedup theorem [26]: Let $\epsilon > 0$ and M be a Turing machine with time complexity T; then M can be simulated by another Turing machine in time $\epsilon T + n$. Hence efforts on speeding up the Turing machine

have focused on asymptotic speedup [4, 10, 16]. The unit-cost RAM, however, does not enjoy the constant speedup theorem [23]; that is, there exist an $\epsilon > 0$ and a unit-cost RAM R with time complexity T such that R cannot be simulated by any unit-cost RAM in time $\epsilon T + n$. Thus it is not trivial to speed up the computation of a unit-cost RAM by a constant factor. Theorem 4.2 shows that it is possible to speed up a unit-cost RAM by an arbitrary constant factor with a CREW PRAM using a polynomial number of processors.

5.3. Is result optimal? We have constructed a simulator that runs in time $O(T^{1/2} \log T)$. We do not know whether our result is optimal, but we believe that it is difficult to reduce the simulation time by more than a $\log T$ factor because this would imply improvements over some best-known results, as explained below. We would like to call the reader's attention to the following previously established results:

1. Every CREW PRAM that runs in time T can be simulated by a Turing machine in space $O(T^2)$ (Fortune and Wyllie [6]).

2. Every Turing machine that runs in time T can be simulated by a unit-cost RAM in time $O(T/\log T)$ (Hopcroft, Paul, and Valiant [10]).

3. Every Turing machine that runs in time T can be simulated by another Turing machine in space $O(T/\log T)$ (Hopcroft, Paul, and Valiant [9]).

4. Every Turing machine that runs in time T can be simulated by a CREW PRAM in time $O(T^{1/2})$ (Dymond and Tompa [4]).

These are the best-known results for the respective simulations. For our problem, namely, simulation of unit-cost RAMs by CREW PRAMs, reducing the simulation time to $o((T \log T)^{1/2})$, together with the first result of Hopcroft et al. above, implies an improvement over the result of Dymond and Tompa. By the same reasoning, if we manage to reduce the simulation time to $o(T^{1/2})$, then we can simulate every Turing machine with time complexity T by a CREW PRAM in time $o((T/\log T)^{1/2})$. It then follows from the above result of Fortune and Wyllie that for Turing machines, time T can be simulated in space $o(T/\log T)$, improving the second result of Hopcroft et al. above. This would be a significant breakthrough in simulating time by space for Turing machines.

Acknowledgments. Many thanks go to my advisor, Professor Michael Loui, for his many contributions to this paper; he proposed the problem, suggested generalizing Theorem 4.1 to Theorem 4.2, and provided useful references. His extensive comments significantly improved the disposition of this paper. Also, I would like to thank Professor Larry Ruzzo for bringing to my attention the work of Reif.

REFERENCES

[1] G. E. Blelloch, *Prefix sums and their applications*, in Synthesis of Parallel Algorithms, J. H. Reif, ed., Morgan Kaufmann, San Mateo, CA, 1993, pp. 35–60.

[2] A. Borodin and J. E. Hopcroft, *Routing, merging, and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130–145.

[3] S. A. Cook and R. A. Reckhow, *Time bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

[4] P. W. Dymond and M. Tompa, *Speedups of deterministic machines by synchronous parallel machines*, J. Comput. System Sci., 30 (1985), pp. 149–161.

[5] F. E. Fich, P. Ragde, and A. Wigderson, *Relations between concurrent-write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[6] S. Fortune and J. Wyllie, *Parallelism in random access machines*, in Proc. 10th Annual ACM Symposium on Theory of Computing, ACM, New York, 1978, pp. 114–118.

[7] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, UK, 1988.

[8] L. M. Goldschlager, *A universal interconnection pattern for parallel computers*, J. Assoc. Comput. Mach., 29 (1982), pp. 1073–1086.
[9] J. Hopcroft, W. Paul, and L. Valiant, *On time versus space*, J. Assoc. Comput. Mach., 24 (1977), pp. 332–337.
[10] J. E. Hopcroft, W. J. Paul, and L. G. Valiant, *On time versus space and related questions*, in Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1975, pp. 57–64.
[11] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.
[12] L. Mak, *Are parallel machines always faster than sequential machines*?, Technical report UILU-ENG-92-2236 (ACT 128), Coordinated Science Laboratory, University of Illinois at Urbana–Champaign, Urbana, IL, 1993.
[13] I. Parberry, *Parallel speedup of sequential machines: A defense of the parallel computation thesis*, ACM SIGACT News, 18 (1986), pp. 54–67.
[14] I. Parberry, *Parallel Complexity Theory*, John Wiley, New York, 1987.
[15] I. Parberry and G. Schnitger, *Parallel computation with threshold functions*, J. Comput. System Sci., 36 (1988), pp. 278–302.
[16] W. Paul and R. Reischuk, *On alternation* II, Acta Inform., 14 (1980), pp. 391–403.
[17] J. H. Reif, *On synchronous parallel computations with independent probabilistic choice*, SIAM J. Comput., 13 (1984), pp. 46–56.
[18] J. M. Robson, *Fast probabilistic RAM simulation of single tape Turing machine computations*, Inform. and Control, 63 (1984), pp. 67–87.
[19] J. M. Robson, *Random access machines with multi-dimensional memories*, Inform. Process. Lett., 34 (1990), pp. 265–266.
[20] J. M. Robson, *Deterministic simulation of a single tape Turing machine by a random access machine in sub-linear time*, Inform. and Comput., 99 (1992), pp. 109–121.
[21] Y. Shiloach and U. Vishkin, *Finding the maximum, merging, and sorting in parallel computation model*, J. Algorithms, 2 (1981), pp. 88–102.
[22] M. Snir, *On parallel searching*, SIAM J. Comput., 14 (1985), pp. 688–708.
[23] I. H. Sudborough and A. Zalcberg, *On families of languages defined by time-bounded random access machines*, SIAM J. Comput., 5 (1976), pp. 217–230.
[24] J. L. Trahan, M. C. Loui, and V. Ramachandran, *Multiplication, division, and shift instructions in parallel random access machines*, Theoret. Comput. Sci., 100 (1992), pp. 1–44.
[25] U. Vishkin, *Implementation of simultaneous memory address access in models that forbid it*, J. Algorithms, 4 (1983), pp. 45–50.
[26] C. K. Yap, *Theory of Complexity Classes*, Oxford University Press, Oxford, UK, to appear.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 173–187, February 1997

010

STOCHASTIC SCHEDULING WITH VARIABLE PROFILE AND PRECEDENCE CONSTRAINTS*

ZHEN LIU† AND ERIC SANLAVILLE‡

Abstract. In this paper, we consider the stochastic profile scheduling problem of a partially ordered set of tasks on uniform processors. The set of available processors varies in time. The running times of the tasks are independent random variables with exponential distributions. We obtain a sufficient condition under which a list policy stochastically minimizes the makespan within the class of preemptive policies. This result allows us to obtain a simple optimal policy when the partial order is an interval order, an in-forest, or an out-forest.

Key words. stochastic scheduling, profile scheduling, makespan, precedence constraint, interval order, in-forest, out-forest, uniform processors, stochastic ordering.

AMS subject classifications. Primary, 90B35, 68M20; Secondary, 90A80

PII. S0097539791218949

1. Introduction.

Consider the following scheduling problem. We are given a set of tasks to be run in a system consisting of uniform processors (i.e., processors having different speeds). The executions of these tasks must satisfy some precedence constraints which are described by a directed acyclic graph, referred to as the task graph. The processing requirements of the tasks are independent random variables with a common exponential distribution. The set of processors available to these tasks varies in time. The availability of the processors is referred to as the profile, and it can be arbitrary. The goal is to find preemptive schedules that stochastically minimize the makespan.

Our study of scheduling under variable profile is motivated by situations where processors are subject to failures and repairs. The failure and repair times are arbitrary. Another motivation is scheduling of multiprogrammed systems. In such a system, execution of tasks of a program may be preempted by tasks of higher-priority programs.

When the task graph is an in-forest and the profile is a constant set of two processors, Chandy and Reynolds [2] proved that the highest-level-first (HLF) policy minimizes the expected makespan. Here the level of a task is simply the distance from it to the root of the tree in which it appears. Bruno [1] subsequently showed that HLF stochastically minimizes the makespan when the system has two identical parallel processors. Pinedo and Weiss [14] extended this last result to the case where tasks at different levels may have different expected task running times. Frostig [7] further generalized the result of Pinedo and Weiss to include increasing likelihood ratio distributions for the task running times. Recently, Kulkari and Chimento [9] extended the result of [1] to the case of variable profile (with two identical parallel processors). When the number of identical parallel processors in the system is arbitrarily fixed and the task running times have a common exponential distribution, Papadimitriou

* Received by the editors September 4, 1991; accepted for publication (in revised form) April 25, 1995.
http://www.siam.org/journals/sicomp/26-1/21894.html
† INRIA Centre Sophia Antipolis, 2004 Route des Lucioles, B.P. 109, 06561 Valbonne, France (liu@sophia.inria.fr).
‡ Laboratoire LITP/IBP, Université Pierre et Marie Curie, 4 Place Jussieu, 75252 Paris cedex 05, France (erik@mustang.ibp.fr).

and Tsitsiklis [12] proved that HLF is asymptotically optimal as the number of tasks tends to infinity.

Coffman and Liu [3] investigated the stochastic scheduling of out-forest on identical parallel processors with constant profile. For a uniform out-forest where all subtrees are ordered by an embedding relation (see definition in section 4.3), they showed that an intuitive priority scheduling policy induced by the embedding relation, referred to as the most-successors (MS) policy in this paper, stochastically minimizes the makespan when there are two processors. If, in addition, the out-forest satisfies a uniform root-embedding constraint, then the greedy policy stochastically minimizes the makespan on an arbitrary number of processors.

Papadimitriou and Yannakakis [13] studied the deterministic scheduling of interval-ordered tasks. Under the assumptions of unit execution time and constant profile, they showed that for an arbitrary number of identical processors, the simple list scheduling induced by the interval order, still referred to as the MS policy in this paper, minimizes the makespan.

The notion of profile scheduling was first introduced by Ullman [16] and later by Garey et al. [8] in the complexity analysis of deterministic scheduling algorithms. Dolev and Warmuth [4, 5, 6] carried out various studies on the deterministic nonpreemptive profile scheduling with parallel identical processors. In such a case, the profile is simply the number of available processors at any time. When the tasks have unit execution time, Dolev and Warmuth obtained polynomial algorithms for specific profiles (e.g., zigzag profile, bounded profile, etc.) and specific task graphs (e.g., in-forest, out-forest, opposing forest, flat graph, etc.). Some of their results were extended to deterministic preemptive profile scheduling by Liu and Sanlaville [11].

In this paper, we investigate profile scheduling in the stochastic setting with uniform processors. The scheduling is allowed to be preemptive; the profile is arbitrary and may be unknown a priori. We obtain a sufficient condition under which a list-scheduling policy is optimal among preemptive policies, such that the makespan is stochastically minimized. This result allows us to prove the optimality of MS policy when the task graph has an interval-order structure, an in-forest structure, or a uniform out-forest structure.

The results concerning interval-ordered tasks are new, even for constant profile and parallel identical processors. The optimality of the MS policy extends the result of [9] for in-forests to uniform processors and the result of [3] for out-forests to variable profile and uniform processors.

Our paper is organized as follows. In section 2, the scheduling problem is described in detail, and some preliminaries are presented. In section 3, a sufficient condition for a list-scheduling policy to stochastically minimize the makespan is established. In section 4, this result is applied to MS policies for the stochastic minimization of makespan for interval-order task graphs, in-forests, and uniform out-forests. Concluding remarks are provided in section 5.

2. Problem description and preliminaries. A *task graph* $G = (V, E)$ is a directed acyclic graph, where $V = \{1, 2, \ldots, |V|\}$ is the set of vertices representing the tasks and $E \subset V \times V$ is the set of edges representing the *precedence constraints*: $(i, j) \in E$ implies that task i must complete execution before task j can start. Denote by $p(i)$ and $s(i)$ the sets of immediate *predecessors* and *successors* of $i \in V$, i.e.,

$$p(i) = \{j : (j, i) \in E\}, \qquad s(i) = \{j : (i, j) \in E\}.$$

A task without predecessors will be called *initial task*. Let $S(i)$ be the set of (not necessarily immediate) successors of $i \in V$, i.e.,

$$\forall i: \quad \text{if} \quad s(i) = \emptyset, \quad \text{then} \quad S(i) = \emptyset \quad \text{else} \quad S(i) = s(i) \bigcup \left(\bigcup_{j \in s(i)} S(j) \right).$$

When necessary, notation $S_G(i)$ is used to indicate that $S(i)$ is defined with respect to graph G.

Particular attention will be paid to the following three classes $\mathcal{C}_{i.o}$, $\mathcal{C}_{i.f}$, and $\mathcal{C}_{o.f}$ of task graphs.

Interval order $G \in \mathcal{C}_{i.o}$: Each vertex i corresponds to an interval b_i in the real line such that $(i, j) \in E$ if and only if $x \in b_i$ and $y \in b_j$ imply $x < y$.

In-forest $G \in \mathcal{C}_{i.f}$: Each vertex has at most one immediate successor: $|s(i)| \leq 1$, $i \in V$. A vertex $i \in V$ is called a leaf of in-forest G if $p(i) = \emptyset$. A vertex $i \in V$ is called a root of in-forest G if $s(i) = \emptyset$.

Out-forest $G \in \mathcal{C}_{o.f}$: Each vertex has at most one immediate predecessor: $|p(i)| \leq 1$, $i \in V$. A vertex $i \in V$ is called a leaf of out-forest G if $s(i) = \emptyset$. A vertex $i \in V$ is called a root of out-forest G if $p(i) = \emptyset$.

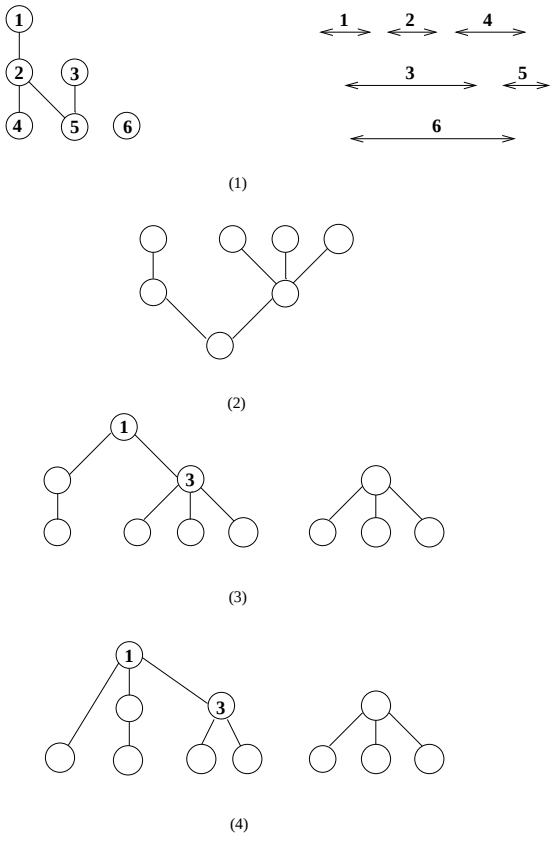


FIG. 1. *Examples of task graphs.*

Four graphs belonging to these classes are illustrated in Figure 1. Graph 1 is an

interval-order graph, with the associated collection of real intervals beside it. Graph 2 is an in-forest. Graphs 3 and 4 are out-forests.

Note that these classes of graphs have the following closure property: for all $\mathcal{C} \in \{\mathcal{C}_{i.o}, \mathcal{C}_{i.f}, \mathcal{C}_{o.f}\}$, if $G = (V, E) \in \mathcal{C}$, then $G - \{v\} \in \mathcal{C}$ for all $v \in V$ such that $p(v) = \emptyset$, where $G - \{v\}$ is the graph obtained by deleting vertex v and its adjacent edges. This *closure property* (by deletion) will be used in establishing our results.

The processing requirements of the tasks are independent and identically distributed (i.i.d.) random variables with a common exponential distribution of a constant parameter, say 1. The running time of a task is the processing requirement of the task divided by the speed of the processor on which the task is running.

There are $K \geq 1$ *uniform processors*, indexed by $1, 2, \ldots, K$. The speed of processor k, $1 \leq k \leq K$, is denoted by α_k. The running times of tasks on processor k are thus i.i.d. random variables with exponential distribution of parameter α_k. We assume, by convention, that $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_K$.

The set of processors available to these tasks varies in time due to, e.g., failures of the processors or executions of higher-priority tasks. The availability of the processors is referred to as the *profile*, and is specified by the sequence $\{a_n, M_n\}_{n=1}^{\infty}$, where the random variables $0 = a_1 < a_2 < \cdots < a_n < \cdots$ are the time epochs where the profile is changed and M_n, $n \geq 1$, is a random set whose elements are the indices of the processors available during the time interval $[a_n, a_{n+1})$. The profile $\{a_n, M_n\}_{n=1}^{\infty}$ is assumed to be independent of the running times of the tasks. Without loss of generality, we assume that for all $n \geq 1$, $M_n \neq \emptyset$. We will also assume that the profile is not changed infinitely often during any finite time interval: for all $x \in I\!R^+$, there is some finite $n \geq 1$ such that $a_n > x$. The profile is said to be *bounded* by $p \in I\!N_+ \equiv \{1, 2, \ldots\}$ if for all $n \in I\!N_+$, $|M_n| \leq p$.

The *scheduling policies* decide when an *enabled task*, i.e., an unfinished task all of whose predecessors have finished, should be assigned to an available processor. At any time, a task can be assigned to at most one processor, and a processor can execute at most one task. Throughout this paper, we assume that the scheduling policies are preemptive. We assume that the scheduler has no information on the samples of the (remaining) processing requirements of the tasks. Let Ψ denote the class of such policies. For any $\pi \in \Psi$, denote by $\pi(G)$ the makespan of the partially ordered set of tasks G, i.e., the maximum of the completion times of the tasks in G.

The goal of the paper is to find policies in Ψ that *stochastically minimize* the makespan of G. A policy π_o is said to be optimal within a class $\mathcal{C}$ if for any policy $\pi \in \Psi$, $\pi_o(G) \leq_{st} \pi(G)$ for all $G \in \mathcal{C}$, where the symbol $\leq_{st}$ refers to the standard stochastic inequality. Random variable $X \in I\!R$ is *stochastically smaller* than random variable $Y \in I\!R$, denoted by $X \leq_{st} Y$, if and only if for any increasing function $f : I\!R \to I\!R$, the inequality $E[f(X)] \leq E[f(Y)]$ holds, provided the expectations exist.

The proofs of our main results will use a coupling argument based on the following well-known result due to Strassen, where $=_{st}$ denotes equality in distribution.

LEMMA 2.1 (Strassen [15]). *Two random variables $X, Y \in I\!R$ satisfy $X \leq_{st} Y$ if and only if there exist two random variables $\hat{X}$ and $\hat{Y}$ defined on a common probability space such that $X =_{st} \hat{X}$, $Y =_{st} \hat{Y}$, and $\hat{X} \leq \hat{Y}$ almost surely (a.s.).*

In order to simplify the proofs of the main results in the paper, we make some restrictions on the class of policies Ψ.

Observe first that due to the memoryless property of exponential distributions, at any time, the distribution of the remaining running time of a task assigned to

processor k is still exponential with parameter α_k, $1 \leq k \leq K$. If we represent the state of the system by the set of available processors, the remaining task graph, and the distributions of the remaining running times of the tasks, then the state does not change between the instants of task completions and of profile modifications. Therefore, we can without loss of generality confine ourselves to the class of policies where preemptions and new task assignments occur only at the instants of task completions and profile modifications. These instants are referred to as the *decision epochs*. Hence we assume that all the policies in Ψ make their scheduling decisions at these time instants only.

A policy is *idling* if it allows a processor to remain idle when there is an initial task waiting for execution. It is easy to see that an optimal policy should never be idling since the distributions of the task running times have infinite support and preemptions are allowed. Furthermore, an optimal policy should always use the *fastest* available processors. (See [10] for a complete proof of these basic properties.) Throughout this paper, we will assume that all the policies in Ψ are nonidling and use the fastest available processors at all decision epochs.

3. Optimal list-scheduling policies. We will pay particular attention to a class of simple scheduling policies, referred to as *list schedules*. A policy λ is called a list schedule if for any task graph $G = (V, E)$, a *priority list* is defined on the set of tasks $V = (v_1, v_2, \ldots, v_n)$, given by, e.g., $v_1 >_\lambda v_2 >_\lambda \cdots >_\lambda v_n$, where $v_i >_\lambda v_j$ means that task v_i has higher priority than v_j in the list. At any decision time epoch, policy λ assigns the enabled tasks with the highest priorities to the fastest available processors. Clearly, all list schedules are in the class Ψ and satisfy the properties mentioned at the end of last section. Note that the priority list can be changed dynamically, i.e., for a given policy, the relative priority order between two tasks can be changed when a task is removed from the graph.

Let $\mathcal{C}$ be a class of graphs which is closed under deletion. The class $\mathcal{C}$ will be said to be $\lambda(p)$-*monotonic* for some list schedule λ and some $p \in I\!N_+$ if for any $G \in \mathcal{C}$ and any initial tasks $u, v \in G$, relation $u >_\lambda v$ implies that for any profile bounded by p,

$$\lambda(G - \{u\}) \leq_{st} \lambda(G - \{v\}),$$

where recall that $\lambda(G)$ denotes the makespan of G under schedule λ.

Observe that according to the definition, if $\mathcal{C}$ is $\lambda(p)$-monotonic, then it is $\lambda(p-1)$-monotonic, Trivially, any class of graphs is $\lambda(1)$-monotonic for any list schedule λ. However, it will be seen in the next section that few classes are $\lambda(2)$-monotonic for some list schedule λ.

THEOREM 3.1. *Let $\mathcal{C}$ be a class of graphs closed under deletion and $\lambda(p)$-monotonic for some list schedule λ and some $p \in I\!N_+$. Then for any $G \in \mathcal{C}$, policy λ stochastically minimizes the makespan of G under any profile $\{a_n, M_n\}_{n=1}^{\infty}$ bounded by p:*

$$\forall \pi \in \Psi : \quad \lambda(G) \leq_{st} \pi(G). \tag{1}$$

The proof of the theorem needs the following lemma.

LEMMA 3.2. *Let $\mathcal{C}$ be a class of graphs which is closed under deletion and is $\lambda(p)$-monotonic for some list schedule λ and some $p \in I\!N_+$. Let $G \in \mathcal{C}$ be a task graph, $\pi, \rho \in \Psi$ two policies of G which follow the priority list of λ all the time except at the first decision epoch. At the first decision epoch, policy π assigns tasks*

$v_1, v_2, \ldots, v_k$ to the available processors $q_1, q_2, \ldots, q_k$, respectively, with, by assumption, $v_1 >_\lambda v_2 >_\lambda \cdots >_\lambda v_k$, whereas policy ρ assigns tasks $v_1, v_2, \ldots, v_k$ to the processors $q_{\chi(1)}, q_{\chi(2)}, \ldots, q_{\chi(k)}$, respectively, where χ is the permutation on $\{1, 2, \ldots, k\}$ such that $q_{\chi(1)} < q_{\chi(2)} < \cdots < q_{\chi(k)}$. Then

$$\rho(G) \leq_{st} \pi(G).$$

Proof. If $q_1 < q_2 < \cdots < q_k$, then ρ is identical to π so that the assertion trivially holds. Assume that there are integers $1 \leq i < j \leq k$ such that $q_i > q_j$. Let χ_1 be the permutation on $\{1, 2, \ldots, k\}$ defined as follows:

$$\chi_1(i) = j, \quad \chi_1(j) = i, \quad \chi_1(n) = n \quad \forall n \in \{1, 2, \ldots, k\} - \{i, j\}.$$

Let $\rho_1 \in \Psi$ be a policy which follows the priority list of λ all the time except at the first decision epoch. At the first decision epoch, policy ρ_1 assigns tasks $v_1, v_2, \ldots, v_k$ to the processors $q_{\chi_1(1)}, q_{\chi_1(2)}, \ldots, q_{\chi_1(k)}$, respectively. We will show that

$$\rho_1(G) \leq_{st} \pi(G). \tag{2}$$

We couple the running times on processors $q_1, \ldots, q_k$ in such a way that under both policies π and ρ_1, the running time on processor q_n starting from time 0 is τ_n, $1 \leq n \leq k$. In such a coupled model, the second decision epoch occurs at the same time under both policies. Let this time epoch be fixed, referred to as d_2,

$$d_2 = \min(a_2, \tau_1, \ldots, \tau_k).$$

If this decision epoch corresponds to a profile modification, i.e., $d_2 = a_2$, then $\rho_1(G) =_{st} \pi(G)$.

Now assume that the second decision epoch corresponds to a task completion. It is easy to see that for any increasing function $f : \mathbb{R} \to \mathbb{R}$,

$$E\left[f\left(\pi(G) - d_2\right)\right] = \frac{1}{\beta} \sum_{n=1}^{k} \alpha_{q_n} E\left[f\left(\lambda(G - \{v_n\})\right)\right], \tag{3}$$

$$E\left[f\left(\rho_1(G) - d_2\right)\right] = \frac{1}{\beta} \sum_{n=1}^{k} \alpha_{q_{\chi_1(n)}} E\left[f\left(\lambda(G - \{v_n\})\right)\right], \tag{4}$$

where $\beta = \sum_{n=1}^{k} \alpha_{q_n}$.

Since $\mathcal{C}$ is $\lambda(p)$-monotonic, we have that $\lambda(G - \{v_i\}) \leq_{st} \lambda(G - \{v_j\})$ so that

$$E\left[f\left(\lambda(G - \{v_i\})\right)\right] \leq E\left[f\left(\lambda(G - \{v_j\})\right)\right].$$

Thus the fact that $q_i > q_j$ (so that $\alpha_{q_i} \leq \alpha_{q_j}$) implies that

$$\begin{aligned} &\alpha_{q_j} E\left[f\left(\lambda(G - \{v_i\})\right)\right] + \alpha_{q_i} E\left[f\left(\lambda(G - \{v_j\})\right)\right] \\ &\quad \leq \alpha_{q_i} E\left[f\left(\lambda(G - \{v_i\})\right)\right] + \alpha_{q_j} E\left[f\left(\lambda(G - \{v_j\})\right)\right]. \end{aligned}$$

Hence for any increasing function $f : \mathbb{R} \to \mathbb{R}$,

$$E\left[f\left(\rho_1(G) - d_2\right)\right] \leq E\left[f\left(\pi(G) - d_2\right)\right],$$

which implies

$$\rho_1(G) - d_2 \leq_{st} \pi(G) - d_2.$$

Unconditioning on d_2 in the above relation yields (2).

Consider now policy ρ_1. If $\chi_1 = \chi$, then we are done. Otherwise, there are integers $1 \le i' < j' \le k$ such that $q_{\chi_1(i')} > q_{\chi_1(j')}$. Let χ_2 be the permutation on $\{1, 2, \ldots, k\}$ defined as follows:

$$\chi_2(i') = \chi_1(j'), \quad \chi_2(j') = \chi_1(i'), \quad \chi_2(n) = \chi_1(n) \quad \forall n \in \{1, 2, \ldots, k\} - \{i', j'\}.$$

Let $\rho_2 \in \Psi$ be a policy which follows the priority list of λ all the time except at the first decision epoch. At the first decision epoch, policy ρ_2 assigns tasks $v_1, v_2, \ldots, v_k$ to the processors $q_{\chi_2(1)}, q_{\chi_2(2)}, \ldots, q_{\chi_2(k)}$, respectively. As above, we can show that

$$\rho_2(G) \le_{st} \rho_1(G).$$

Repeating this procedure for at most $k(k-1)/2$ times finally yields policy ρ such that

$$\rho(G) \le_{st} \cdots \le_{st} \rho_2(G) \le_{st} \rho_1(G) \le_{st} \pi(G). \qquad \square$$

Proof of Theorem 3.1. We prove by induction on n that for any $G = (V, E) \in \mathcal{C}$ such that $|V| \le n$, the relation

$$\lambda(G) \le_{st} \pi(G) \tag{5}$$

holds for any profile.

If G is a singleton, i.e., $|V| = 1$, then (5) is trivial since both policies λ and π assign the task to the fastest available processor.

Assume that for some $n \ge 1$, relation (5) holds for all G such that $|V| \le n$. Now consider the task graphs $G \in \mathcal{C}$ such that $|V| = n + 1$.

Fix the task graph $G \in \mathcal{C}$, the profile $\{a_n, M_n\}_{n=1}^{\infty}$, and the policy $\pi \in \Psi$. Denote by $\{d_n\}_{n=1}^{\infty}$ the sequence of decision epochs of π for finishing tasks in G, with $d_1 = 0$. Without loss of generality, we assume that d_2 corresponds to a task completion. The other case can be analyzed similarly.

Since all policies of Ψ are nonidling and use the fastest available processors, the numbers of tasks assigned for execution at the first decision epoch are the same under policies λ and π. At the first decision epoch, policy π assigns initial tasks $v_1, v_2, \ldots, v_k$ to the available processors $q_1, q_2, \ldots, q_k$, respectively, where $v_1 >_\lambda v_2 >_\lambda \cdots >_\lambda v_k$, whereas policy λ assigns initial tasks $u_1, u_2, \ldots, u_k$ to the processors $q_{\chi(1)}, q_{\chi(2)}, \ldots, q_{\chi(k)}$, respectively, where $u_1 >_\lambda u_2 >_\lambda \cdots >_\lambda u_k$ and χ is the permutation on $\{1, 2, \ldots, k\}$ such that $q_{\chi(1)} < q_{\chi(2)} < \cdots < q_{\chi(k)}$.

We construct an intermediate policy ρ which follows the priority list of λ all the time except at the first decision epoch. At the first decision epoch, policy ρ assigns tasks $u_1, u_2, \ldots, u_k$ to the processors $q_1, q_2, \ldots, q_k$, respectively.

According to Lemma 3.2, $\lambda(G) \le_{st} \rho(G)$. We show below that

$$\rho(G) \le_{st} \pi(G), \tag{6}$$

which will complete the proof.

Under the assumption that d_2 corresponds to a task completion, we have that for any increasing function $f : \mathbb{R} \to \mathbb{R}$,

$$E\left[f\left(\rho(G) - d_2\right)\right] = \frac{1}{\beta} \sum_{i=1}^{k} \alpha_{q_i} E\left[f\left(\lambda(G - \{u_i\})\right)\right], \tag{7}$$

$$E\left[f\left(\pi(G) - d_2\right)\right] = \frac{1}{\beta} \sum_{i=1}^{k} \alpha_{q_i} E\left[f\left(\pi(G - \{v_i\})\right)\right], \tag{8}$$

where $\beta = \sum_{i=1}^{k} \alpha_{q_i}$.

The definition of the list schedule λ implies that $u_i >_\lambda v_i$, $1 \le i \le k$, where relation $>_\lambda$ is understood to be antisymmetric. Therefore, the $\lambda(p)$-monotonicity of G implies that

$$\lambda(G - \{u_i\}) \le_{st} \lambda(G - \{v_i\}),$$

which, together with the inductive assumption, implies

$$\lambda(G - \{u_i\}) \le_{st} \lambda(G - \{v_i\}) \le_{st} \pi(G - \{v_i\}).$$

Therefore (cf. (7) and (8)), $E[f(\rho(G) - d_2)] \le E[f(\pi(G) - d_2)]$, which readily implies (6). □

4. Optimality of MS policies. A class of intuitively good list schedules is the MS policies, where the priority of a task is defined by the number of (not necessarily immediate) successors, i.e., $|S(u)| > |S(v)|$ implies $u \prec_{MS} v$. The difference between MS policies is the way that ties are broken (the ways of assigning priorities to tasks having the same number of successors). In this section, we will show that these policies are optimal for some special classes of task graphs, i.e., interval-order graphs, in-forests, and uniform out-forests.

In order to prove these optimality properties, we need to compare task graphs, which is done using the following majorization. Let $G^1 = (V^1, E^1)$ and $G^2 = (V^2, E^2)$ be two task graphs, with the vertices of $V^1 = \{v_1^1, \ldots, v_{n_1}^1\}$ and $V^2 = \{v_1^2, \ldots, v_{n_2}^2\}$ ordered according to the number of successors: $g^1 = |S_{G^1}(v_1^1)| \ge |S_{G^1}(v_2^1)| \ge \cdots \ge |S_{G^1}(v_{n_1}^1)| = 0$ and $g^2 = |S_{G^2}(v_1^2)| \ge |S_{G^2}(v_2^2)| \ge \cdots \ge |S_{G^2}(v_{n_2}^2)| = 0$. We say that G^1 is *majorized* by G^2, denoted by $G^1 \prec_s G^2$, if

$$n_1 \le n_2 \quad \text{and} \quad \forall i,\ 1 \le i \le n_1 : \quad |S_{G^1}(v_i^1)| \le |S_{G^2}(v_i^2)|.$$

In our proofs, the following equivalent definition will also be used. Let V^1 and V^2 be partitioned into sets $T_0^1, T_1^1, T_2^1, \ldots, T_{g^1}^1$ and $T_0^2, T_1^2, T_2^2, \ldots, T_{g^2}^2$, where $T_k^j = \{i \in V^j, |S_{G^j}(i)| = k\}$, $j = 1, 2$, $1 \le k \le g^j$. In words, T_k^j is the set of vertices having k successors in graph G^j. As an example, for the interval-order graph in Figure 1, $T_3 = \{1\}$, $T_2 = \{2\}$, $T_1 = \{3\}$, and $T_0 = \{4, 5, 6\}$. When the graph is an in-forest, T_k is the set of vertices at level k. Now the equivalent definition is given as follows. Graph G^1 is majorized by G^2 if

$$g^1 \le g^2 \quad \text{and} \quad \forall i,\ 0 \le i \le g^1 : \quad \sum_{k=i}^{g^1} |T_k^1| \le \sum_{k=i}^{g^2} |T_k^2|.$$

4.1. Stochastic profile scheduling of interval-order graphs. Let $G = (V, E) \in \mathcal{C}_{i.o}$ be an interval-order graph. Note that except for a possible set of isolated vertices (i.e., vertices without predecessors and without successors), an interval-order graph is connected.

An equivalent definition of interval-order graphs (cf. [13]) is that for all $i, j \in V$, either $S(i) \subseteq S(j)$ or $S(j) \subseteq S(i)$. An immediate consequence of this definition is that if $|S(i)| \le |S(j)|$, then $S(i) \subseteq S(j)$, and consequently, if $|S(i)| = |S(j)|$, then $S(i) = S(j)$. Thus any subset $T_k = \{i \in V, |S(i)| = k\}$ of V contains vertices whose sets of successors are identical (and of cardinality k).

THEOREM 4.1. *For any profile* $\{a_n, M_n\}_{n=1}^{\infty}$ *and any set of interval-ordered tasks* $G \in \mathcal{C}_{i.o}$,

$$\forall \pi \in \Psi: \quad MS(G) \leq_{st} \pi(G). \tag{9}$$

Proof. In view of Theorem 3.1, we only need to show that $\mathcal{C}_{i.o}$ is $MS(p)$-monotonic for any positive integer p, i.e., for any two initial tasks u and v of G, if $|S(u)| \geq |S(v)|$, then

$$MS(G - \{u\}) \leq_{st} MS(G - \{v\}). \tag{10}$$

Let $G^1 = (V^1, E^1) = G - \{u\}$ and $G^2 = (V^2, E^2) = G - \{v\}$. We will show that there is a common probability space such that

$$MS(G^1) \leq MS(G^2) \quad \text{a.s.} \tag{11}$$

Further, applying Strassen's theorem (cf. Lemma 2.1) yields (10).

Owing to the memoryless property of the exponential distributions, we can consider a coupled processing model where all processors $1, \ldots, K$, whenever they are available, are continually executing tasks. When a completion occurs and there is no task assigned to that processor, it corresponds to the completion of a fictitious task. When a task is assigned to a processor, it is assigned a running time equal to the remainder of the running time already underway at that processor. Thus if tasks $u \in G^1$ and $v \in G^2$ are assigned to the same processor at some time, they have the same (remaining) running time.

Denote by $\{c_n\}_{n=1}^{\infty}$ the (increasing) sequence of completion times of the tasks in G^1 and G^2 at the available processors under the MS policy. Let $\{d_n\}_{n=1}^{\infty}$ be the superposition of the sequences of the decision epochs of MS for G^1 and G^2 in such a probability space. More specifically, $\{d_n\}_{n=1}^{\infty}$ is the superposition of the sequences of profile modification times $\{a_n\}_{n=1}^{\infty}$ and of the task completion times $\{c_n\}_{n=1}^{\infty}$. Clearly, $d_1 = a_1 = 0$.

For $j = 1, 2$ and $n \geq 1$, let $G^j(n) = (V^j(n), E^j(n))$ be the remaining graph of G^j at time d_n under MS in the coupled model. Let $g = \max_{i \in V} |S(i)|$. Denote $T_k^j(n) = \{i \in V^j(n), |S(i)| = k\}$, $j = 1, 2$, $0 \leq k \leq g$, and $n \geq 1$. We show that for all $n \geq 1$,

$$G^1(n) \prec_s G^2(n), \tag{12}$$

which immediately implies (11).

Relation (12) is proved by induction on n. For $n = 1$, it is trivial that

$$G^1(1) = G^1 \prec_s G^2 = G^2(1).$$

Assume that (12) holds for some $n \geq 1$.

Let there be m available processors at time d_n^+. Without loss of generality, we assume that the processors $1, 2, \ldots, m$ are available. Recall that, by convention, these processors are ordered by their speed: $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_m$.

Under the MS policy, the initial tasks with the largest sets of successors are assigned to the fastest processors. Let $u_1, u_2, \ldots, u_{m_1}$ be the tasks of $G^1(n)$ that are assigned to the processors $1, 2, \ldots, m_1$, respectively, under the MS policy. Similarly, let $v_1, v_2, \ldots, v_{m_2}$ be the tasks of $G^2(n)$ that are assigned to the processors $1, 2, \ldots, m_2$, respectively, under the MS policy. By definition, $m \geq \max(m_1, m_2)$, and

$$S(u_1) \supseteq S(u_2) \supseteq \cdots \supseteq S(u_{m_1}), \qquad S(v_1) \supseteq S(v_2) \supseteq \cdots \supseteq S(v_{m_2}).$$

If $G^1(n)$ is empty, then (12) trivially holds for $n+1$. If the time epoch d_{n+1} corresponds to a profile modification, then

$$G^1(n+1) = G^1(n) \prec_s G^2(n) = G^2(n+1)$$

so that relation (12) holds for $n+1$. Now assume that $G^1(n)$ is not empty and that d_{n+1} corresponds to a completion at some processor, say processor h, $1 \le h \le \max(m_1, m_2)$. Let u_h, if any, belong to $T_a^1(n)$, and v_h, if any, belong to $T_b^2(n)$. Tasks u_h and/or v_h finish at time d_{n+1}. There are three cases to be investigated.

Case 1: $m_1 > m_2$ *and* $m_2 + 1 \le h \le m_1$. In this case, only task u_h is finished. It is easy to see that

$$\forall i,\ 0 \le i \le g: \quad \sum_{k=i}^{g} |T_k^1(n+1)| \le \sum_{k=i}^{g} |T_k^1(n)| \le \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|$$

so that relation (12) holds for $n+1$.

Case 2: $h \le \min(m_1, m_2)$. In this case, both tasks u_h and v_h are finished. There are two subcases:

Case 2.1: $b \le a$. It is simple that

$$\begin{aligned}
\forall i,\ a+1 \le i \le g: \quad & \sum_{k=i}^{g} |T_k^1(n+1)| = \sum_{k=i}^{g} |T_k^1(n)| \le \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|, \\
\forall i,\ b+1 \le i \le a: \quad & \sum_{k=i}^{g} |T_k^1(n+1)| = -1 + \sum_{k=i}^{g} |T_k^1(n)| \le -1 + \sum_{k=i}^{g} |T_k^2(n)| \\
& = -1 + \sum_{k=i}^{g} |T_k^2(n+1)| \le \sum_{k=i}^{g} |T_k^2(n+1)|, \\
\forall i,\ 0 \le i \le b: \quad & \sum_{k=i}^{g} |T_k^1(n+1)| = -1 + \sum_{k=i}^{g} |T_k^1(n)| \\
& \le -1 + \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|.
\end{aligned}$$

Therefore, relation (12) holds for $n+1$.

Case 2.2: $b > a$. Observe first that

$$\begin{aligned}
\forall i,\ b+1 \le i \le g: \quad & \sum_{k=i}^{g} |T_k^1(n+1)| = \sum_{k=i}^{g} |T_k^1(n)| \le \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|, \\
\forall i,\ 0 \le i \le a: \quad & \sum_{k=i}^{g} |T_k^1(n+1)| = -1 + \sum_{k=i}^{g} |T_k^1(n)| \\
& \le -1 + \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|.
\end{aligned}$$

We examine the case $a+1 \le i \le b$. Let $0 \le i_1, i_2 \le g$ be the integers defined as follows:

$$\begin{aligned}
T_g^1(n) = \cdots = T_{i_1+1}^1(n) = \emptyset, & T_{i_1}^1(n) \ne \emptyset, \\
T_g^2(n) = \cdots = T_{i_2+1}^2(n) = \emptyset, & T_{i_2}^2(n) \ne \emptyset.
\end{aligned}$$

Note that i_1 and i_2 are uniquely defined. Under the inductive assumption, one has that

$$\sum_{k=i_1}^{g} |T_k^2(n)| \geq \sum_{k=i_1}^{g} |T_k^1(n)| > 0.$$

Thus $i_1 \leq i_2$.

For $0 \leq i, k \leq g$, let $n_k(i) = |T_k \cap S(T_i)|$, where $S(T_i) \equiv S(w)$ for some $w \in T_i$. Note that $n_k(i) = 0$ if $k \geq i$. Note also that $n_k(i) \leq n_k(i+1)$. According to the definitions, $n_k(i_j)$ is the number of noninitial tasks of $T_k^j(n)$ in $G^j(n)$, $j = 1, 2$, $1 \leq k \leq g$.

For all i, $a+1 \leq i \leq b$, we have $u_h \notin T_g \cup \cdots \cup T_i$ so that there are at most $h-1$ initial tasks $u_1, u_2, \ldots, u_{h-1}$ in $T_g^1(n) \cup \cdots \cup T_i^1(n)$. Thus

$$\sum_{k=i}^{g} |T_k^1(n)| \leq h - 1 + \sum_{k=i}^{g} n_k(i_1).$$

On the other hand, $v_h \in T_b$ implies that the h initial tasks $v_1, v_2, \ldots, v_h$ are in $T_g^2(n) \cup \cdots \cup T_i^2(n)$. Hence

$$\sum_{k=i}^{g} |T_k^2(n)| \geq h + \sum_{k=i}^{g} n_k(i_2).$$

The two inequalities above together with the fact that $n_k(i_1) \leq n_k(i_2)$ imply that

$$\sum_{k=i}^{g} |T_k^1(n)| \leq -1 + \sum_{k=i}^{g} |T_k^2(n)|.$$

Therefore, for all i, $a+1 \leq i \leq b$,

$$\sum_{k=i}^{g} |T_k^1(n+1)| = \sum_{k=i}^{g} |T_k^1(n)| \leq -1 + \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|,$$

which completes the proof of (12) for $n+1$ in Case 2.2.

Case 3: $m_1 < m_2$ *and* $m_1 + 1 \leq h \leq m_2$. In this case, only task v_h finishes.

It is clear that for all $b+1 \leq i \leq g$,

$$\sum_{k=i}^{g} |T_k^1(n+1)| = \sum_{k=i}^{g} |T_k^1(n)| \leq \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|.$$

Since $m \geq m_2 \geq m_1 + 1$, and since only m_1 tasks of $G^1(n)$ are assigned to the available processors, we obtain that for all $i \leq b$,

$$\sum_{k=i}^{g} |T_k^1(n)| \leq m_1 + \sum_{k=i}^{g} n_k(i_1). \tag{13}$$

On the other hand, $v_h \in T_b$ implies that the h initial tasks $v_1, v_2, \ldots, v_h$ are in $T_g^2(n) \cup \cdots \cup T_i^2(n)$. Hence for all $i \leq b$,

$$\sum_{k=i}^{g} |T_k^2(n)| \geq h + \sum_{k=i}^{g} n_k(i_2) \geq m_1 + 1 + \sum_{k=i}^{g} n_k(i_2) \geq m_1 + 1 + \sum_{k=i}^{g} n_k(i_1). \tag{14}$$

Inequalities (13) and (14) imply that

$$\sum_{k=i}^{g} |T_k^1(n)| \le -1 + \sum_{k=i}^{g} |T_k^2(n)|.$$

Therefore, for all $i \le b$,

$$\sum_{k=i}^{g} |T_k^1(n+1)| = \sum_{k=i}^{g} |T_k^1(n)| \le -1 + \sum_{k=i}^{g} |T_k^2(n)| = \sum_{k=i}^{g} |T_k^2(n+1)|$$

so that (12) holds for $n+1$ in Case 3.

Therefore, by induction, relation (12) holds for all $n \ge 1$. Consequently, $MS(G^1) \le MS(G^2)$ a.s. in that probability space. □

4.2. Stochastic profile scheduling of in-forests. Let $G = (V, E) \in \mathcal{C}_{i.f}$ be an in-forest. A task of in-forest G is an initial task if and only if it is a leaf of G. For any task, the number of its successors is equal to its level in the in-forest, i.e., the distance from it to the root of the tree in which it appears. (The level of the roots is zero by convention.) Thus the MS policy coincides with the HLF policy.

THEOREM 4.2. *For any profile* $\{a_n, M_n\}_{n=1}^{\infty}$ *bounded by* 2 *and for any in-forest* $G \in \mathcal{C}_{i.f}$,

$$\forall \pi \in \Psi : \quad MS(G) \le_{st} \pi(G). \tag{15}$$

The proof of the theorem is similar to that of Theorem 4.1, using the fact that $\mathcal{C}_{if}$ is $MS(2)$-monotonic, and is omitted here. The reader is referred to [10] for a detailed proof.

Note that the above theorem generalizes the result of Kulkari and Chimento [9] to uniform processors (with a different scheme of proof). Note also that such a result holds only when $|M_n| \le 2$. Simple counterexamples can be found when there are three processors (see [2]).

4.3. Stochastic profile scheduling of out-forests. Let $G = (V, E) \in \mathcal{C}_{o.f}$ be an out-forest. A task of out-forest G is an initial task if and only if it is a root of G. Vertex $v \in V$ and all its successors is a subtree of G, denoted by $T_G(v)$ or simply $T(v)$ when there is no ambiguity.

In general, MS policies are not optimal within the class of out-forests $\mathcal{C}_{o.f}$. Counterexamples are provided in [3]. However, we will show that within the classes of *uniform* and *r-uniform* out-forests (introduced in Coffman and Liu [3]), a policy is optimal if and only if it is MS.

Let $T_1, T_2 \in \mathcal{C}_{o.f}$ be two out-trees. The out-tree T_2 is said to *embed* the out-tree T_1, or T_1 is *embedded* in T_2, denoted by $T_2 \succ_e T_1$ or $T_1 \prec_e T_2$, if T_1 is isomorphic to a subgraph of T_2. Formally, T_2 embeds T_1 if there exists an injective function f from T_1 into T_2 such that $\forall u, v \in T_1$, $v \in s(u)$ implies $f(v) \in s(f(u))$. The function f is called an *embedding function*.

Let r_1 and r_2 be the roots of the out-trees T_1 and T_2, respectively. If $T_2 \succ_e T_1$ and if there is an embedding function f such that $f(r_1) = r_2$, then f is a *root-embedding function*, and we write $T_2 \succ_r T_1$ or $T_1 \prec_r T_2$.

An out-forest $G \in \mathcal{C}_{o.f}$ is said to be *uniform* (respectively, *r-uniform*) if all of its subtrees $\{T(v),\ v \in G\}$ can be ordered by the embedding (respectively, root-embedding) relation. The class of uniform (respectively, r-uniform) out-forests is

denoted by $\mathcal{C}_{u.o.f}$ (respectively, $\mathcal{C}_{r.o.f}$). It is clear that $\mathcal{C}_{r.o.f} \subset \mathcal{C}_{u.o.f} \subset \mathcal{C}_{o.f}$. In Figure 1, graph 3 is a uniform out-forest but not a r-uniform out-forest. Graph 4 is a r-uniform out-forest.

The embedding relation is extended to uniform out-forests as follows. Let $G^1 = (V^1, E^1)$ and $G^2 = (V^2, E^2)$ be two uniform out-forests. Denote by $T_G(v)$ the subtree of out-forest G composed of v and all the successors of v in G. Assume that the vertices of G^1 and G^2 are indexed in such a way that

$$T_{G^1}(1) \succ_e T_{G^1}(2) \succ_e \cdots \succ_e T_{G^1}(|V^1|),$$

$$T_{G^2}(1) \succ_e T_{G^2}(2) \succ_e \cdots \succ_e T_{G^2}(|V^2|).$$

Out-forest G^1 is embedded in G^2, referred to as $G^1 \prec_e G^2$, if and only if

$$|V^1| \leq |V^2|, \quad \text{and} \quad \forall i,\ 1 \leq i \leq |V^1| : \quad T_{G^1}(i) \prec_e T_{G^2}(i).$$

Similarly, $G^1 \prec_r G^2$ if and only if

$$T_{G^1}(1) \succ_r T_{G^1}(2) \succ_r \cdots \succ_r T_{G^1}(|V^1|),$$

$$T_{G^2}(1) \succ_r T_{G^2}(2) \succ_r \cdots \succ_r T_{G^2}(|V^2|),$$

$$|V^1| \leq |V^2|, \quad \text{and} \quad \forall i,\ 1 \leq i \leq |V^1| : \quad T_{G^1}(i) \prec_r T_{G^2}(i).$$

We will show that MS policies are optimal for uniform out-forests and profiles bounded by 2, as well as for r-uniform out-forests and arbitrary profiles. In order to establish these optimality properties, we first prove that $\mathcal{C}_{u.o.f}$ (resp. $\mathcal{C}_{r.o.f}$) is $MS(2)$-monotonic (resp. $MS(p)$-monotonic for any positive integer p).

Observe that for any two roots u and v of a uniform out-forest $G \in \mathcal{C}_{u.o.f}$ (resp. r-uniform out-forest $G \in \mathcal{C}_{r.o.f}$), $T(u) \succ_e T(v)$ (resp. $T(u) \succ_r T(v)$) if and only if $|S(u)| \geq |S(v)|$. Thus for any two subgraphs G^1 and G^2 of G which are obtained from deleting initial tasks of G, $G^1 \prec_e G^2$ (resp. $G^1 \prec_r G^2$) if and only if $G^1 \prec_s G^2$. This last property allows us to use the arguments of Coffman and Liu [3] for the establishment of the following MS-monotonicities.

LEMMA 4.3. *Let $G \in \mathcal{C}_{r.o.f}$ be an arbitrary r-uniform forest and let u and v be two roots of G such that $|S(u)| \geq |S(v)|$. Then for any profile $\{a_n, M_n\}_{n=1}^{\infty}$ bounded by 2,*

$$MS(G - \{u\}) \leq_{st} MS(G - \{v\}).$$

The proof of the above lemma is analogous to the proof of Theorem 1 of Coffman and Liu [3]. Although the proof of Theorem 1 in [3] was given for constant profile and parallel identical processors, the fact that both graphs $G - \{u\}$ and $G - \{v\}$ are scheduled by an MS policy allows us to use the same argument. The detailed proof is left to the interested reader.

LEMMA 4.4. *Let $G \in \mathcal{C}_{r.o.f}$ be an arbitrary r-uniform forest and let u and v be two roots of G such that $|S(u)| \geq |S(v)|$. Then for any profile $\{a_n, M_n\}_{n=1}^{\infty}$,*

$$MS(G - \{u\}) \leq_{st} MS(G - \{v\}).$$

The assertion of the lemma can be shown by mimicking the proof of Theorem 2 of Coffman and Liu [3]. The detailed proof is omitted.

Now applying Theorem 3.1 yields the following result.

THEOREM 4.5. *For any profile* $\{a_n, M_n\}_{n=1}^{\infty}$ *bounded by* 2*, and for any uniform out-forest* $G \in \mathcal{C}_{u.o.f}$*,*

$$\forall \pi \in \Psi : \quad MS(G) \leq_{st} \pi(G). \tag{16}$$

THEOREM 4.6. *For any profile* $\{a_n, M_n\}_{n=1}^{\infty}$ *and any r-uniform out-forest* $G \in \mathcal{C}_{r.o.f}$*,*

$$\forall \pi \in \Psi : \quad MS(G) \leq_{st} \pi(G). \tag{17}$$

5. Concluding remarks. We have considered the scheduling problem for the stochastic minimization of the makespan of task graphs under a variable profile. Under the assumption that task running times are independent random variables with exponential distributions, we have established a general condition for a list-scheduling policy to stochastically minimize the makespan. This result has allowed us to show the optimality of MS policies when the partial order is an interval order, an in-forest, or an out-forest.

We can further show that, except in the degenerate case where there is a single available processor all the time, MS is the only optimal policy for the stochastic minimization of the makespan. The reader is referred to [10] for details.

All results of the paper hold if the speed of the processors, as well as their availability, is allowed to vary. The task assignment will change whenever the speed ratio between two processors is reversed. Such an extension allows one to analyze systems with processor sharing among different jobs (i.e., sets of tasks).

Acknowledgments. We would like to thank the referees for various constructive comments which helped to improve both the contents and the presentation of the paper.

REFERENCES

[1] J. BRUNO, *On scheduling tasks with exponential service times and in-tree precedence constraints*, Acta Inform., 22 (1985), pp. 139–148.

[2] K. M. CHANDY AND P. F. REYNOLDS, *Scheduling partially ordered tasks with probabilistic execution times*, Oper. System Rev., 9 (1975), pp. 169–177.

[3] E. G. COFFMAN AND Z. LIU, *On the optimal stochastic scheduling of out-forests*, Oper. Res., 40 (1992), pp S67–S75.

[4] D. DOLEV AND M. K. WARMUTH, *Scheduling precedence graphs of bounded height*, J. Algorithms, 5 (1984), pp. 48–59.

[5] D. DOLEV AND M. K. WARMUTH, *Scheduling flat graphs*, SIAM J. Comput., 14 (1985), pp. 638–657.

[6] D. DOLEV AND M. K. WARMUTH, *Profile scheduling of opposing forests and level orders*, SIAM J. Algebraic Discrete Meth., 6 (1985), pp. 665–687.

[7] E. FROSTIG, *A stochastic scheduling problem with intree precedence constraints*, Oper. Res., 36 (1988), pp. 937–943.

[8] M. R. GAREY, D. S. JOHNSON, R. E. TARJAN, AND M. YANAKAKIS, *Scheduling opposite forests*, SIAM J. Algebraic Discrete Meth., 4 (1983), pp. 72–93.

[9] V. G. KULKARI AND P. F. CHIMENTO, JR., *Optimal scheduling of exponential tasks with intree precedence constraints on two parallel processors subject to failure and repair*, Oper. Res., 40 (1992), pp. S263–S271.

[10] Z. LIU AND E. SANLAVILLE, *Stochastic scheduling with variable profile and precedence constraints*, Research Report 1525, INRIA, Valbonne, France, 1991.

[11] Z. Liu and E. Sanlaville, *Preemptive scheduling with variable profile, precedence constraints and due dates*, Research Report 1622, INRIA, Valbonne, France, 1992; Discrete Appl. Math., 58 (1995), pp. 253–280.

[12] C. H. Papadimitriou and J. N. Tsitsiklis, *On stochastic scheduling with in-tree precedence constraints*, SIAM J. Comput., 16 (1987), pp. 1–6.

[13] C. H. Papadimitriou and M. Yannakakis, *Scheduling interval-ordered tasks*, SIAM J. Comput., 8 (1979), pp. 405–409.

[14] M. Pinedo and G. Weiss, *Scheduling jobs with exponentially distributed processing times and intree precedence constraints on two parallel machines*, Oper. Res., 33 (1985), pp. 1381–1388.

[15] V. Strassen, *The existence of probability measures with given marginals*, Ann. Math. Stat., 36 (1965), pp. 423–439.

[16] J. D. Ullman, *NP-complete scheduling problems*, J. Comput. System Sci., 10 (1975), pp. 384–393.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 188–209, February 1997

011

ON BOUNDED QUERIES AND APPROXIMATION*

RICHARD CHANG†, WILLIAM I. GASARCH‡, AND CARSTEN LUND§

Abstract. This paper investigates the computational complexity of approximating several NP-optimization problems using the number of queries to an NP oracle as a complexity measure. The results show a tradeoff between the closeness of the approximation and the number of queries required. For an approximation factor $k(n)$, $\log\log_{k(n)} n$ queries to an NP oracle can be used to approximate the maximum clique size of a graph within a factor of $k(n)$. However, this approximation cannot be achieved using fewer than $\log\log_{k(n)} n - c$ queries to *any oracle* unless P = NP, where c is a constant that does not depend on k. These results hold for approximation factors $k(n) \geq 2$ that belong to a class of functions which includes any integer constant function, $\log n$, $\log^a n$, and $n^{1/a}$. Similar results are obtained for Graph Coloring, Set Cover, and other NP-optimization problems.

Key words. bounded queries, approximation algorithm, NP-completeness, maximum clique, chromatic number, set cover

AMS subject classifications. 68Q15, 03D15

PII. S0097539794266481

1. Introduction.

The approximability of NP-optimization problems is a central theme both in the study of algorithms and in computational complexity theory. Most NP-optimization problems have decision versions that are NP-complete and are hence equivalent to each other as decision problems. However, the approximability of the optimization problems may vary greatly. For some NP-optimization problems, there are efficient algorithms that find good approximate solutions. For others, no such algorithm can exist unless some standard intractability assumption is violated (e.g., P = NP or the polynomial hierarchy (PH) collapses). Recently, Arora et al. [3] showed that the problem of finding the largest clique in a graph is in the latter category. Following a series of breakthrough results [4, 5, 18, 26, 29], they showed that there exists a constant ϵ such that no deterministic polynomial-time algorithm can approximate the maximum clique size $\omega(G)$ of a graph G with n vertices within a factor of n^ϵ unless P = NP. While this result strongly suggests that no efficient algorithm can find good approximations to the maximum-clique problem, it does not resolve all of the questions about the computational complexity of approximating the maximum clique size of a graph. In particular, it is not clear what computational resources are sufficient and/or necessary to compute an approximation of the maximum clique size using any of the traditional resource-bounded measures (e.g., time, space, random bits, and alternation).

In this paper, we use the number of queries to an NP-complete oracle as a complexity measure. Krentel [24] used this measure to show that the maximum clique size

*Received by the editors April 21, 1994; accepted for publication (in revised form) April 25, 1995. A preliminary version of this paper appeared in *Proc. 34th IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 547–556.

http://www.siam.org/journals/sicomp/26-1/26648.html

†Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, Baltimore, MD 21228 (chang@umbc.edu). The research of this author was supported in part by NSF research grant CCR-9309137 and by the University of Maryland Institute for Advanced Computer Studies.

‡Department of Computer Science and University of Maryland Institute for Advanced Computer Studies, University of Maryland College Park, College Park, MD 20742 (gasarch@cs.umd.edu). The research of this author was supported in part by NSF research grant CCR-9020079.

§AT&T Research, 600 Mountain Avenue, Murray Hill, NJ, 07974 (lund@ research.att.com).

TABLE 1.1
Upper and lower bounds for approximating the maximum clique size.

Factor	Upper bound	Lower bound (unless P = NP)
2	$\log\log n$	$\log\log n - \log 1/\epsilon$
k	$\log\log n - \log\log k$	$\log\log n - \log\log k - \log 1/\epsilon$
$\log^a n$	$\log\log n - \log\log\log^a n$	$\log\log n - \log\log\log^a n - \log 1/\epsilon$
$n^{1/a}$	$\log a$	$\log a - \log 1/\epsilon$

is complete for polynomial-time functions which use only $O(\log n)$ queries, denoted $\mathrm{PF}^{\mathrm{NP}[O(\log n)]}$. Since Krentel's original work, many connections between bounded query classes and standard complexity classes have been discovered [1, 2, 6, 7, 13, 14, 20, 23, 32, 33]. In many circumstances, these results show that one cannot decrease the number of queries needed to solve a problem by even a single query unless P = NP or PH collapses. For example, Hoene and Nickelsen [21] showed that to determine how many of the formulas in $F_1, \ldots, F_r$ are satisfiable, $\lceil \log(r+1) \rceil$ queries are both sufficient and necessary (unless P = NP). A naïve binary search can determine the number of satisfiable formulas using $\lceil \log(r+1) \rceil$ queries to the NP oracle. The number of queries needed is $\lceil \log(r+1) \rceil$ rather than $\lceil \log r \rceil$ because there are $r+1$ possible answers ranging from 0 to r. A tree-pruning technique shows that no polynomial-time machine can determine the number of satisfiable formulas using one fewer query to any oracle unless P = NP. Thus the algorithm which uses the fewest queries is simply the naïve binary search algorithm. In this paper, we show that for several NP-optimization problems, binary search is also the approximation algorithm that uses the fewest queries. In the first parts of the paper, we will focus on the complexity of approximating the size of the maximum clique in a graph.

In order to state the results in this paper correctly, we need to be more precise with the term "approximation." Let $\omega(G)$ denote the size of the maximum clique in the graph G. We say that a number x is an approximation of $\omega(G)$ within a factor of $k(n)$ if $\omega(G)/k(n) \leq x \leq \omega(G)$. Our results show a tradeoff between the closeness of the approximation and the number of queries needed to solve the approximation problem—finding closer approximations requires more queries.

For example we can approximate $\omega(G)$ within a factor of 2 using only $\log\log n$ queries to NP, where n is the number of vertices in the graph G. In contrast, computing $\omega(G)$ exactly can be done with $\log n$ queries and requires $\Omega(\log n)$ queries (unless P = NP) [24]. Moreover, we show that no function using fewer than $(\log\log n) - \log 1/\epsilon$ queries to *any oracle* can approximate $\omega(G)$ within a factor 2 unless P = NP. (Here ϵ is the constant given in Corollary 3 of [3].) In general, our results show that for any "nice" approximation factor $k(n) \geq 2$, $\omega(G)$ can be approximated within a factor of $k(n)$ using $\log\log_{k(n)} n$ queries but not with fewer than $\log\log_{k(n)} n - \log 1/\epsilon$ queries to any oracle unless P = NP. In Corollary 3.3, we show that the difference, $\log 1/\epsilon$, between the upper and lower bounds has a natural interpretation. Table 1.1 summarizes our results for some common approximation factors.

We make a few observations about these results. First, since ϵ is a constant, for a large enough constant k, $\log\log k$ would exceed $\log 1/\epsilon$. Hence for this k, the upper bound on approximating $\omega(G)$ within a factor of k will be strictly less than the lower bound for approximating within a factor of 2. Hence for this large k, the problem of approximating $\omega(G)$ within a factor k has strictly lower complexity in terms of the number of queries than approximating within a factor of 2 unless P = NP. Similarly,

approximating within a factor of $\log n$ has a lower complexity than approximating within any constant; and approximating within a factor of $n^{1/k}$ has an even lower complexity. We believe that these are the first results which show a tradeoff between complexity and closeness of approximation. In contrast, Garey and Johnson [19] showed that if $\omega(G)$ can be approximated within a constant factor in P, then it can be approximated within *any* constant factor in P. While these are not contradictory theorems, they certainly have very different flavors.

In the next section, we state the lemmas, definitions, and notations needed to prove the results. In section 3, we show that a uniform binary search routine provides the upper bounds we have mentioned. To prove the lower bound results, we start in section 4.2 with a simple proof which shows that no function in $\mathrm{PF}^{X[1]}$ for any oracle X can approximate $\omega(G)$ within a factor of 2 unless P = NP. We go on to the proof of the general lower-bound results. In section 5, we discuss how these results extend to other NP optimization problems (e.g., Chromatic Number). In section 6, we ponder the value of the constant ϵ. Finally, in section 7, we combine our techniques with the results of Lund and Yannakakis [27] on the hardness of approximating the Set Cover problem to derive upper and lower bounds on the query complexity of approximating the minimum set-cover size.

2. Preliminaries.

DEFINITION 2.1. *For every graph G, let $|G|$ denote the number of vertices in the graph and let $\omega(G)$ denote the size of the largest clique in G. We say that a function $A(G)$ approximates the maximum clique size within a factor of $k(n)$ if for all graphs G with n vertices,*

$$\omega(G)/k(n) \leq A(G) \leq \omega(G).$$

Some papers use the alternative condition that $\omega(G)/k(n) \leq A(G) \leq k(n) \cdot \omega(G)$, but we find it unintuitive to consider $A(G)$ an approximation of $\omega(G)$ if there does not exist a clique of size $A(G)$ in G. However, the results in this paper still hold under the alternative definition.

To prove the lower bounds, we need the result of Arora et al. [3], which showed that there exists an $\epsilon > 0$ such that the maximum clique size cannot be approximated within a factor of n^ϵ in deterministic polynomial time unless P = NP, where n is the number of vertices in the graph. Their construction yields the next lemma.

LEMMA 2.2 (see [3]). *There exist constants s, b, and d, $0 < s < b < d$, such that given a Boolean formula F with t variables, we can construct in polynomial time a graph G with $m = t^d$ vertices, where*

$$F \in \mathrm{SAT} \Longrightarrow \omega(G) = t^b \quad \textit{and}$$
$$F \notin \mathrm{SAT} \Longrightarrow \omega(G) < t^s.$$

The constants b, s, and d will be fixed for the remainder of the paper. Of particular interest is the ratio $(b-s)/d$ because it is equal to the ϵ mentioned before Lemma 2.2. To prove the intractability of n^ϵ approximation, first assume that we have a polynomial-time algorithm to approximate $\omega(G)$ within n^ϵ. Take any Boolean formula F, and let t be the number of variables in F. Construct G as described in Lemma 2.2. Use the polynomial-time algorithm to obtain an approximation x of $\omega(G)$. Since $|G| = t^d$, an n^ϵ approximation is within a factor of $(t^d)^\epsilon = t^{b-s}$. Therefore, if $F \in \mathrm{SAT}$, the algorithm must guarantee that $x \geq t^b/t^{b-s} = t^s$. On the other hand, if $F \notin \mathrm{SAT}$, then $x < t^s$. Thus $F \in \mathrm{SAT} \iff x \geq t^s$.

DEFINITION 2.3. *Let* $\mathrm{PF}^{X[q(n)]}$ *be the class of functions computed by polynomial-time oracle Turing machines which ask at most* $q(n)$ *queries to the oracle* X*. Since the queries are adaptive, the query strings may depend on answers to previous queries.*

3. Upper bounds. We first examine the upper bounds on the complexity of approximating NP-optimization problems in terms of bounded query classes. The general idea is to use each query to SAT to narrow down the range where the optimal solution exists. For example, for a graph G with n nodes, $1 \le \omega(G) \le n$. We can compute $\omega(G)$ exactly using binary search and $\log n$ queries of the form: "Is $\omega(G)$ greater than x?"

On the other hand, to approximate $\omega(G)$ within a factor of 2, we only need to find a number x such that $x \le \omega(G) \le 2x$. Thus we first partition the numbers from 1 to n into the intervals

$$[1,2],\ (2,4],\ (4,8],\ldots,\left(2^{\lceil \log n\rceil -1}, 2^{\lceil \log n\rceil}\right].$$

If $\omega(G)$ is in the interval $(2^i, 2^{i+1}]$, then we can simply use 2^i as a factor-2 approximation of $\omega(G)$. Thus our strategy is to use binary search on the left endpoints of the intervals to determine which interval contains $\omega(G)$. Since there are only $\lceil \log n\rceil$ intervals, we only need $\lceil \log \lceil \log n\rceil\rceil$ queries to SAT to perform this binary search. In the general case, if we want to find an approximation of $\omega(G)$ that is within a factor of $k(n)$, there will be $\lceil \log_{k(n)} n\rceil$ intervals of the form $\left(k(n)^i, k(n)^{i+1}\right]$ and binary search would use $\lceil \log \lceil \log_{k(n)} n\rceil\rceil$ queries to SAT. Thus, we have the following lemma.

LEMMA 3.1. *Let* $k(n)$ *be a polynomial-time-computable function such that* $1 < k(n) < n$*. Then there exists a function in* $\mathrm{PF}^{\mathrm{SAT}[\lceil \log \lceil \log_{k(n)} n\rceil\rceil]}$ *which approximates* $\omega(G)$ *within a factor of* $k(n)$ *for all graphs* G *with* n *vertices.*

The lemma is stated for the maximum-clique-size problem, but it obviously holds for any NP-optimization problem where the solution is an integer ranging from 1 to n. Note that it does not matter if $k(n)$ is not an integer because checking whether $\omega(G)$ is greater than x is still an NP question when x is a fractional number. Once we have determined that $\omega(G)$ is contained in the interval $(k(n)^i, k(n)^{i+1}]$, we can output $\lceil k(n)^i\rceil$ as an approximation to $\omega(G)$. Also, if we drop the ceilings from our notation, then we can derive more readable upper bounds on the complexity of approximating $\omega(G)$ for some common approximation factors (see Table 1.1).

The binary search strategy used to find the approximation to $\omega(G)$ may seem naïve. However, we shall see later that the upper bounds differ from the lower bounds by at most an additive constant. In any case, we can improve the upper bounds if there exists a polynomial-time algorithm which gives an approximate solution within a factor of $f(n)$. Then our strategy is to first use the polynomial-time algorithm to obtain an approximation x. We know that the solution is between x and $x \cdot f(n)$. Now, to find an approximation that is within a factor of $k(n)$, we divide the numbers from x to $x \cdot f(n)$ into intervals:

$$[x, xk(n)],\ \left(xk(n), xk(n)^2\right],\ldots,\left(x\frac{f(n)}{k(n)}, xf(n)\right].$$

In this case, the number of intervals is $\lceil \log_{k(n)} f(n)\rceil$, and we have the following lemma.

LEMMA 3.2. *Let* $k(n)$ *be a polynomial-time-computable function such that* $1 < k(n) < n$*. Suppose that there exists a polynomial-time algorithm which approximates*

$\omega(G)$ within a factor of $f(n)$. Then there exists a function in $\mathrm{PF}^{\mathrm{SAT}[\lceil \log \lceil \log_{k(n)} f(n) \rceil \rceil]}$ *which approximates $\omega(G)$ within a factor of $k(n)$ for all graphs G with n vertices.*

Again, we note that this lemma applies to any NP-optimization problem whose solutions range from 1 to n. This lemma may seem somewhat useless since the best known polynomial-time algorithm can only approximate the size of the maximum clique within a factor of $O(n/(\log n)^2)$ [10]. If we were to use the new strategy outlined above, we would reduce the number of queries needed to find a factor-2 approximation of $\omega(G)$ to $\log(\log n - 2 \log \log n)$, which would save us at most one query for various values of n. However, the following corollary of the lemma does allow us to gauge the quality of our lower bound results.

COROLLARY 3.3. *If no function in* $\mathrm{PF}^{\mathrm{SAT}[\log \log_{k(n)} n - \log 1/\delta]}$ *approximates $\omega(G)$ within a factor of $k(n)$, then no polynomial-time algorithm can approximate $\omega(G)$ within a factor of n^δ.*

Corollary 3.3 gives us a natural interpretation of the difference between the upper bound of $\log \log_{k(n)} n$ in Lemma 3.1 and the lower bound of $\log \log_{k(n)} - \log 1/\epsilon$ (Theorem 4.8). This difference of $\log 1/\epsilon$ reflects the fact that we do not know if there exists a polynomial-time algorithm which approximates $\omega(G)$ within a factor of n^δ for $\epsilon < \delta < 1$. Thus an improvement of either the upper or the lower bound is possible.

Moreover, the observations about Lemma 3.2 are most useful when they are applied to NP-optimization problems such as Set Cover. In the Set Cover problem, we are given a finite collection C of subsets of $\{1, \ldots, n\}$ and asked to find the size of the smallest subcollection $C' \subseteq C$ that covers $\{1, \ldots, n\}$. Since the size of the smallest set cover *can* be approximated within a factor $\ln(n) + 1$ [22, 25], we have the following lemma.

LEMMA 3.4. *Let $k(n)$ be a polynomial-time-computable function such that $1 < k(n) < n$. Then there exists a function in* $\mathrm{PF}^{\mathrm{SAT}[\lceil \log \lceil \log_{k(n)} (\ln n + 1) \rceil \rceil]}$ *which approximates the size of the minimum set cover within a factor of $k(n)$.*

Now, by comparing Lemma 3.1 against Lemma 3.4, we can obtain a quantitative difference between the complexity of approximating $\omega(G)$ and the complexity of approximating the size of the minimum set cover—not just a qualitative difference. For example, if we are allowed to make $\log \log(\ln n + 1)$ queries to SAT, then we can approximate the minimum set cover within a factor of 2. However, using the same number of queries, we can only approximate $\omega(G)$ within a factor of $n^{1/\log(\ln n + 1)}$. Thus we can conclude that approximating the set cover within a factor of 2 has about the same complexity as approximating $\omega(G)$ within a factor of $n^{1/\log(\ln n + 1)}$. Such a comparison is only possible by taking a quantitative view of the complexity of approximations in terms of the number of queries.

Note that the existence of a "good" polynomial-time approximation algorithm for a problem has a greater effect on the complexity of approximating the problem than on the complexity of finding the exact solution. For example, suppose that we have some NP-optimization problem where the solution ranges from 1 to n. Without the help of an approximation algorithm, we would need $\log n$ queries to find the exact solution and $\log \log n$ queries to find a factor-2 approximation. Now suppose that we are given a polynomial-time algorithm that guarantees a factor-4 approximation. To find the exact solution we would still need $\log(n - n/4) = O(\log n)$ queries. However, we only need one query ($\log \log 4 = 1$) to approximate within a factor of 2.

The upper bounds proven in this section use a naïve binary search strategy to determine an interval that contains the optimal solution. One might suspect that a more clever algorithm could use substantially fewer queries to the oracle. In the rest

of the paper, we show that unless some intractability assumption is violated (e.g., P = NP and RP = NP), no polynomial-time algorithm can reduce the number of queries by more than an additive constant. These results give us relative lower bounds on the number of queries needed to approximate the maximum clique size of a graph. We are also able to extend these techniques to determine the query complexity of approximating the chromatic number of a graph and the minimum set cover.

4. Lower bounds.

4.1. Promise problems and clique arithmetic. To prove our lower-bound results, we need to introduce some more definitions and notations.

DEFINITION 4.1. *Let* $\#^{\mathrm{SAT}}_{r(n)}(F_1, \ldots, F_{r(n)})$ *be the number of formulas in* $\{F_1, \ldots, F_{r(n)}\}$ *which are satisfiable. In other words,* $\#^{\mathrm{SAT}}_{r(n)}(F_1, \ldots, F_{r(n)}) = |\{F_1, \ldots, F_{r(n)}\} \cap \mathrm{SAT}|$.

DEFINITION 4.2. *Let* $r(t)$ *be a polynomially bounded polynomial-time function. We define* $\mathcal{P}_{r(t)}$ *to be the following promise problem. Given a sequence of Boolean formulas* $F_1, \ldots, F_{r(t)}$*, where each* F_i *has* t *variables, and the promise that for all* i*,* $2 \le i \le r(t)$*,* $F_i \in \mathrm{SAT}$ *implies that* $F_{i-1} \in \mathrm{SAT}$*, output* $\#^{\mathrm{SAT}}_{r(t)}(F_1, \ldots, F_{r(t)})$.

Technically, the size of the input to the promise problem is $|F_1, \ldots, F_{r(t)}|$. This size is $O(tr(t))$ when the Boolean formulas are restricted to ones where each variable occurs only a constant number of times. To simplify our notation, in the following lemma, we count the queries as a function of t rather than $|F_1, \ldots, F_{r(t)}|$. The following lemma provides a tight lower bound on the complexity of $\mathcal{P}_r$ and can be proven using the self-reducibility of SAT and a tree-pruning technique [21]. We include the proof for the sake of completeness.

LEMMA 4.3. *Let* $r(t)$ *be a logarithmically bounded polynomial-time-computable function. If there exists an oracle* X *such that some polynomial-time function solves* $\mathcal{P}_{r(t)}$ *using fewer than* $\lceil \log(r(t)+1) \rceil$ *queries to* X*, then* P = NP.

Proof. Let $q(t) = \lceil \log(r(t)+1) \rceil - 1$ and let M be a polynomial-time oracle Turing machine which solves $\mathcal{P}_r$ using $q(t)$ queries to X. We know that $q(t) = O(\log \log n)$, so the entire oracle computation tree of M on input $F_1, \ldots, F_{r(t)}$ is polynomially bounded and can be searched deterministically. In fact, the oracle computation tree has at most $r(t)$ leaves since for all x, $2^{\lceil \log x \rceil} < 2x$. One of these leaves represents the correct computation of $M^X(F_1, \ldots, F_{r(t)})$ and contains the value of $\#^{\mathrm{SAT}}_{r(t)}(F_1, \ldots, F_{r(t)})$. However, there are $r(t)+1$ possible answers for $\#^{\mathrm{SAT}}_{r(t)}(F_1, \ldots, F_{r(t)})$ ranging from 0 to $r(t)$. Therefore, one possible answer, call it z, does not appear in any leaf. Moreover, one of the leaves contains the correct answer, so $z \neq \#^{\mathrm{SAT}}_{r(t)}(F_1, \ldots, F_{r(t)})$. Thus we can construct a polynomial time Turing machine M' which on input $F_1, \ldots, F_{r(t)}$ prints out a number $z \le r(t)$ such that $z \neq \#^{\mathrm{SAT}}_{r(t)}(F_1, \ldots, F_{r(t)})$.

Now we show that P = NP under this condition. Given a Boolean formula F, consider its disjunctive self-reduction tree. Each node in the tree is a formula; the children of a node are the two formulas obtained by instantiating one of the variables in the formula by 0 and by 1. With F at the root of the tree, the tree has height t and exponential size. However, we will only examine $r(t)$ levels of the tree at a time. Now let $F = F_1, \ldots, F_{r(t)}$ be a path in this tree. Suppose that $F_{i+1} \in \mathrm{SAT}$. Since F_{i+1} is a child of F_i, we can assume that $F_i \in \mathrm{SAT}$. Thus the promise condition of $\mathcal{P}_{r(t)}$ holds and we can use M' to find a number z such that $z \neq \#^{\mathrm{SAT}}_{r(t)}(F_1, \ldots, F_{r(t)})$. Now replace the subtree rooted at F_z with the subtree rooted at F_{z+1}. If $F_z \notin \mathrm{SAT}$, then $F_{z+1} \notin \mathrm{SAT}$ by the promise condition. If $F_z \in \mathrm{SAT}$ and $F_{z+1} \notin \mathrm{SAT}$, then z would equal $\#^{\mathrm{SAT}}_{r(t)}(F_1, \ldots, F_{r(t)})$, which contradicts our construction of M'. Thus

$F_{z+1} \in \text{SAT}$ iff $F_z \in \text{SAT}$ and we have shortened the path by 1. Repeat this process for all paths in any order until all the paths in the tree have length less than $r(t)$. Now the original formula F is satisfiable iff one of the leaves (in which all the variables have been instantiated) evaluates to true. Since the final tree is polynomially bounded, we can check every leaf exhaustively. □

In the proofs that follow, we need to construct graphs in which the maximum clique size can occur only at restricted intervals. To assist in this construction, we define two operators on graphs: $\oplus$ and $\otimes$. We also use K_i to denote the complete graph with i vertices.

DEFINITION 4.4. *Given two graphs G_1 and G_2, the graph $H = G_1 \oplus G_2$ is constructed by taking the disjoint union of the vertices of G_1 and G_2. The edges of H are all the edges of G_1 and G_2, plus the edges (u, v) for each vertex u in G_1 and v in G_2. (Thus every vertex in G_1 is connected to every vertex in G_2).*

DEFINITION 4.5.[1] *Given two graphs G_1 and G_2, the graph $H = G_1 \otimes G_2$ is constructed by replacing each vertex of G_1 with a copy of G_2. Furthermore, for each edge (u, v) in G_1, each vertex in the copy of G_2 replacing u is connected to every vertex in the copy of G_2 replacing v. Note that $\otimes$ is not commutative and that we give $\otimes$ higher precedence than $\oplus$.*

LEMMA 4.6. *Let G_1 and G_2 be any two graphs, then*

$$\omega(G_1 \oplus G_2) = \omega(G_1) + \omega(G_2) \quad \textit{and}$$
$$\omega(G_1 \otimes G_2) = \omega(G_1) \cdot \omega(G_2).$$

4.2. A simple lower bound. As a first lower-bound result, we show that for all oracles X, no function in $\text{PF}^{X[1]}$ can approximate $\omega(G)$ within a factor of 2 unless P = NP. To do this, we start with the assumption that some function $f \in \text{PF}^{X[1]}$ does approximate $\omega(G)$ within a factor of 2 and show that using this function we can solve the promise problem $\mathcal{P}_2$ using only one query to X. Then P = NP by Lemma 4.3.

In our construction, we start with the input to the promise problem $\mathcal{P}_2$, which is a pair of Boolean formulas F_1 and F_2, each with t variables. Now, using the reduction from Lemma 2.2, we construct two graphs G_1 and G_2 with t^d vertices such that

$$F_i \in \text{SAT} \Longrightarrow \omega(G_i) = t^b \quad \text{and}$$
$$F_i \notin \text{SAT} \Longrightarrow \omega(G_i) < t^s.$$

Then let $H = G_1 \oplus (K_2 \otimes G_2)$. By Lemma 4.6, $\omega(H) = \omega(G_1) + 2 \cdot \omega(G_2)$. Also, assume that t is sufficiently large so that $t^b > 6 \cdot t^s$.

Consider the effect of the satisfiability of F_1 and F_2 on $\omega(H)$. If $F_1 \notin \text{SAT}$ and $F_2 \notin \text{SAT}$, then

$$\omega(H) = \omega(G_1) + 2 \cdot \omega(G_2) < 3 \cdot t^s.$$

On the other hand, if $F_1 \in \text{SAT}$ and $F_2 \notin \text{SAT}$, then $t^b \leq \omega(H) < t^b + 2 \cdot t^s$. Finally, if $F_1 \in \text{SAT}$ and $F_2 \in \text{SAT}$, then $\omega(H) = 3 \cdot t^b$.

Because of the promise condition of the promise problem, we do not have the case where $F_1 \notin \text{SAT}$ and $F_2 \in \text{SAT}$. This restricts the value of $\omega(H)$ to three nonoverlapping intervals. In fact, these intervals are separated by a factor of 2. Thus a factor-of-2 approximation of $\omega(H)$ will tell us which formulas are satisfiable. For

[1] This is identical to graph composition in Garey and Johnson [19]. The cover of [19] is the picture of a 3-clique composed with a graph that has three points connected by two edges.

example, if we are given an approximation x guaranteed to be within a factor of 2, and $x \geq 1.5 \cdot t^b$, then we know that both F_1 and F_2 are satisfiable. If this approximation can be achieved using only one query to X, then we can solve the promise problem $\mathcal{P}_2$ in $\mathrm{PF}^{X[1]}$ and P = NP.

4.3. The general construction. In this section, we prove the general theorem for the lower bound on approximating the size of the maximum clique in a graph. First, we define a class of approximation factors.

DEFINITION 4.7. *We call a function $k : \mathbb{N} \to \mathbb{R}$ a* nice approximation factor *if it is computable in polynomial time and all of the following hold:*

1. $\exists n_0\ \forall n > n_0,\ 2 \leq k(n) < n$.
2. $\exists n_0\ \forall m > n > n_0,\ k(m) \geq k(n)$.
3. $\forall \delta > 0,\ \exists n_0,\ \forall m > n > n_0$,

$$(1+\delta)\frac{\log m}{\log k(m)} > \frac{\log n}{\log k(n)}.$$

Please note that $k(n)$ is nice if $k(n)$ equals $n^{1/a}$, $\log n$, $(\log n)^a$, or a constant ≥ 2. The natural interpretation of the third condition is the following. Consider the function $f(n) = (\log n)/(\log k(n))$. The function $f(n)$ is also related to $k(n)$ by $k(n) = n^{1/f(n)}$. The third condition is satisfied if $f(n)$ is increasing, if $f(n)$ is constant, or if $f(n)$ is decreasing but converges to a limit (e.g., when $k(n) = \sqrt{n}/2$). Since $k(n) < n$, $f(n)$ is bounded below by 1. Thus if $f(n)$ is decreasing almost everywhere, it must converge to a limit. Hence the third condition is not satisfied only when $f(n)$ alternates between increasing and decreasing infinitely often. We rule out these functions as approximation factors.

THEOREM 4.8. *Let $k(n)$ be a nice approximation factor which* (1) *is unbounded,* (2) *converges to an integral constant, or* (3) *converges to a sufficiently large constant. Then for all oracles X, no polynomial-time function can approximate $\omega(G)$ within a factor of $k(n)$ using $\lfloor \log \log_{k(n)} n - c \rfloor$ or fewer queries to X unless* P = NP*, where $c = 1 + \log(1/\epsilon)$.*

Proof. The general strategy of this proof is to reduce the promise problem $\mathcal{P}_r$ to the problem of approximating the maximum clique size of a graph H within a factor of $k(n)$. Since Lemma 4.3 provides us with a lower bound on the complexity of $\mathcal{P}_r$ (assuming that P $\neq$ NP), we obtain a lower bound on the complexity of approximating $\omega(H)$.

Therefore, we begin with the input to the promise problem $\mathcal{P}_r$, the Boolean formulas $F_1, \ldots, F_r$, each with t variables. (The actual value of r will be chosen later.) We convert each formula F_i into a graph G_i with $m = t^d$ vertices according to the construction described in Lemma 2.2. The values of $\omega(G_i)$ are restricted by

$$F_i \in \mathrm{SAT} \Longrightarrow \omega(G_i) = t^b \quad \text{and}$$
$$F_i \notin \mathrm{SAT} \Longrightarrow \omega(G_i) < t^s.$$

Then we choose a gap size g. In the simple example above, g is 2. In this proof, the value of g and r will depend on t. However, for notational convenience, we do not make this dependency explicit. Moreover, we can only choose g to be a whole number. Now, given the choices of r and g, we construct the graph H as follows:

$$H = G_1 \oplus (K_g \otimes G_2) \oplus (K_{g^2} \otimes G_3) \oplus \cdots$$
$$\oplus (K_{g^{i-1}} \otimes G_i) \oplus \cdots \oplus (K_{g^{r-1}} \otimes G_r).$$

At first glance, this does not appear to be a polynomial-time construction because we could double the size of each succeeding graph. However, r will turn out to be logarithmically bounded, so $|H|$ will be polynomially bounded. Finally, let

$$\nu_j = \sum_{i=0}^{j-1} g^i = \frac{g^j - 1}{g - 1};$$

then $n = |H| = \nu_r \cdot m = \nu_r \cdot t^d$.

Now suppose there exists a polynomial-time function which approximates $\omega(H)$ within a factor of $k(n)$ using $\log\log_{k(n)} n - c$ queries to X. We want to show that the factor-$k(n)$ approximation of $\omega(H)$ also tells us the value of $\#_r^{\mathrm{SAT}}(F_1, \ldots, F_r)$. Also, we want to constrain the choice of g and r so that $\log\log_{k(n)} n - c < \lceil \log(r+1) \rceil$. Then by Lemma 4.3, being able to approximate $\omega(H)$ within a $k(n)$ factor using only $\log\log_{k(n)} n - c$ queries to X would imply that P = NP. To make this claim, our choices of g and r are critical. We have already encountered one constraint on g and r. The other constraints arise when we analyze the possible values of $\omega(H)$.

For the moment, assume that there are exactly z satisfiable formulas in $F_1, \ldots, F_r$. Then $F_1, \ldots, F_z$ are in SAT and $F_{z+1}, \ldots, F_r$ are in $\overline{\mathrm{SAT}}$ by the promise condition of $\mathcal{P}_r$. We can calculate $\omega(H)$ as follows:

$$\begin{aligned}
\omega(H) &= \sum_{i=1}^{r} g^{i-1} \omega(G_i) \\
&= \left(\sum_{i=1}^{z} g^{i-1} \omega(G_i) \right) + \left(\sum_{i=z+1}^{r} g^{i-1} \omega(G_i) \right) \\
&= \nu_z t^b + \left(\sum_{i=z+1}^{r} g^{i-1} \omega(G_i) \right).
\end{aligned}$$

Since F_i is unsatisfiable, for $z+1 \le i \le r$, we can estimate the size of the second term by $\nu_r \cdot t^s$. Then we can bound the size of the maximum clique of H:

$$\nu_z t^b \le \omega(H) \le \nu_z t^b + \nu_r t^s.$$

We want to show that an approximation of $\omega(H)$ within a factor of g will also allow us to calculate $\#_r^{\mathrm{SAT}}(F_1, \ldots, F_r)$. Therefore, assume that we are provided with a function which approximates $\omega(H)$ within a factor g. To distinguish the case where z formulas are satisfiable from the case where $z+1$ formulas are satisfiable, we must have $g \cdot (\nu_z t^b + \nu_r t^s) < \nu_{z+1} t^b$. That is, the upper bound on $\omega(H)$ when z formulas are satisfiable must be smaller by a factor of g than the lower bound on $\omega(H)$ when $z+1$ formulas are satisfiable. Since $\nu_{z+1} = g\nu_z + 1$, this condition is satisfied if we have the constraint that $g\nu_r t^s < t^b$. Similarly, under this constraint, we would also have $g(\nu_{z-1} t^b + \nu_r t^s) < \nu_z t^b$. Hence we have restricted the possible values of $\omega(H)$ to $r+1$ disjoint intervals which are separated by a factor of g. Thus given a number x which is guaranteed to be an approximation of $\omega(H)$ within a factor of $k(n)$, $k(n) \le g$, we can find the largest z such that $gx \ge \nu_z t^b$. This largest z is equal to $\#_r^{\mathrm{SAT}}(F_1, \ldots, F_r)$. It is important to note here that the approximation factor $k(n)$ depends on n which is the size of H and not the size of the original input. Furthermore, the value of n depends on the value of g. Thus it is not a simple task to choose g and have $k(n) \le g$. (Recall that $n = \nu_r t^d$.)

In summary, we must choose g and r such that the following constraints hold. (Recall that we use g and $g(t)$ interchangeably.)

Constraint 1. $g(t)\nu_r t^s \stackrel{\text{def}}{=} g \cdot (g^r - 1)/(g-1) \cdot t^s < t^b$.

Constraint 2. $k(n) \le g(t)$.

Constraint 3. $\lfloor (\log \log_{k(n)} n) - c \rfloor < \lceil \log(r(t)+1) \rceil$.

The main difficulty of this proof is to choose the correct values for the parameters. For example, we can satisfy Constraint 3 by picking a large value for r. However, if r is large, then Constraint 1 is harder to satisfy. We also satisfy Constraint 2 easily by choosing a large g, but a large g would force us to pick a small r, which then violates Constraint 3. Let $n' = t^{b-s+d}$. We will show that the following choices satisfy the constraints:

- $g = \lceil k(t^{b-s+d}) \rceil \stackrel{\text{def}}{=} \lceil k(n') \rceil$.
- $r = \lfloor \log_g t^{b-s}(g-1)/g \rfloor$.

Note that g and r are chosen to be integers. This is important for our construction, but it is also the source of some difficulties in our calculations. The reader may find the proof easier to follow by substituting 2 or $\sqrt{n}$ for g. These are the two extreme possibilities.

First, we show that Constraint 1 holds. By our choice of r, $r \le \log_g(t^{b-s}(g-1)/g)$. By removing the $\log_g$, isolating t^b, and using the fact that $g^r - 1 < g^r$, we obtain the following, and hence Constraint 1 holds:

$$g\nu_r t^s = g \cdot \frac{g^r - 1}{g-1} \cdot t^s < g \cdot \frac{g^r}{g-1} \cdot t^s \le t^b. \tag{4.1}$$

We can also use equation (4.1) to estimate n in terms of t. From our construction, we know that $n = \nu_r t^d$. From equation (4.1), we know that $\nu_r < t^{b-s}$. Hence $n < t^{b-s+d} = n'$. Since $k(n)$ is a nice approximation factor, it is monotonic after some point. Hence for large enough t, $k(n) \le k(n') \le \lceil k(n') \rceil = g(t)$. Thus Constraint 2 holds.

Finally, we have to show that Constraint 3 holds. Since $c = 1 + \log(1/\epsilon)$, $2^{-c} = \epsilon/2$. It suffices to show that $(\log \log_{k(n)} n) - c < \log(r+1)$ or that $(\epsilon/2)\log_{k(n)} n < r+1$. By substituting the definition of r and being careful with the floor notation, we can satisfy Constraint 3 by showing that

$$\frac{2}{\epsilon} \log_g \frac{g}{g-1} + \log_{k(n)} n < \frac{2}{\epsilon} \log_g t^{b-s}. \tag{4.2}$$

In the next step, we rewrite $(2/\epsilon)$ as follows. Recall that $\epsilon = (b-s)/d$.

$$\frac{2}{\epsilon} = \frac{2}{(1+\epsilon)} \frac{(1+\epsilon)}{\epsilon} = \left(\frac{1-\epsilon}{2+2\epsilon} + \frac{3+\epsilon}{2+2\epsilon} \right) \frac{b-s+d}{b-s}.$$

Thus

$$\frac{2}{\epsilon} \log_g t^{b-s} = \frac{1-\epsilon}{2+2\epsilon} \log_g n' + \frac{3+\epsilon}{2+2\epsilon} \log_g n'.$$

Since ϵ must be less than 1, $(3+\epsilon)/(2+2\epsilon) > 1$ and $(1-\epsilon)/(2+2\epsilon) > 0$. We can satisfy Constraint 3 by showing the following two inequalities:

$$\frac{2}{\epsilon} \log_g \frac{g}{g-1} < \frac{1-\epsilon}{2+2\epsilon} \log_g n', \tag{4.3}$$

$$\log_{k(n)} n < \frac{3+\epsilon}{2+2\epsilon} \log_g n'. \tag{4.4}$$

Equation (4.3) holds since $g/(g-1)$ is bounded by 2 and $n' = t^{b-s+d}$ is unbounded.

Now we have to show that equation (4.4) holds. First, pick $\delta > 0$ small enough so that

$$(1+\delta)^2 < \frac{3+\epsilon}{2+2\epsilon}.$$

Recall that $g = \lceil k(n') \rceil$. Thus

$$\frac{3+\epsilon}{2+2\epsilon} \log_g n' > (1+\delta)^2 \cdot \frac{\log n'}{\log k(n')} \cdot \frac{\log k(n')}{\log \lceil k(n') \rceil}.$$

Consider the ratio $\log k(n')/\log \lceil k(n') \rceil$. If $k(x)$ is always an integer, then the ratio is just 1. If $k(x)$ is growing monotonically, then the ratio converges to 1 from below and will eventually rise above the constant $1/(1+\delta)$. In fact, it is sufficient to assume that $k(x)$ is an unbounded function. The proof also works when $k(x)$ converges to an integral constant or a sufficiently large constant. The proof *does not* work when $k(x)$ is a small fractional constant (e.g., 2.5). Hence we exclude this case in the hypothesis of the theorem. (In that case, however, we can prove the same theorem with $c = 2 + \log(1/\epsilon)$, which results in a worse lower bound.) Thus we may assume that for sufficiently large t,

$$\frac{3+\epsilon}{2+2\epsilon} \log_g n' > (1+\delta) \frac{\log n'}{\log k(n')}.$$

Using the third niceness property of $k(x)$, for sufficiently large t,

$$(1+\delta) \frac{\log n'}{\log k(n')} > \frac{\log n}{\log k(n)} = \log_{k(n)} n.$$

Thus, equations (4.3) and (4.4) are true, and all of the constraints are satisfied for large enough t. □

In the preceding theorem, we make some additional assumptions about $k(n)$ beyond niceness to show that equation (4.4) holds. If we are willing to settle for a slightly worse lower bound, we can prove a similar result for all nice approximation functions $k(n)$. Alternatively, we can make the assumption that $\epsilon \leq 1/4$. The proof for each case is nearly identical to the proof of Theorem 4.8 except we use the fact that the ratio $\log k(n')/\log \lceil k(n') \rceil$ is bounded below by $\log 2/\log 3 \approx 0.6309$ since $k(n) \geq 2$ for all n.

COROLLARY 4.9. *Let $k(n)$ be a nice approximation factor. Then for all oracles X, no polynomial-time function can approximate $\omega(G)$ within a factor of $k(n)$ using $\lfloor \log \log_{k(n)} n - c \rfloor$ or fewer queries to X unless* P = NP*, where $c = 2 + \log(1/\epsilon)$.*

COROLLARY 4.10. *Let $k(n)$ be a nice approximation factor. If $\epsilon \leq 1/4$, then for all oracles X, no polynomial-time function can approximate $\omega(G)$ within a factor of $k(n)$ using $\lfloor \log \log_{k(n)} n - c \rfloor$ or fewer queries to X unless* P = NP*, where $c = 1 + \log(1/\epsilon)$.*

5. Approximating the chromatic number. The results that we have stated so far also hold for many other NP-optimization problems. For any NP-optimization problem where the solutions range from 1 to n (or even n^a), the upper bound on the number of queries needed to approximate the problem can be easily derived from the techniques in section 3. To show that the lower bounds hold, we need a reduction

from SAT to the new problem similar to the one for Clique in Lemma 2.2. Recently, Lund and Yannakakis have discovered such reductions for Graph Coloring and some related problems [27]. To repeat the proof, special attention must be given to any differences that may arise between minimization and maximization (see section 7). Thus we could obtain results analogous to the ones in Theorem 4.8 for the problem of approximating the chromatic number of a graph.

In this section, we take an alternative approach and prove the lower bounds using an approximation-preserving reduction from the maximum-clique-size problem to the chromatic number of a graph [27]. However, this reduction increases the size of the graphs, so the proof does not produce the best lower bounds. This approach can be extended to most of the problems which Lund and Yannakakis show to have approximability properties similar to that of Graph Coloring. We can show that Clique Partition, Clique Cover and Biclique Cover have similar lower bounds. This follows from approximation preserving reductions due to Simon [30] for Clique Cover and Biclique Cover. These reductions preserve the approximation ratio within $1+\epsilon$ for any $\epsilon > 0$ where the new problems have size $n^{O(1/\epsilon)}$. On the other hand, we have not been able to obtain a similar result for Fractional Chromatic Number. The reason is that the reduction there only preserves the approximation ratio with a $\log n$ multiplicative factor.

In the following, let $\alpha(G)$ and $\chi(G)$ denote, respectively, the size of the largest independent set and the chromatic number of a graph G. The next lemma is an application of the results of Lund and Yannakakis.

LEMMA 5.1. *There exists a polynomial-time transformation $T(G)$ such that for all graphs G with n vertices and for all primes p with $n \le p \le 2n$, $H = T(G)$ has the property that*

$$\frac{p^3 n^2}{\alpha(G)} \le \chi(H) \le \frac{p^3 n^2 (1+1/n)}{\alpha(G)}.$$

Furthermore, $|H| = n^2 \cdot p^5 \le 32 \cdot n^7$.

Proof. By Proposition 4.1[2] in the appendix of Lund and Yannakakis [27], there exists a polynomial-time transformation $T'(G,p,r)$ such that if G is a graph, p is a prime, r is a number, $p^2 \ge r \ge \alpha(G)$, $p \ge n$, and $H = T'(G,p,r)$, then

$$p^3 \cdot \frac{r}{\alpha(G)} \le \chi(H) \le p^3 \cdot \left\lceil \frac{r}{\alpha(G)} \right\rceil.$$

Given a graph G on n vertices, we define T as follows. Let $r = n^2$ and find a prime p such that $n \le p \le 2n$. (Such primes exist by Bertrand's theorem and can be found easily since the length of the input is n not $\log n$.) Let $T(G) = T'(G,p,r)$. If $H = T(G)$, then

$$p^3 \cdot \frac{n^2}{\alpha(G)} \le \chi(H) \le p^3 \cdot \left\lceil \frac{n^2}{\alpha(G)} \right\rceil < p^3 \cdot \frac{n^2(1+1/n)}{\alpha(G)}. \qquad \square$$

The following theorem shows that we can derive a lower bound on the complexity of approximating the chromatic number of a graph using the lemma above.

[2]Proposition 4.1 as stated by Lund and Yannakakis requires that $p > r$, but the proofs show that in fact $p^2 > r$ and $p \ge n$ are sufficient. We use this proposition instead of their main theorem because it deals with general graphs instead of the special graphs produced by the reduction in Lemma 2.2.

THEOREM 5.2. *Let $k(n)$ be a nice approximation factor such that for large n, $k(n^8) < n/2$. Then for all oracles X, no polynomial-time function can approximate $\chi(G)$ within a factor of $k(n)$ using $\lfloor \log\log_{k(n)} n - c \rfloor$ or fewer queries to X unless* P = NP*, where $c = 6 + \log(1/\epsilon)$.*

Proof. We reduce approximating the clique size of a graph Q to approximating the chromatic number of a graph H. If $\chi(H)$ can be approximated using too few queries, then the lower bound on approximating $\omega(Q)$ from Corollary 4.9 would be violated and we can conclude that P = NP.

We are given a nice approximation factor $k(n)$. Suppose that some polynomial-time algorithm $A(G)$ approximates $\chi(G)$ within a factor of $k(n)$ using $\lfloor \log\log_{k(n)} n - c \rfloor$ queries to X for all graphs G with n vertices. Let $k'(n) = k(n^8)$. It is simple to check that $2k'(n)$ is also a nice approximation factor.

Now, given any graph Q with n vertices, we approximate $\omega(Q)$ within a factor of $2k'(n)$ as follows. Construct $H' = T(Q')$ using Lemma 5.1, where Q' is the complement of Q. Thus we know that $\alpha(Q') = \omega(Q)$ and $|H'| \leq 32 \cdot n^7$. Now we construct the graph H by adding dummy vertices to H' so that $|H| = n^8 = N$ and $\chi(H) = \chi(H')$. Finally, we use the algorithm A to compute an approximation of $\chi(H)$ within a factor of $k(N)$. This uses no more than

$$\lfloor \log\log_{k(N)} N - 6 - \log(1/\epsilon) \rfloor = \lfloor \log\log n - \log\log k(n^8) - 3 - \log(1/\epsilon) \rfloor$$

queries to X. Since $A(H)$ is a factor-$k(N)$ approximation, we know that

$$\chi(H) \leq A(H) \leq k(N)\chi(H).$$

From Lemma 5.1, we also know that

$$\frac{p^3 n^2}{\alpha(Q')} \leq \chi(H') < \frac{p^3 n^2 (1 + 1/n)}{\alpha(Q')}.$$

Since $\alpha(Q') = \omega(Q)$ and $\chi(H) = \chi(H')$, we obtain

$$\frac{\omega(Q)}{k(N)(1 + 1/n)} \leq \frac{p^3 n^2}{A(H)} \leq \omega(Q).$$

Thus the value $p^3 n^2 / A(H)$ approximates $\omega(Q)$ within a factor of

$$k(N)\left(1 + \frac{1}{n}\right) \leq 2k(n^8) = 2k'(n).$$

Since $2k'(n)$ is a nice approximation factor, by Corollary 4.9, approximating $\omega(Q)$ within factor $2k'(n)$ has a lower bound of

$$\left\lfloor \log\log_{2k'(n)} n - 2 - \log\left(\frac{1}{\epsilon}\right) \right\rfloor = \left\lfloor \log\log n - \log\log 2k(n^8) - 2 - \log\left(\frac{1}{\epsilon}\right) \right\rfloor.$$

Finally, since $(\log\log k(n^8)) + 1 > \log((\log k(n^8)) + 1)$, computing $A(H)$ used no more than $\lfloor \log\log_{2k'(n)} n - 2 - \log(1/\epsilon) \rfloor$ queries to X. Thus if such an algorithm exists, P = NP. □

Theorem 5.2 decreases the lower bound of Corollary 4.9 by four queries. This decrease is due in part to the fact that $|H| \approx |Q|^7$. Thus a more efficient reduction from clique size to chromatic number would yield a tighter lower bound. Also, for specific approximation factors, especially where the relationship between $k(n)$ and $k(n^7)$ is explicit, we can obtain slightly better lower bounds by reproducing the proof of Theorem 4.8.

6. The value of ϵ. The lower-bound results in the preceding sections depend on the value of the constant ϵ, where $0 < \epsilon \leq 1$. Recall that this is the same ϵ used by Arora et al. to show that no polynomial-time algorithm can approximate $\omega(G)$ within a factor of n^{ϵ} unless P = NP. Note that a larger value of ϵ indicates a better nonapproximability result which in turn provides tighter upper and lower bounds in the results of previous sections. Also, recall that for bounded query classes, even an improvement of one query is significant.

Currently, the exact value of ϵ is not known. However, by weakening the assumption that P $\neq$ NP to BPP $\neq$ NP, Bellare et al. [8] have shown that no polynomial-time algorithm can approximate $\omega(G)$ within a factor of $n^{1/30-o(1)}$. A further improvement was made by Bellare and Sudan [9], who showed that no polynomial-time algorithm can approximate $\omega(G)$ within a factor of $n^{1/5-o(1)}$ unless NP = ZPP. In this section, we use these results to obtain explicit lower bounds on the number of queries needed to approximate $\omega(G)$ under the stronger assumption that RP $\neq$ NP.

To prove these lower bound results, we need to adapt the proof techniques of the previous section to work with randomized functions instead of deterministic functions. A naïve approach would use a straightforward modification of Lemma 2.2 to produce a randomized reduction f from SAT to Clique Size such that

$$F \in \text{SAT} \Longrightarrow \text{Prob}_z[\omega(f(F,z)) = t^b] = 1,$$
$$F \notin \text{SAT} \Longrightarrow \text{Prob}_z[\omega(f(F,z)) < t^s] \geq 1 - \delta.$$

The difficulty with this approach is that the randomized version of Lemma 4.3 will use this randomized reduction polynomially many times. Thus when we show that RP = NP if the lower bounds are violated, we have to be very careful with the value of δ to make certain that the success probability of the overall procedure remains high enough. Such detailed analysis is possible using the techniques developed by Rohatgi [28]. However, the analysis can be made much simpler by observing that the randomized reduction from SAT to Clique Size can be achieved with "uniform" probability—that is, the same random string z can be used to correctly reduce any instance of SAT of a certain length. In the following, let $|F|$ denote the length of the encoding of the Boolean formula F.

LEMMA 6.1. *There exist constants s, b, and d with $0 < s < b < d$, polynomials $p(\cdot)$ and $q(\cdot)$, and a deterministic polynomial-time function f such that*

$$(\forall z \leq p(t))(\forall F,\ |F| = t)[F \in \text{SAT} \Longrightarrow \omega(f(F,z)) = t^b] \quad \textit{and}$$

$$\text{Prob}_{z \leq p(t)}[(\forall F,\ |F| = t)[F \notin \text{SAT} \Longrightarrow \omega(f(F,z)) < t^s]] \geq 1 - 2^{-q(t)},$$

where for each Boolean formula F, $|F| = t$, and each random string $z \leq p(t)$, $f(F,z)$ produces a graph with t^d vertices.

Proof sketch. This reduction is implicit in the work of Zuckerman [34]; we include a proof sketch for completeness. In this proof, the reduction f constructs a graph G from the formula F and a random string z. The random string z is used to choose a disperser graph H which allows f to amplify probability bounds. Choosing this disperser is the only random step used by f. If the randomly chosen H is indeed a disperser, then the same H can be used to reduce any formula F with t variables to a graph G. Hence the success probability of the reduction is independent of the particular formula F.

The starting point of the proof is not the deterministic reduction in Lemma 2.2 but the probabilistically checkable proof for SAT. As reported by Arora et al., there is

a probabilistically checkable proof for SAT where the verifier V uses $c_1 \log n$ random bits and looks at c_2 bits of the proof such that

$$F \in \text{SAT} \Longrightarrow \exists \pi,\ \forall z \in \{0,1\}^{c_1 \log n},\ V^{\pi}(F,z) \text{ accepts},$$

$$F \notin \text{SAT} \Longrightarrow \forall \pi,\ \text{Prob}_{z \in \{0,1\}^{c_1 \log n}}[V^{\pi}(F,z) \text{ accepts}] < \frac{1}{2}.$$

From the verifier V and the input F with t variables, we can use the construction of Feige et al. [18] to construct a graph G such that

$$F \in \text{SAT} \Longrightarrow \omega(G) = t^{c_1},$$
$$F \notin \text{SAT} \Longrightarrow \omega(G) < \frac{1}{2} \cdot t^{c_1}.$$

In this construction, each vertex of G represents one computation path of the verifier V (for every possible random string and every sequence of answer bits from the proof). An edge is added between two vertices if some proof π contains the answer bits which agree with both computation paths.

The construction above gives an approximation "gap" of only $1/2$. To obtain better results, we have to decrease the probability that V accepts an incorrect proof when $F \notin \text{SAT}$ *without using too many additional random bits.* This decrease can be achieved deterministically [15], which leads to Lemma 2.2, but then we lose track of the values of b, s, and d. Alternatively, as Zuckerman [34] pointed out, we can decrease the verifier's error randomly.

The key idea to Zuckerman's construction is to use special graphs called dispersers which were first introduced by Sipser [31]. For our purposes, a disperser may be defined as a bipartite graph with t^{c_3} vertices on the left and t^{c_1} vertices on the right such that each left vertex has degree $D = c_3 \log t + 2$ and every set of t^{c_1} left vertices is connected to at least $t^{c_1}/2$ right vertices. The value of the constant c_3 will be chosen later. By a theorem which Zuckerman attributes to Sipser [34, Theorem 3], such a disperser can be randomly generated with probability $1 - 2^{-t^{c_1}}$ by choosing the D neighbors of each left vertex randomly.

Suppose that we have a disperser H. We use H to construct a new verifier V'. We interpret each left vertex of H as a random string for V' and each right vertex of H as a random string for V. V' simulates V by randomly choosing a left vertex of H. This uses $c_3 \log t$ random bits. Let $z_1, \ldots, z_D$ be the right vertices connected to the chosen left vertex. V' simulates V D times using each of $z_1, \ldots, z_D$ as the random string for V. If every simulation of V accepts the proof π, then V' accepts the proof π. The complete simulation uses $c_3 \log t$ random bits and looks at Dc_2 bits of the proof.

Clearly, if $F \in \text{SAT}$, then V' will always accept. In fact, V' will accept even when H does not have the desired properties. Conversely, consider the case when $F \notin \text{SAT}$. We want to show that V' accepts with probability less than t^{c_1}/t^{c_3}. Therefore, suppose that V' accepted with probability $\geq t^{c_1}/t^{c_3}$. Then t^{c_1} of the left vertices cause V' to accept. Thus all the right vertices connected to these left vertices must cause V to accept. Since H is a disperser with the properties mentioned above, at least $t^{c_1}/2$ right vertices must cause V to accept. This contradicts our assumption about the error probability of V. Thus when $F \notin \text{SAT}$, V' accepts a proof π with probability less than t^{c_1}/t^{c_3}.

Now we construct a graph G from V' as described above. Since V' uses $c_3 \log t$ random bits and looks at Dc_2 bits of the proof, the graph G will have $n = 2^{c_3 \log t + Dc_2}$

vertices. When $F \in \mathrm{SAT}$, G has a "big" clique of size t^{c_3}. When $F \notin \mathrm{SAT}$, G has a "small" clique of size no more than t^{c_1}. The ratio of the size of the "big" clique to the size of the "small" clique expressed in terms of n is

$$t^{c_3-c_1} = n^{(c_3-c_1)\log t/(c_3 \log t + Dc_2)}.$$

Substituting $c_3 \log t + 2$ for D, we derive

$$\epsilon = \frac{b-s}{d} = \frac{1 - c_1/c_3}{1 + c_2 + 2c_2/(c_3 \log t)}.$$

Thus for all $\delta > 0$, we can have $\epsilon > (1+c_2)^{-1} - \delta$ by choosing c_3 to be large enough. Recall that c_2 is the number of bits in the proof that the verifier reads. This calculation shows the effect of c_2 on the value of ϵ. Finally, observe that the only random step used in the reduction is choosing the disperser H. □

The results of Bellare et al. [8] can be used to show that in Lemma 6.1 the ratio $(b-s)/d = 1/30 - o(1)$ because they produce a verifier for SAT that uses only 29 bits of the proof and follow Zuckerman's construction as described above. However, we are unable to exploit the results of Bellare and Sudan [9] because they use randomness for sampling—not just to generate pseudorandom bits. Nevertheless, we can give explicit lower bounds on the query complexity of approximating $\omega(G)$.

THEOREM 6.2. *Let $k(n)$ be a nice approximation factor. Then for all oracles X, no polynomial-time function can approximate $\omega(G)$ within a factor of $k(n)$ using $\lfloor \log \log_{k(n)} n - 6 \rfloor$ or fewer queries to X unless* RP = NP.

Proof sketch. To prove this theorem, we simply follow the proof of Theorem 4.8 except that we use $\epsilon = 1/31$ and Lemma 6.1 instead of Lemma 2.2 to reduce SAT to Clique Size. Since the success probability of the reduction is independent of the formula F, repeated use of the reduction does not decrease the success probability of the overall procedure. Again, the only random step in the entire procedure is randomly choosing a disperser graph H. This is also the case when we use the tree-pruning procedure in Lemma 4.3 to look for a satisfying assignment for the given Boolean formula. Since our procedure accepts a formula only when a satisfying assignment is found, it will never accept an unsatisfiable formula. The procedure may reject a satisfiable formula if the graph H turns out not to be a disperser. However, the probability of this happening is small. Thus the overall procedure is an RP algorithm for SAT.

Finally, note that in equation (4.2) of Theorem 4.8, we need to show that

$$\frac{2}{\epsilon} \log_g \frac{g}{g-1} + \log_{k(n)} n < \frac{2}{\epsilon} \log_g t^{b-s}.$$

In this proof, the value of ϵ is known, so we can rewrite the $2/\epsilon$ as

$$\frac{2}{\epsilon} = \frac{2}{(1+\epsilon)} \frac{(1+\epsilon)}{\epsilon} = 1.9375 \cdot \frac{1+\epsilon}{\epsilon} = (0.1375 + 1.8) \cdot \frac{b-s+d}{b-s}.$$

Then, as before,

$$\frac{2}{\epsilon} \log_g \frac{g}{g-1} < 0.1375 \cdot \log_g n'$$

because $g/(g-1) \le 2$. Also, we do not need any additional assumptions on $k(n)$ to show that

$$1.8 \cdot \log_g n' = 1.8 \cdot \frac{\log n'}{\log k(n')} \cdot \frac{\log k(n')}{\log \lceil k(n') \rceil} > (1+\delta) \frac{\log n'}{\log k(n')}$$

because $\log k(n')/\log\lceil k(n')\rceil \geq \log 2/\log 3 \approx 0.6309$. Hence $1.8 \cdot \log k(n')/\log\lceil k(n')\rceil > 1.1$ and we can let $\delta = 0.1$. Thus the assumption that $k(n)$ is nice and $c = 1 + \log 31 < 6$ suffices. □

Using the proof techniques described above, we can also extend our results to lower bounds for approximating the chromatic number.

COROLLARY 6.3. *Let $k(n)$ be a nice approximation factor such that for large n, $k(n) < n/2$. Then for all oracles X, no polynomial-time function can approximate $\chi(G)$ within a factor of $k(n)$ using $\lfloor \log\log_{k(n)} n - 10\rfloor$ or fewer queries to X unless* RP = NP.

7. Lower bounds for set cover. An instance of the Set Cover problem is a set system $\mathcal{S} = (n; S_1, \ldots, S_m)$ such that for all i, $S_i \subseteq \{1, \ldots, n\}$. We are asked to find the size of the smallest collection of S_i which covers $\{1, \ldots, n\}$. We denote the size of the minimum cover of $\mathcal{S}$ as $\texttt{SETCOVER}(\mathcal{S})$.

As we have discussed in the section on upper bounds, the Set Cover problem has a different complexity compared to Clique or Chromatic Number because there exist polynomial-time algorithms that approximate the size of the minimum set cover within a factor of $\ln n + 1$. In this section, we derive a lower bound on the complexity of approximating the size of the minimum set covering in terms of the number of queries to SAT.

One difficulty arises when we apply our techniques to minimization problems. To illustrate this, consider the construction of the graph H in section 4.2. We use the reduction from SAT to Clique to construct a graph G_1 from a formula F_1. This reduction has the special property that if $F_1 \in$ SAT, then $\omega(G_1) = t^b$. This *equality* is very important because it allows us to put only two copies of G_2 in H. If we only knew that $\omega(G_1) \geq t^b$, then $\omega(G_1)$ could be as large as m. Thus to ensure a large enough gap between the case where $F_1 \in$ SAT and $F_2 \notin$ SAT and the case where $F_1, F_2 \in$ SAT, we would have to use $2m/t^b$ copies of G_2. This would make the graph H too big and produce very bad lower bounds in the general theorem.

If equality is not possible, then we can settle for a good upper bound. For example, if the reduction from SAT to Clique guaranteed that $F_1 \in \text{SAT} \Longrightarrow t^b \leq \omega(G_1) \leq 3t^b$, then we only need to put six copies of G_2 in H. In the general case, we would obtain a lower bound of $\log\log_{3k(n)} n - c$.

The reduction from SAT to Set Cover in [27] does not give an upper bound on the size of the minimum set cover when the original formula is unsatisfiable. Thus we must use the following lemma and theorem.

LEMMA 7.1. *There exists a constant* $\texttt{CONST}$ *such that for all $l, m \geq \texttt{CONST}$ where $l \leq m$ and $\log\ln m < 0.09l$, there exist a set B and subsets $C_1, C_2, \ldots, C_m \subseteq B$ such that the following hold:*

1. *For any sequence of indices $1 \leq i_1 < i_2 < \cdots < i_l \leq m$, no collection $D_{i_1}, D_{i_2}, \ldots, D_{i_l}$ covers B where D_{i_j} is either C_{i_j} or the complement of C_{i_j}.*
2. *$C_1, \ldots, C_{1.1l}$ do cover B.*
3. *$|B| = (l + l\ln(m) + 2)2^l$.*

Furthermore, there exists a probabilistic Turing machine which on input l, m (in binary) produces such a set system with probability 2/3.

Proof. Let $B = \{1, \ldots, (l + l\ln(m) + 2)2^l\}$. Let the subsets $C_1, \ldots, C_m$ be a collection of subsets of B chosen randomly and independently—i.e., for each $x \in B$ and each C_i, $x \in C_i$ with probability one half. We show that with probability over 2/3 this collection suffices. Fix $D_{i_1}, D_{i_2}, \ldots, D_{i_l}$ as in the statement of the lemma. The probability that $B = D_{i_1} \cup \cdots \cup D_{i_l}$ is $(1 - (1/2)^l)^{|B|}$. The number of different D_{i_j}'s

is at most $2^l\binom{m}{l}$. Thus the probability that some collection of $D_{i_1}, \ldots, D_{i_l}$ covers B is bounded by

$$2^l\binom{m}{l}\left(1-\left(\frac{1}{2}\right)^l\right)^{|B|} \le e^{l+l\ln(m)-|B|/2^l} < \frac{1}{6}.$$

Hence the probability that item 1 occurs is at least 5/6.

Second, note that the probability that the first $1.1l$ sets cover B is

$$\left(1-\left(\frac{1}{2}\right)^{1.1l}\right)^{|B|} \ge e^{-2(l+l\ln(m)+2)2^l/2^{1.1l}} = e^{-2(l+l\ln(m)+2)2^{-0.1l}},$$

using the fact that $(1-x) \ge e^{-2x}$ for $x \in [0, 1/2]$. We need this quantity to be greater than $5/6 = e^{-c}$, where $c = \ln(6/5) > 0$. Hence we need

$$\begin{aligned} 2(l + l\ln(m) + 2)2^{-0.1l} &< \frac{5}{6}, \\ \frac{12}{5}(l + l\ln(m) + 2) &< 2^{0.1l}, \\ \log\frac{12}{5} + \log l + \log(\ln(m) + 2) &< 0.1l. \end{aligned}$$

Since $\log\ln m < 0.09l$, this holds for large enough l and m. Therefore, the desired constant `CONST` can be found. Hence the probability of satisfying item 2 is at least 5/6. Since the probability of satisfying item 1 is $\ge 5/6$ and the probability of satisfying item 2 is $\ge 5/6$, the probability of satisfying both is at least 2/3. ∎

THEOREM 7.2. *Given a formula φ, let $\mathcal{S}_\varphi$ be the instance of Set Cover described below. Let N be the size of $\mathcal{S}_\varphi$. Then there exists an integer K (depending only on the size of φ) such that*

$$\begin{aligned} &\varphi \in \text{SAT} \Longrightarrow \texttt{SETCOVER}(\mathcal{S}_\varphi) = K, \\ &\varphi \notin \text{SAT} \Longrightarrow \frac{0.99K}{2}\log N \le \texttt{SETCOVER}(\mathcal{S}_\varphi) \le \frac{1.1K}{2}\log N, \end{aligned}$$

where the last property holds with probability at least 2/3. Furthermore, the reduction can be applied to any number of formulas φ of the same size, and with probability 2/3, all the instances of Set Cover obtained will have the property above.

Proof. The proof of this theorem is a modification of the construction by Lund and Yannakakis [27]. In the rest of this proof, we assume that the reader is familiar with the notation and the proof in [27]. Given a formula φ, we carry out the construction in section 3.1 of [27] except that we use the sets $B, C_1, \ldots, C_m$ as specified in Lemma 7.1 as our building blocks. Using the ideas in [27], we obtain the following:

1. $\varphi \in$ SAT implies $\texttt{SETCOVER}(\mathcal{S}_\varphi) = K$, where $K = |Q_1| + |Q_2|$.
2. $\varphi \notin$ SAT implies $\texttt{SETCOVER}(\mathcal{S}_\varphi) \ge \frac{l}{2}(1-o(1))K$. Moreover, $\texttt{SETCOVER}(\mathcal{S}_\varphi) \le 1.1l(|Q_2|)$ since the first $1.1l$ answers for every query in Q_2 cover all the points by Lemma 7.1. Then since $|Q_1| = |Q_2|$,

$$\frac{l}{2}(1-o(1))K \le \texttt{SETCOVER}(\mathcal{S}_\varphi) \le \frac{1.1l}{2}K.$$

Now the theorem follows since l can be chosen such that $l = (1-\epsilon)\log N$ for any $\epsilon > 0$. Also, note that $m = 2^{O(l)}$ and thus we can apply Lemma 7.1.

Furthermore, note that once we have chosen one set system from Lemma 7.1, we can use it in any number of reductions involving instances of the same size. Thus if the set system has the required property, then all the reduced instances will have the required properties. □

The consequence of this theorem is that there exists a *randomized* reduction from SAT to Set Cover which runs in time $O(n^{\text{polylog}\, n})$. This reduction allows us to duplicate the construction of Theorem 4.8 for Set Cover and obtain the following lower bound.

THEOREM 7.3. *Let $k : \mathbb{N} \to \mathbb{R}$ be a function such that for all n large enough, $2 \le k(n) < n$; $m > n$ implies $k(m) \ge k(n)$; and $\forall \delta > 0,\ m > n$ implies that*

$$(1+\delta)\frac{\log\log m}{\log k(m)} > \frac{\log\log n}{\log k(n)}.$$

Let $\mathcal{S}$ be an instance of Set Cover. Then for all oracles X, no polynomial-time function can approximate the size of the minimum set cover within a factor of $k(n)$ using $\log\log_{k(n)} \ln n - 1$ or fewer queries to X unless NP $\subseteq$ RTIME$[n^{\text{polylog}\, n}]$.

Proof. This proof is analogous to the proof of Theorem 6.2. We start with the promise problem $\mathcal{P}_r$ with r to be chosen later. Since our construction is in time $O(n^{\text{polylog}\, n})$, we need a different lower bound on the complexity of $\mathcal{P}_r$. Using the proof technique of Lemma 4.3, one can show that for $r = O(\text{polylog}\, n)$, if some DTIME$[n^{\text{polylog}\, n}]$ machine solves $\mathcal{P}_r$ using fewer than $\lceil \log(r+1) \rceil$ queries to any oracle X, then NP $\subseteq$ DTIME$[n^{\text{polylog}\, n}]$. As in Theorem 6.2, it is important that in Theorem 7.2 the randomized reduction from SAT to Set Cover can be used repeatedly without decreasing the success probability of the overall procedure.

Now let $F_1, \ldots, F_r$ be a sequence of Boolean formulas with t variables each which satisfies the promise condition of the promise problem $\mathcal{P}_r$. We use Theorem 7.2 to construct r instances of Set Cover $\mathcal{S}_1, \ldots, \mathcal{S}_r$. Let m be the size of the underlying set of $\mathcal{S}_i$. Since the construction takes time $O(n^{\text{polylog}\, n})$, m is $O(t^{\text{polylog}\, t})$. We know with probability $2/3$ that for each i,

$$F_i \in \text{SAT} \Longrightarrow \texttt{SETCOVER}(\mathcal{S}_i) = K,$$
$$F_i \notin \text{SAT} \Longrightarrow \frac{0.99K}{2}\log m \le \texttt{SETCOVER}(\mathcal{S}_i) \le \frac{1.1K}{2}\log m.$$

We will now restrict our attention to this case. We combine the r instances of Set Cover into a single instance of Set Cover $\mathcal{T}$:

$$\mathcal{T} = g^{r-1} \otimes \mathcal{S}_1 \oplus g^{r-2} \otimes \mathcal{S}_2 \oplus \cdots \oplus g \otimes \mathcal{S}_{r-1} \oplus \mathcal{S}_r.$$

Let n be the size of the underlying set for $\mathcal{T}$. We define $\oplus$ and $\otimes$ so that

$$\texttt{SETCOVER}(\mathcal{S}_1 \oplus \mathcal{S}_2) = \texttt{SETCOVER}(\mathcal{S}_1) + \texttt{SETCOVER}(\mathcal{S}_2),$$

and for any positive integer a,

$$\texttt{SETCOVER}(a \otimes \mathcal{S}) = a \cdot \texttt{SETCOVER}(\mathcal{S}).$$

This is accomplished as follows. Let $\mathcal{S}_1 = (n_1; S_{1,1}, \ldots, S_{1,p})$ and $\mathcal{S}_2 = (n_2; S_{2,1}, \ldots, S_{2,q})$ be two instances of Set Cover. We define $\mathcal{S}_1 \oplus \mathcal{S}_2$ to be

$$(n_1 + n_2; S_{1,1}, \ldots, S_{1,p}, S'_{2,1}, \ldots, S'_{2,q}),$$

where $S'_{2,i} = \{x + n_1 \mid x \in S_{2,i}\}$. Then we can define $a \otimes \mathcal{S}$ simply as $\mathcal{S} \oplus \cdots \oplus \mathcal{S}$ repeated a times.

As in the proof of Theorem 4.8, the value of g in the construction of $\mathcal{T}$ will be chosen later. Note that we construct $\mathcal{T}$ using g^{r-1} copies of $\mathcal{S}_1$ instead of g^{r-1} copies of $\mathcal{S}_r$. This is "backwards" compared to the construction H in Theorem 4.8. We need to make this change because $\texttt{SETCOVER}(\mathcal{S}_i)$ is small when $F_i \in \text{SAT}$ and large when $F_i \notin \text{SAT}$ (again, backwards compared to Clique). Then a good approximation of $\texttt{SETCOVER}(\mathcal{T})$ will solve the promise problem $\mathcal{P}_r$. However, here the approximation must be within a factor of $0.9g$ instead of g because we only know that $\texttt{SETCOVER}(\mathcal{S}_i)$ is in the interval between $(0.99/2)K \log m$ and $(1.1/2)K \log m$ when $F_i \notin \text{SAT}$.

In this construction, when there are exactly z satisfiable formulas in $F_1, \ldots, F_r$, we have the following bounds:

$$\frac{0.99}{2} v_{r-z} K \log m \le \texttt{SETCOVER}(\mathcal{T}) \le v_r K + \frac{1.1}{2} v_{r-z} K \log m.$$

Thus we can obtain a lower bound for approximating the size of the minimum set cover (assuming NP $\not\subseteq$ RTIME$[n^{\text{polylog}\, n}]$) if we show that there exist r and g which satisfy the following constraints. Recall that m and n are the sizes of the underlying sets for $\mathcal{S}$ and $\mathcal{T}$, respectively. The value of m is expressible in terms of t. To make our notation simpler, we express g and r in terms of m instead of t or drop the argument altogether.

Constraint 1. $g(m)\nu_r < (1.1/2) \log m$.

Constraint 2. $k(n) \le 0.9g(m)$.

Constraint 3. $\lfloor \log \log_{k(n)} \ln n - 1 \rfloor < \lceil \log(r(m) + 1) \rceil$.

For this proof, we let $n' = (1.1/2)m \log m$ and choose g and r as follows:

- $g(m) = \lceil k(n')/0.9 \rceil$.
- $r = \lfloor \log_g (((1.1/2) \log m) \cdot (g-1)/g) \rfloor$.

As in the proof of Theorem 4.8, our choice of r implies that Constraint 1 holds and that $n' > n$. Since $k(n)$ is nondecreasing almost everywhere, it follows that $k(n) \le 0.9g(m)$ and that Constraint 2 also holds.

Finally, we show that Constraint 3 holds. It suffices to show that the following equations hold. (They are analogous to equations (4.3) and (4.4) in Theorem 4.8.)

$$2 \cdot \log_g \frac{g}{g-1} < 0.25 \cdot \log_g \left(\frac{1.1}{2} \log m \right), \tag{7.1}$$

$$\log_{k(n)} \ln n < 1.75 \cdot \log_g \left(\frac{1.1}{2} \log m \right). \tag{7.2}$$

Equation (7.1) is satisfied for m large since $g \ge 2$, $g/(g-1) \le 2$. Equation (7.2) is proved as follows. We start with the "niceness" assumptions on $k(n)$ and obtain

$$\log_{k(n)} \ln n \le \frac{\log \log n}{\log k(n)} < 1.05 \cdot \frac{\log \log n'}{\log k(n')}.$$

Recall that $n' = (1.1/2)m \log m$. Then for large m, $(\log n')^{1.05} < (\log m)^{1.1}$. Thus we have

$$1.05 \cdot \frac{\log \log n'}{\log k(n')} < 1.1 \cdot \frac{\log \log m}{\log k(n')}.$$

Then using the fact that for $x \geq 2$, $\log 3 \cdot \log x / \log\lceil x/0.9 \rceil > 1$, we have

$$1.1 \cdot \frac{\log\log m}{\log k(n')} < 1.1 \cdot \log 3 \cdot \frac{\log\log m}{\log k(n')} \cdot \frac{\log k(n')}{\log\lceil k(n')/0.9 \rceil} = 1.1 \cdot \log 3 \cdot \frac{\log\log m}{\log g}.$$

Now $\log 3 \approx 1.584963\ldots$, so $1.1 \cdot \log 3 < 1.75 - 0.005$. Also, we know that for m large enough, $0.005 \log\log m > 1.75 \cdot \log(2/1.1)$. Therefore,

$$1.1 \cdot \log 3 \cdot \frac{\log\log m}{\log g} < \frac{1.75 \cdot \log\log m - 0.005 \cdot \log\log m}{\log g} < 1.75 \cdot \log_g \left(\frac{1.1}{2} \log m \right).$$

Therefore, equation (7.2) holds and we have completed the proof. □

8. Updates. Since the original submission of this paper, additional connections between bounded query classes and NP-approximation problems have been discovered. Chang [11] showed that approximating clique size is actually complete for certain bounded query classes. This completeness produces reductions between NP-approximation problems (e.g., from approximating the chromatic number to approximating clique size). In addition, bounded query classes can also be used to measure the difficulty of finding the vertices of an approximate clique, not just its size [12]. Also, Crescenzi et al. [16] have shown that finding the vertices of the largest clique cannot be reduced to finding the vertices of an approximate clique unless the polynomial hierarchy collapses. Bounded query classes have also been used to measure the hardness of optimization problems [17] and to compare various kinds of approximation preserving reductions [16].

Acknowledgments. The authors would like to thank Richard Beigel and Suresh Chari for many helpful discussions. Thanks also go to Samir Khuller, Martin Kummer, and Frank Stephan for proofreading drafts of this paper.

REFERENCES

[1] A. Amir, R. Beigel, and W. I. Gasarch, *Some connections between bounded query classes and non-uniform complexity*, in Proc. 5th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 232–243.

[2] A. Amir and W. I. Gasarch, *Polynomial terse sets*, Inform. and Comput., 77 (1988), pp. 37–56.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation problems*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 14–23.

[4] S. Arora and S. Safra, *Probabilistic checking of proofs*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 2–13.

[5] L. Babai, L. Fortnow, and C. Lund, *Non-deterministic exponential time has two-prover interactive protocols*, Comput. Complexity, 1 (1991), pp. 3–40.

[6] R. Beigel, *A structural theorem that depends quantitatively on the complexity of SAT*, in Proc. 2nd Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 28–32.

[7] R. Beigel, *Bounded queries to SAT and the Boolean hierarchy*, Theoret. Comput. Sci., 84 (1991), pp. 199–223.

[8] M. Bellare, S. Goldwasser, C. Lund, and A. Russell, *Efficient probabilistically checkable proofs and applications to approximations*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 294–304.

[9] M. Bellare and M. Sudan, *Improved non-approximability results*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1994, pp. 184–193.

[10] R. B. Boppana and M. M. Halldórsson, *Approximating maximum independent sets by excluding subgraphs*, BIT, 32 (1992), pp. 180–196.

[11] R. Chang, *On the query complexity of clique size and maximum satisfiability*, in Proc. 9th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 31–42; J. Comput. System Sci., to appear.

[12] R. Chang, *Structural complexity column: A machine model for NP-approximation problems and the revenge of the Boolean hierarchy*, Bull. European Assoc. Theoret. Comput. Sci., 54 (1994), pp. 166–182.

[13] R. Chang and J. Kadin, *The Boolean hierarchy and the polynomial hierarchy: A closer connection*, SIAM J. Comput., 25 (1996), pp. 340–354.

[14] R. Chang, J. Kadin, and P. Rohatgi, *On unique satisfiability and the threshold behavior of randomized reductions*, J. Comput. System Sci., 50 (1995), pp. 359–373.

[15] A. Cohen and A. Wigderson, *Dispersers, deterministic amplification, and weak random sources*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 14–19.

[16] P. Crescenzi, V. Kann, R. Silvestri, and L. Trevisan, *Structure in approximation classes*, in Proc. 1st Computing and Combinatorics Conference, Lecture Notes in Comput. Sci. 959, Springer-Verlag, Berlin, 1995, pp. 539–548.

[17] P. Crescenzi and L. Trevisan, *On approximation scheme preserving reducibility and its applications*, in Proc. 14th Conference on the Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci. 880, Springer-Verlag, Berlin, 1994, pp. 330–341.

[18] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy, *Approximating clique is almost NP-complete*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 2–12.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[20] W. Gasarch, M. W. Krentel, and K. Rappoport, *OptP-completeness as the normal behavior of NP-complete problems*, Math. Systems Theory, 28 (1995), pp. 487–514.

[21] A. Hoene and A. Nickelsen, *Counting, selecting, sorting by query-bounded machines*, in Proc. 10th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 665, Springer-Verlag, Berlin, 1993, pp. 196–205.

[22] D. S. Johnson, *Approximation algorithms for combinatorial problems*, J. Comput. System Sci., 9 (1974), pp. 256–278.

[23] J. Kadin, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263–1282.

[24] M. W. Krentel, *The complexity of optimization problems*, J. Comput. System Sci., 36 (1988), pp. 490–509.

[25] L. Lovász, *On the ratio of optimal integral and fractional covers*, Discrete Math., 13 (1975), pp. 383–390.

[26] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic methods for interactive proof systems*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 2–10.

[27] C. Lund and M. Yannakakis, *On the hardness of approximating minimization problems*, J. Assoc. Comput. Mach., 41 (1994), pp. 960–981.

[28] P. Rohatgi, *Saving queries with randomness*, J. Comput. System Sci., 50 (1995), pp. 476–492.

[29] A. Shamir, *IP = PSPACE*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 11–15.

[30] H. U. Simon, *On approximate solutions for combinatorial optimization problems*, SIAM J. Discrete Math., 3 (1990), pp. 294–310.

[31] M. Sipser, *Expanders, randomness, or time versus space*, in Structure in Complexity Theory, Lecture Notes in Comput. Sci. 223, Springer-Verlag, Berlin, 1986, pp. 325–329.

[32] K. Wagner, *More complicated questions about maxima and minima and some closures of NP*, in Proc. 13th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 226, Springer-Verlag, Berlin, 1986, pp. 434–443.

[33] K. Wagner and G. Wechsung, *On the Boolean closure of NP*, in Proc. 1985 International Conference on Fundamentals of Computation Theory, Lecture Notes in Comput. Sci. 199, Springer-Verlag, Berlin, 1985, pp. 485–493.

[34] D. Zuckerman, *NP-complete problems have a version that's hard to approximate*, in Proc. 8th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 305–312.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 210–230, February 1997

012

SPARSE DYNAMIC PROGRAMMING FOR EVOLUTIONARY-TREE COMPARISON*

MARTIN FARACH† AND MIKKEL THORUP‡

Abstract. Constructing evolutionary trees for species sets is a fundamental problem in biology. Unfortunately, there is no single agreed upon method for this task, and many methods are in use. Current practice dictates that trees be constructed using different methods and that the resulting trees should be compared for consensus. It has become necessary to automate this process as the number of species under consideration has grown. We study one formalization of the problem: the *maximum agreement-subtree* (`MAST`) *problem*.

The `MAST` problem is as follows: given a set A and two rooted trees $\mathcal{T}_0$ and $\mathcal{T}_1$ leaf-labeled by the elements of A, find a maximum-cardinality subset B of A such that the topological restrictions of $\mathcal{T}_0$ and $\mathcal{T}_1$ to B are isomorphic. In this paper, we will show that this problem reduces to unary weighted bipartite matching (`UWBM`) with an $O(n^{1+o(1)})$ additive overhead. We also show that `UWBM` reduces linearly to `MAST`. Thus our algorithm is optimal unless `UWBM` can be solved in near linear time. The overall running time of our algorithm is $O(n^{1.5}\log n)$, improving on the previous best algorithm, which runs in $O(n^2)$. We also derive an $O(nc^{\sqrt{\log n}})$-time algorithm for the case of bounded degrees, whereas the previously best algorithm runs in $O(n^2)$, as in the unbounded case.

Key words. sparse dynamic programming, computational biology, evolutionary trees

AMS subject classifications. 05C05, 05C85, 05C90, 68C25, 92B05

PII. S0097539794262422

1. Introduction.

An evolutionary tree, or phylogeny, is a model of the evolutionary history for a set of species. Constructing such trees from observations on a set of living species is one of the fundamental tasks of computational biology. This is because the evolutionary relation of species provides a great deal of information about their biochemical machinery. For example, RNA's secondary structure is most accurately determined by selecting correlated mutations of a class of related species.

To construct a tree from a set of species, one must have a model of what makes one tree better than another. Many criteria have been proposed, but in general, these turn out to be NP-hard to optimize [15, 24]. There is also no consensus in the biology community as to what makes a good tree. As is typically the case when there is no really good solution to a problem, the number of solutions actually in use is quite large. Within the biology literature, various heuristics have been proposed (see, e.g., [8, 9, 11, 19, 22, 20]). More recently, a variety of solutions have been examined rigorously [1, 6, 16]. Not surprisingly, these various methods do not always give the same answer on the same inputs. Given that there is no "gold standard" for constructing evolutionary trees, current practice dictates that several different methods be applied to the data. The resulting trees may agree in some parts and

* Received by the editors January 27, 1994; accepted for publication (in revised form) April 26, 1995. A preliminary version of this paper appeared in *Proc.* 1994 *Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1994.
http://www.siam.org/journals/sicomp/26-1/26242.html

† DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), Rutgers University, Box 1179, Piscataway, NJ 08855 (farach@cs.rutgers.edu). The research of this author was supported by DIMACS, a National Science Foundation Science and Technology Center, under NSF contract STC-8809648.

‡ Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark (mthorup@diku.dk). The research of this author was supported by the Danish Technical Research Council, by the Danish Research Academy, and by DIMACS under NSF contract STC-8809648. Most of the research in this paper was done while this author was visiting DIMACS.

differ in others. In general, one is interested in finding the largest set of species on which the trees agree [18]. In other settings, a particular method may be applied to different data sets for the same set of species [13] or on a single data set which has been permuted some number of times for statistical analysis [9]. The resulting trees are then compared in order to arrive at some consensus. Many consensus techniques have been proposed and are currently in use (see [5] for a review). One of the most extensively studied consensus methods was defined by Finden and Gordon [10] as follows.

Let A be a set of species. We will define an *evolutionary tree* T on A to be a rooted tree with no degree-1 nodes such that the leaves of T are uniquely labeled with the elements of A. In such a tree, the leaves represent the species under consideration, and the internal nodes represent posited ancestors. Now suppose that we are given two trees T_0 and T_1 which are evolutionary trees on the same species set A. If the two trees differ, it is reasonable to ask for the "intersection" of the information contained in the trees. By viewing the input trees as the outcomes of experiments performed to discover the history of some species, we will typically have more confidence in information given in the "intersection" than in information unique to each tree. But what is the intersection of two evolutionary trees?

Finden and Gordon's answer to this question involves the notion of a "topological restriction" of an evolutionary tree to a subset of the species. Given an evolutionary tree T on set A and given $B \subseteq A$, then the *topological restriction of* T *to* B, written $T|B$, is the evolutionary tree on B such that B has the same history in $T|B$ as it does in T. More formally, first, given any rooted tree T, by a *topological subtree of* T, we mean a rooted tree U with no degree-1 nodes, obtained from a normal subtree S of T by replacing dipaths with single arcs. That is, U can be obtained from the subtree S by repetition of the following operation: if a vertex v has only one child w, we may delete ("jump") v, making the original parent of v the parent of w. Note that the topological subtree U is uniquely defined in terms of its leaf set. The full vertex set of U is the closure of the leaf set under the least-common-ancestor operation in T. Now, formally, $T|B$ denotes the topological subtree of T whose leaves are the the leaves of T with labels in B. This restriction operator immediately implies the following similarity measure on trees.

Problem. The maximum agreement-subtree (`MAST`) problem.

Input. A pair (T_0, T_1) of evolutionary trees for some common set A of species.

Output. A maximum-cardinality subset B of A such that $T_0|B$ and $T_1|B$ have a leaf-label preserving isomorphism.

Finden and Gordon [10] gave a heuristic method for computing the maximum agreement subtree of two binary trees. However, their algorithm, which has an $O(n^5)$ running time, does not guarantee an optimal solution. In [17], Kubicka et al. presented an $O(n^{(\frac{1}{2}+\epsilon)\log n})$-time algorithm for the binary `MAST` problem. Steel and Warnow [21] gave the first polynomial algorithm, which we will refer to as SW. The SW algorithm is a dynamic-programming approach which runs in $O(n^2)$-time on bounded-degree trees and in $O(n^{4.5}\log n)$ time on unbounded-degree trees. We showed in [7] that the SW algorithm can be modified to yield an $O(n^2)$-time algorithm for the unbounded case.

Both the SW algorithm and our modification of it perform some computation—a weighted bipartite matching—for each pair of nodes from the input trees. Hence this approach cannot give a $o(n^2)$-time algorithm for the `MAST` problem. We therefore run into the dynamic-programming bottleneck which is endemic in computational molecular biology. A wide variety of biocomputing problems, e.g., in the string-

edit-distance problem, RNA secondary structure, etc., have solutions which involve dynamic programming. To avoid the too costly quadratic complexity of these algorithms, researchers have either turned to heuristics (e.g., the BLAST program [2]) or to the design of *input-sensitive* algorithms. Notable in the latter class is the algorithm of Hunt and Szymanski [14]. This latter class of algorithms has the property that they speed up various dynamic programs to almost linear time in the *best case* but have at least quadratic-time worst cases.

In this paper, we use sparsity conditions to break the dynamic-programming bottleneck and have subquadratic running times in the worst case. In fact, we show tight bounds. Our main result is an algorithm which solves MAST within the same asymptotic time bound as that for solving *unary weighted bipartite matching* (UWBM), i.e., a weighted bipartite matching where the size of the input is measured as the sum of the weights of all edges—so unweighted bipartite matching is a special case (see [12] for definitions of matchings, etc.). More precisely, we show that $\texttt{time}(\texttt{MAST}(n)) = O(n^{1+o(1)} + \texttt{time}(\texttt{UWBM}(n)))$.[1] Using the best known algorithm [12] for weighted bipartite matching, this gives us an $O(n^{1.5} \log n)$-time algorithm for the MAST problem, thus breaking the $\Omega(n^2)$ bottleneck. If the degrees are bounded, our algorithm runs in $O(nc^{\sqrt{\log n}}) = O(n^{1+o(1)})$ time, beating the previous $O(n^2)$ bound [21] for this case.

While UWBM does not often appear as a natural upper bound, and typically one sees reference to either unweighted or fully weighted bipartite matching instead, we show that the unary weighting is inherent in bounding the complexity of MAST by observing that, in fact, $\texttt{time}(\texttt{MAST}(n)) = \Omega(\texttt{time}(\texttt{UWBM}(n)))$. Thus, for all intents and purposes, our reduction is optimal since getting the complexity of UWBM, or just bipartite matching, down anywhere near $O(n^{1+o(1)})$ is a long-standing open problem.

The fact that our algorithm works by sparsity means that we identify a small set of *significant computations* in the dynamic program. The exact size of this set depends on the running time of the bipartite-matching algorithm used; thus we carefully balance the time spent at a single node pair in the dynamic program with the number of such node pairs that we evaluate. The key to this balancing is the *parameterized core trees* which we introduce in this paper. The core tree is a generalization of the separator of a tree, which will turn out to be useful in guiding our computation.

Finally, we note that two variants of the MAST problem have been investigated. First, the unrooted version was the primary focus of the Steel and Warnow paper [21]. The above cited complexities for their algorithms ($O(n^{4.5} \log n)$ for unbounded degrees and $O(n^2)$ for bounded degrees) apply to the unrooted case. However, they noticed that the rooted case reduces to the unrooted case and hence that these complexities carry over to the rooted case. In [7], our main result was to improve the complexity of the unbounded unrooted case to $O(n^2 c^{\sqrt{\log n}})$. We believe that the unrooted case is much harder than the rooted case and know of no $O(n^2)$ algorithm for this problem. Another variant is that of considering three or more trees. Amir and Keselman [3] showed that the MAST problem on three or more trees is NP-hard for trees of unbounded degree, while polynomial if just one of the trees has bounded degree. While the constant-degree restriction may be suitable in some settings, the unbounded-degree algorithm is important since there are many tree-construction techniques available which place no restrictions on the degree of the trees produced [22].

Another problem related to our MAST problem is the tree-homeomorphism prob-

[1] This complexity is understood to be modulo a class of "well-behaved" functions explicitly defined in Theorem 4.6.

lem. In [4], Chung presents an $O(n^{2.5})$ algorithm for the rooted-tree-homeomorphism problem of deciding whether one rooted tree (no leaf labels) is a topological subtree of another. This algorithm is quite similar to the SW algorithm. Unfortunately, none of our optimizations to the SW algorithm apply to the tree-homeomorphism problem since they all rely on the restriction that the agreement isomorphism should be leaf-label preserving.

In section 2, we show the reduction from UWBM to MAST. In section 3, we give some basic definitions and an overview of our algorithm. In section 4, we prove the correctness of the algorithm and give a general sketch of its time complexity. In section 5, we reduce the number of comparisons in the dynamic program and finish the time analysis for the bounded-degree case. In section 6, we describe how to reduce the work of the matchings and finish the time analysis for the unbounded-degree case.

2. Lower bound. We show a reduction from a size-n UWBM problem to an $O(n)$-leaf MAST problem. We define the size of a UWBM instance to be the sum of the edge weights; hence an n-size UWBM instance can code a WBM instance with total integer edge weight n and up to $O(n)$ edges and $O(n)$ nodes. Using the the best algorithm known [12] for WBM, such a problem can be solved in $O(n^{1.5} \log n)$.

We now show a reduction from UWBM to MAST. Given a connected weighted bipartite graph $G = (U \cup V, E, W : E \to \mathbb{N})$ such that $\sum_{e \in E} W(e) = n$, construct evolutionary trees T_U and T_V with $O(n)$ labels as follows. For each $X \in \{U, V\}$, set r_X to be the root of T_X, and for each $x \in X$, create a child c_x of r_X. For each $e = \{u, v\} \in E$, add $W(e)$ leaf children to c_u in T_U and label them $\langle x, y, i \rangle$, $1 \le i \le W(e)$. Add leaf children with the same labels to c_v in T_V. Finally, pick $n + 2$ new labels and build a star tree $\mathcal{S}$ on the labels. Attach the root of one copy of $\mathcal{S}$ to the root of T_X and attach the root of another copy of $\mathcal{S}$ to the root of T_Y. The size of the star $\mathcal{S}$ guarantees that any solution to MAST will map the roots to each other. Hence there is a bijection between the maximum-weight matchings M and the maximum-agreement subsets B such that if $\{v, w\} \in M$ then $\mathtt{A}(c_v) \cap \mathtt{A}(c_w) \subseteq B$.

THEOREM 2.1. *There is a linear reduction from* UWBM *to* MAST. □

In [3], Amir and Keselman use a similar reduction from *three-dimensional matching* to prove that the MAST problem on three or more trees is NP-Hard.

3. Ideas and outline.

3.1. Preliminaries. In this subsection, we will describe a rooted version of the SW algorithm. This algorithm will form the basis of our discussion of more efficient algorithms. For simplicity, all algorithms presented will compute only the maximal cardinality of an agreement set, i.e., the cardinality of the output set for the MAST problem. However, after this cardinality has been found, the computation can easily be traced back in order to derive a concrete set. In this subsection, we will also introduce the main notation and terminology for the rest of the paper.

Given an evolutionary tree T, by $\mathtt{V}(T)$ we denote the set of its vertices, by $\mathtt{r}(T)$ its root, by $\mathtt{A}(T)$ the set of its leaf labels. For $v \in \mathtt{V}(T)$, let $\mathtt{p}(v)$ be the parent of v, $\mathtt{C}(v)$ be the set of its children, and $\mathtt{d}(v)$ be its degree. If $v \in \mathtt{V}(T)$, then $\mathtt{t}(v)$ denotes the subtree descending from v. For a set of nodes $V \subseteq \mathtt{V}(T)$, we define $\mathtt{LCA}(V)$ to be the closure of V with respect to least common ancestors.

For the rest of the paper, fix an instance of the MAST problem consisting of two evolutionary trees $\mathcal{T}_0$ and $\mathcal{T}_1$ for some common set A of species. We measure the size n of the problem as the cardinality of A, which equals the number of leaves for both $\mathcal{T}_0$ and $\mathcal{T}_1$. To simplify some boundary cases in the discussion below, we allow the situation where $\mathcal{T}_0$ and $\mathcal{T}_1$ are empty. In this case, $\mathtt{MAST}(\mathcal{T}_0, \mathcal{T}_1) = |\mathtt{A}| = n = 0$. Finally,

fix an arbitrary left to the right ordering of the children of each node. Thus $\mathcal{T}_0$ and $\mathcal{T}_1$ are henceforth considered ordered.

The roles of the evolutionary trees $\mathcal{T}_0$ and $\mathcal{T}_1$ are symmetric. In order to avoid unnecessary repetitions in our definitions, we will commonly use $\bar{0}$ to mean 1 and vice versa. Also, we introduce the generic term *opposing pair*, by which we refer to a pair $\{x, y\}$ where x is contained in $\mathcal{T}_i$ and y is contained in $\mathcal{T}_{\bar{i}}$. Here x and y might be of different types. For example, x might be a vertex of $\mathcal{T}_i$ while y is a subset of the vertices of $\mathcal{T}_{\bar{i}}$.

For any opposing pair $\{v, w\}$ of vertices, let $\mathtt{mast}\{v, w\}$ denote the MAST of the subtrees rooted, respectively, at v and w. Here the subtrees are understood to be topologically restricted to the intersection of their label sets. Thus $\mathtt{mast}\{v, w\}$ is the MAST of $\mathcal{T}_0|B$ and $\mathcal{T}_1|B$, where $B = \mathtt{A}(\mathtt{t}(v)) \cap \mathtt{A}(\mathtt{t}(w))$.

The following lemma appears with some minor modifications in [21] and is the basis for their dynamic-program approach to the unrooted version of this problem.

LEMMA 3.1 (see [21]). *For all $v \in V(\mathcal{T}_0)$ and $w \in V(\mathcal{T}_1)$,*

$$\mathtt{mast}\{v, w\} = \begin{cases} |\mathtt{A}(\mathtt{t}(v)) \cap \mathtt{A}(\mathtt{t}(w))| & \text{if } v \text{ or } w \text{ is a leaf;} \\ \max\{\mathtt{Diag}\{v, w\}, \mathtt{match}\{v, w\}\} & \text{otherwise,} \end{cases}$$

where $\mathtt{Diag}(v, w) = \{\mathtt{mast}\{v, w_1\} | w_1 \in \mathtt{C}(w)\} \cup \{\mathtt{mast}\{v_0, w\} | v_0 \in \mathtt{C}(v)\}$ *and where* $\mathtt{match}\{v, w\}$ *is the value of the maximum-weight matching of the weighted bipartite graph* $((\mathtt{C}(v) \cup \mathtt{C}(w), \mathtt{C}(v) \times \mathtt{C}(w), \mathtt{mast}\{\cdot, \cdot\})$.

The lemma is illustrated in Figure 3.1. It is clear from this lemma that we need some values on subtrees in order to compute the mast of two nodes. To simplify the discussion, we introduce the following notation. By the *base pairs* of an opposing vertex pair $\{x_0, x_1\}$, we understand the set of opposing pairs $\{w_0, w_1\}$ such that either $w_0 \in \mathtt{C}(v_0)$ and $w_1 \in \mathtt{C}(v_1) \cup \{v_1\}$ or, conversely, $w_0 \in \mathtt{C}(v_0) \cup \{v_0\}$ and $w_1 \in \mathtt{C}(v_1)$. By the *base* of $\{x_0, x_1\}$, we mean the set of values $\mathtt{mast}\{w_0, w_1\}$ for all base pairs of $\{x_0, x_1\}$. Thus the base forms the set of values needed by a dynamic program to compute the mast values for $\{x_0, x_1\}$.

Later in the paper, we will need a generalized version of a base defined over pairs of opposing *sets* of vertices. First, for any set V of vertices in a tree, set $\mathtt{C}(V) = \bigcup\{\mathtt{C}(v) | v \in V\} \setminus V$. We call the members of $\mathtt{C}(V)$ the *proper children* of V. Now if V_0 and V_1 are subsets of $\mathtt{V}(\mathcal{T}_0)$ and $\mathtt{V}(\mathcal{T}_1)$, then the *base pairs* of $\{V_0, V_1\}$ are the opposing vertex pairs $\{w_0, w_1\}$ such that either $w_0 \in \mathtt{C}(V_0)$ and $w_1 \in \mathtt{C}(V_1) \cup V_1$ or, conversely, $w_0 \in \mathtt{C}(V_0) \cup V_0$ and $w_1 \in \mathtt{C}(V_1)$. By the *base* of $\{V_0, V_1\}$, we mean the values $\mathtt{mast}\{w_0, w_1\}$ for all base pairs of $\{V_0, V_1\}$. Thus the base of $\{V_0, V_1\}$ contains all the values needed for a dynamic bottom-up computation of $\mathtt{mast}\{v_0, v_1\}$ for all $v_0 \in V_0$ and $v_1 \in V_1$.

Lemma 3.1 suggests the following dynamic-programming algorithm for the MAST problem.

ALGORITHM A: first algorithm for MAST:

A.1. Input $\mathcal{T}_0$ and $\mathcal{T}_1$.

A.2. Let $\mathcal{O}$ be the lexicographic ordering of $\mathtt{V}(\mathcal{T}_0) \times \mathtt{V}(\mathcal{T}_1)$, where the vertices in each $\mathcal{T}_i$ are postordered.

A.3. For each (v_0, v_1) in increasing order in $\mathcal{O}$ do

A.3.1. Compute $\mathtt{mast}\{v_0, v_1\}$.

A.4. Return $\mathtt{mast}\{\mathtt{r}(\mathcal{T}_0), \mathtt{r}(\mathcal{T}_1)\}$.

Algorithm A computes MAST by applying Lemma 3.1 to all the $O(n^2)$ pairs of opposing vertex pairs. The bottom-up ordering of $\mathcal{O}$ guarantees that the base of a

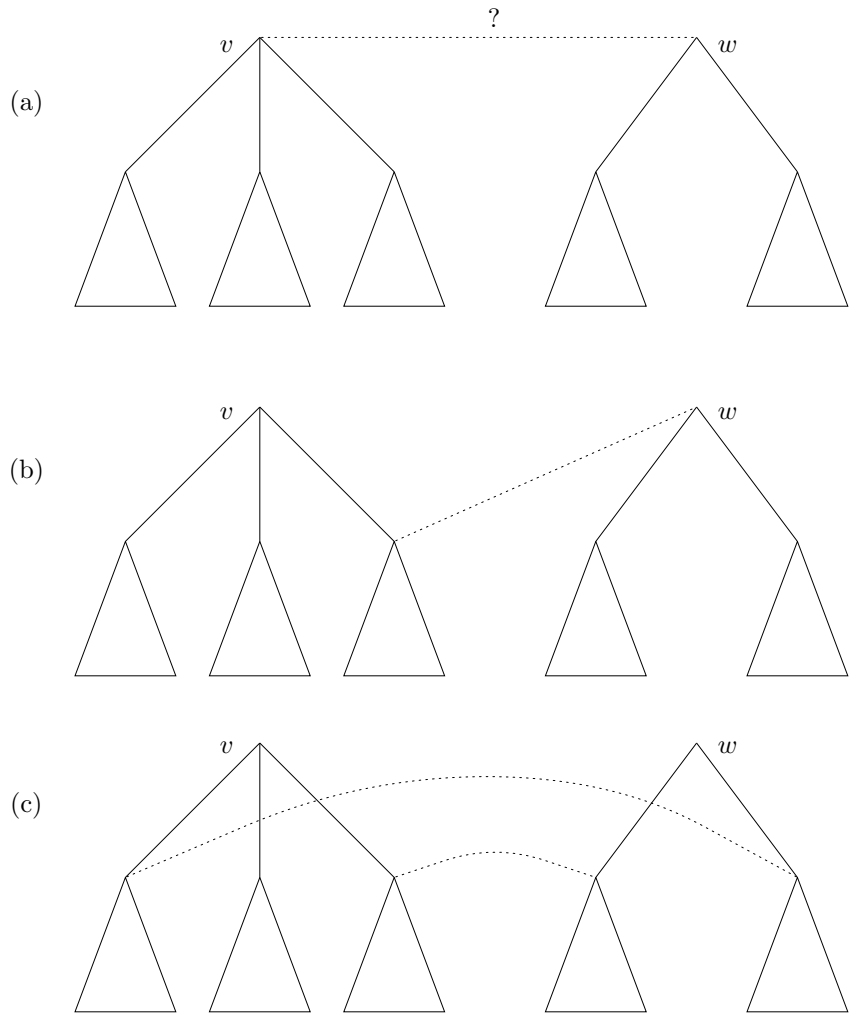


FIG. 3.1. *In order to find* $\texttt{mast}\{v, w\}$ *as in* (a)*, we can either try a "diagonal," as in* (b)*, or we can find a maximum-weight matching between the sets of children, as in* (c).

node pair is ready whenever `mast` is evaluated. For bounded degree, `mast` can be computed in $O(1)$ time, thus giving $O(n^2)$ total work. For unbounded degree, the bottleneck in the comparisons is the matchings, which sum up to $O(\sum_{v \in \texttt{V}(T_1), w \in \texttt{V}(T_2)} \texttt{d}(v) \cdot \texttt{d}(w) \sqrt{\texttt{d}(v) + \texttt{d}(w)} \log n) = O(n^2 \sqrt{n} \log n)$ using Gabow and Tarjan's matching algorithm [12]. Note that the Gabow–Tarjan algorithm is not only the fastest known for normal weighted bipartite matching where the weights are assumed to be in a binary encoding, but it is also the fastest known for the case of unary weighted matching, i.e., the case where the input is measured as the sum of all of the edge weights in the graph. It is in this latter sense that we are interested in the Gabow–Tarjan algorithm.

In [7], we showed that most matching graphs can be preprocessed so that they are quite sparse. We were able to show a bound of $O(n^2)$ time for all computations of `mast`, thus showing that the `MAST` of two trees can be computed in $O(n^2)$, even when the degree is unbounded.

We improve this result by two types of sparsification. First of all, we reduce

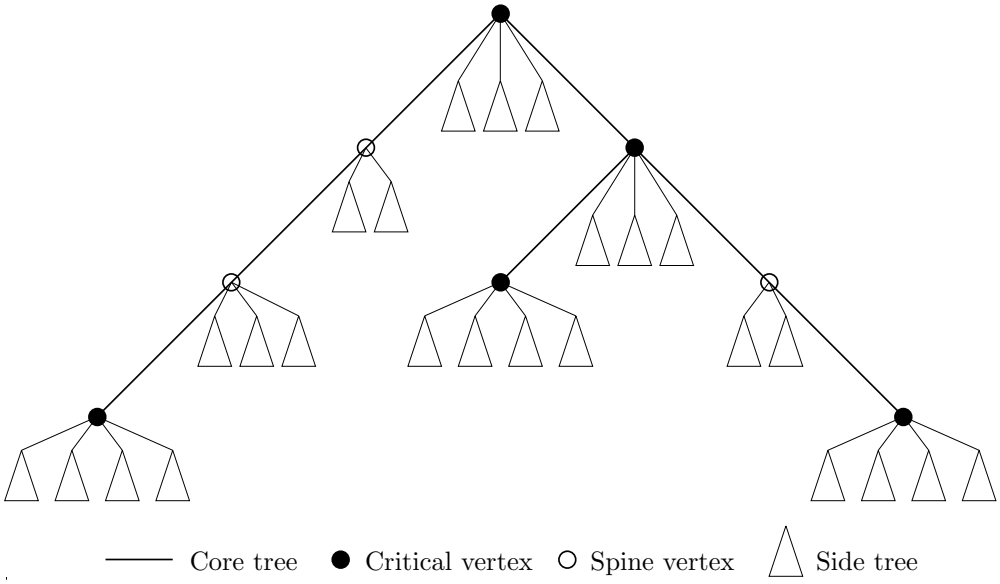


FIG. 3.2. *The core-tree-related concepts.*

the number of `mast` values evaluated in the dynamic program. Second, we reduce the work done in the matchings. This second phase is analogous to our previous work [7] in reducing the matching work, but here we require stronger techniques in order to achieve optimality in the sense of equivalence to *one* unary weighted bipartite matching. For the moment, we will focus entirely on reducing the number of comparisons, returning to weighted matchings in section 6.

3.2. A faster algorithm. Our key to sparsifying the dynamic program is the *core tree*, which is defined in terms of some parameter κ (which will turn out to be 16 for unbounded degrees and $4^{\sqrt{\log n}}$ for bounded degrees). We say that a node is a *core node* if it has more than n/κ descendant leaves. Otherwise, it is a *side node*. Then the *core tree* is the component induced by the core nodes and the *side trees* are the components induced by side nodes. We denote by $\mathcal{U}_i$ the core tree of tree $\mathcal{T}_i$. Note that the core tree is indeed a tree since it is connected. Further, note that the core tree has at most κ leaves since the leaves are roots of disjoint subtrees, each with at least n/κ leaves.

We make one final distinction within the core tree. We partition the nodes of the core tree into *critical nodes* and *spine nodes*. A critical node is either the root, a leaf of the core tree, or a branching node, i.e., a core node with at least two children which are core nodes. If a core node is not a critical node, then it is a spine node. Notice that the spine nodes can be further partitioned into connected components. In fact, each such connected component forms a chain of nodes where all nodes but the last have exactly one core child—but possibly $\Omega(n)$ side children. We call each such component a *spine*. The concepts are illustrated at Figure 3.2. Note that we have $O(\kappa)$ critical nodes and spines but $O(n)$ spine nodes. The following trivial fact gives one of the main uses of side trees.

FACT 3.2. *Let $S_1, \ldots, S_x$ and $S'_1, \ldots, S'_y$ be partitionings of the side trees of $\mathcal{T}_0$ and $\mathcal{T}_1$, respectively. Let $l_{ij} = |\bigcup_{t \in S_i} \mathtt{A}(t) \cap \bigcup_{t \in S'_j} \mathtt{A}(t)|$. Then $\sum_{i=1}^{x} \sum_{j=1}^{y} l_{ij} = n$.*

Given the core trees, we naturally divide the `MAST` computation into two phases.

First, we compute the base of the opposing core trees, that is, `mast` values of opposing vertex pairs where one is the root of a side tree and the other is either the root of a side tree or a core vertex. Second, we concentrate on computing `mast` values for opposing pairs of core nodes, including the root pair. For the base computation, we note that the base of the core trees is of size $\Theta(n^2)$, so we cannot afford to compute the whole base. Instead, we will build a data structure allowing us to retrieve any desired base value quickly. More specifically, we will implement the following procedure.

PROCEDURE 1. CORE-BASE: *This procedure computes a data structure such that every* `mast` *value in the base of the opposing core trees, i.e., of* $\{\mathtt{V}(\mathcal{U}_0), \mathtt{V}(\mathcal{U}_1)\}$*, can be determined in* $O(\log n)$ *time.*

This data structure will be computed by a routine which recurses on all side trees. Thus within the recursion, all subproblems considered are of size proportional to that of the side trees ($O(n/\kappa)$). Using this fact, we will show that for unbounded-degree trees, the recursion will have no asymptotic consequence for the overall computation time. More precisely, we will show that we have a dominating bottleneck in the maximum-weighted-matching evaluations in computing `match` for opposing pairs of core vertices. For bounded-degree trees, `match` takes constant time, and in this case, it will turn out that the recursion contributes to the overall running time by multiplicative factor of $O(c^{\sqrt{\log n}})$ for some constant c.

Second, we will find an efficient implementation of the following procedure.

PROCEDURE 2. CORE-TREES: *Given the base of the opposing core trees (which we can compute as needed by the* CORE-BASE *data structure), this procedure computes* $\mathtt{mast}\{\mathtt{r}(\mathcal{U}_0), \mathtt{r}(\mathcal{U}_1)\} = \mathtt{mast}\{\mathtt{r}(\mathcal{T}_0), \mathtt{r}(\mathcal{T}_1)\} = \mathtt{MAST}(\mathcal{T}_0, \mathcal{T}_1)$.

This procedure will be implemented by a sparse version of the dynamic program described in Algorithm A. In $O(\kappa n \ \mathrm{polylog}\, n)$ time, it will select $O(\kappa n)$ significant `mast` values to be computed, including, in particular, $\mathtt{mast}\{\mathtt{r}(\mathcal{U}_0), \mathtt{r}(\mathcal{U}_1)\}$. For bounded degrees, each of these `mast` values can be computed in constant time. For unbounded degrees, it will be shown that they can be computed in the same total time as that of *one* unary weighted bipartite matching of size $O(n)$. Thus, very generally, `MAST` is implemented as follows.

ALGORITHM B:

B.1. Input $\mathcal{T}_0$ and $\mathcal{T}_1$.

B.2. Find their core trees $\mathcal{U}_0$ and $\mathcal{U}_1$.

B.3. Compute CORE-BASE.

B.4. Compute CORE-TREES.

B.5. Return $\mathtt{mast}\{\mathtt{r}(\mathcal{T}_0), \mathtt{r}(\mathcal{T}_1)\}$.

As noted above, a naïve algorithm would simply apply Lemma 3.1 to each core–core pair. This would yield once again an $\Omega(n^2)$ algorithm, the bottleneck of which is in comparing spines. The following procedure will be used to circumvent this bottleneck. We will use it to complete a sketch of CORE-TREES.

PROCEDURE 3. SPINES$\{S_0, S_1\}$: *For opposing spines* S_0 *and* S_1*, this procedure computes* $\mathtt{mast}\{\mathtt{r}(S_i), v_{\bar{i}}\}$ *for* $i = 0, 1$*, and for all* $v_{\bar{i}} \in \mathtt{V}(S_{\bar{i}})$.

With this procedure we get the following algorithm for CORE-TREES.

ALGORITHM C:

C.1. For $i \in \{0, 1\}$, let $\mathcal{U}_i'$ denote the tree obtained by identifying each spine of $\mathcal{U}_i$ with a single vertex.

C.2. Let $\mathcal{O}$ be the lexicographic ordering of $\mathtt{V}(\mathcal{U}_0') \times \mathtt{V}(\mathcal{U}_1')$ where the vertices in each $\mathcal{U}_i'$ are postordered.

C.3. For each (c_0, c_1) in increasing order in $\mathcal{O}$ do
C.3.1. Case: c_0 and c_1 are critical. Do
C.3.1.1. Compute $\texttt{mast}\{c_0, c_1,\}$.
C.3.2. Case: c_i is a spine node and $c_{\bar{i}}$ is a critical node, for $i \in \{0,1\}$. Do:
C.3.2.1. Let $s_1, \ldots, s_k$ be the spines nodes of c_i in ascending order.
C.3.2.2. For $j \leftarrow 1$ to k compute $\texttt{mast}\{c_{\bar{i}}, s_j\}$.
C.3.3. Case: c_0 and c_1 are spine representing nodes. Do:
C.3.3.1. Compute SPINES$\{c_0, c_1\}$.

OBSERVATION 3.3. CORE-TREES *makes* $O(\kappa n)$ *direct* `mast` *computations* (*Steps* C.3.1.1 *and* C.3.2.2) *and calls* SPINES $O(\kappa^2)$ *times* (*Step* C.3.3.1). *All other processing is done in time* $O(\kappa n)$.

4. Computing CORE-BASE. The goal of CORE-BASE is to preprocess $\mathcal{T}_0$ and $\mathcal{T}_1$ so that we may quickly retrieve values from the base of the core trees, that is, `mast` values of opposing vertex pairs where one is the root of a side tree and the other is either the root of a side tree or a core vertex. This will be done recursively using the following extension of `MAST`, which for both roots computes the `mast` value against all opposing vertices.

PROCEDURE 4. TREES$\{T_0, T_1\}$, *where each* T_i *is an evolutionary tree: For* $i = 0, 1$, *this procedure computes* $\texttt{mast}\{\texttt{r}(T_i), v_{\bar{i}}\}$ *for all* $v_{\bar{i}} \in \texttt{V}(T_{\bar{i}})$.

Assuming an appropriate implementation of TREES, we will apply it to each side tree s against the opposing tree restricted to the label set of s. Thus all problems considered are of size $O(n/\kappa)$. The following lemma shows that such a computation for all side trees essentially gives us the whole base of the core trees, or even more: for each root of a side tree, it allows us to derive the `mast` value against any opposing vertex.

LEMMA 4.1. *Let* s *be a side tree of* $\mathcal{T}_i$, $t_s = \mathcal{T}_{\bar{i}}|\texttt{A}(s)$, *and let* $W = \texttt{V}(t_s)$. *Then for all* $v \in \texttt{V}(\mathcal{T}_{\bar{i}})$, $\texttt{mast}\{\texttt{r}(s), v\} = 0$ *if* v *has no descendant in* W; *otherwise,* $\texttt{mast}\{\texttt{r}(s), v\} = \texttt{mast}\{\texttt{r}(s), w\}$, *where* w *is the unique first descendant of* v *in* W.

Proof. Set $B = \texttt{A}(s) \cap A(\texttt{t}(v))$. Then $\texttt{V}(\mathcal{T}_{\bar{i}}|B)$ is exactly the set of descendants of v in $W = \texttt{V}(t_s) = \texttt{V}(\mathcal{T}_{\bar{i}}|\texttt{A}(s))$. By definition, $\texttt{mast}\{\texttt{r}(s), v\} = \texttt{MAST}\{\mathcal{T}_0|B, \mathcal{T}_1|B\}$. Thus $\texttt{mast}\{\texttt{r}(s), v\} = 0$ if $B = \emptyset$, but then v has no descendants in W. We may therefore assume that $B \neq \emptyset$. Then $\texttt{r}(\mathcal{T}_{\bar{i}}|B)$ is the first descendant of v in W. Moreover, $\texttt{A}(\texttt{t}(v)) \supseteq \texttt{A}(\texttt{t}(\texttt{r}(\mathcal{T}_{\bar{i}}|B))) \supseteq B$, so $\texttt{A}(s) \cap \texttt{A}(\texttt{t}(\texttt{r}(\mathcal{T}_{\bar{i}}|B))) = B$, and hence $\texttt{mast}\{\texttt{r}(s), v)\} = \texttt{mast}\{\texttt{r}(s), \texttt{r}(\mathcal{T}_{\bar{i}}|B)\}$. This completes the proof. □

Notice that TREES is a strengthening of `MAST` in that it computes more values. Algorithm B implements TREES except for the `mast` values between the roots and their opposing side nodes. Call these missing values the *side values*. In order to make Algorithm B implement TREES completely, we extend our specification of CORE-BASE to compute the side values as well. That is, CORE-BASE should make retrievable not only the values from the base of the core trees but also these side values. However, the following trivial lemma shows that this is already done by our recursion over the side trees.

LEMMA 4.2. *For any* $v \in \texttt{V}(s)$, *where* s *is a side tree of* $\mathcal{T}_i$, $t_s = \mathcal{T}_{\bar{i}}|\texttt{A}(s)$, $\texttt{mast}\{v, \texttt{r}(\mathcal{T}_{\bar{i}})\} = \texttt{mast}\{v, \texttt{r}(t_s)\}$. □

We have now shown that we can compute all values needed for CORE-BASE by recursively applying TREES on all side trees s along with their opposing restrictions t_s. Thus the remaining question is how to implement the various steps described in an efficient manner. First, we need to compute the t_s's. The following lemma states that all t_s's can be computed in linear time since the label set of different side trees of the same evolutionary tree are disjoint.

LEMMA 4.3 (see [7]). *Let T be a rooted tree with vertex set V, and let $\{V_1, \ldots, V_k\}$ be a family of subsets of V. Then in time $O(\sum |V_i| + |V|)$, we can compute all of the topological subtrees T_i with $\mathtt{V}(T_i) = \mathtt{LCA}(V_i)$.*

Thus given a partition $L_0, \ldots, L_k$ of the labels of an evolutionary tree T of size n, we can compute all of $T|L_0, \ldots, T|L_k$ in $O(n)$ total time.

Now in order to implement the descendant operation from Lemma 4.1, we need the following technical lemma.

LEMMA 4.4. *Let T be a rooted tree with vertex set V, and let $\{W_1, \ldots, W_k\}$ be a family of subsets of V, each of which is closed under least common ancestors. Then in $O(\sum |W_i| + |V|)$ time, we can build a data structure such that given any vertex $v \in V$ and index $i \leq k$, we can return the nearest descendant of v in W_i, if any, in time $O(\log |W_i|)$.*

Proof. First, we organize the vertices of T in an Euler tour E, that is, we make a depth-first traversal from the root, noting each time we visit every vertex. For every vertex v, denote the first occurrence in E by $f(v)$ and the last occurrence in E by $l(v)$. Now w is a descendant of v if and only if $f(v) \leq f(w) < l(v)$. Next, for $i = 1, \ldots, k$, we construct the subsequence E_i^f of E containing $f(w)$ for each vertex w in W_i. Clearly, both the Euler tour and the splitting of it into the E_i^f's can all be done in time $O(\sum |W_i| + |V|)$.

Now, given a vertex $v \in V$ and index $i \leq k$, by an $O(\log |E_i^f|) = O(\log |W_i|)$-time binary search, we find the first element $f(w)$ in E_i^f greater than or equal to $f(v)$. If $f(w) \geq l(v)$, we may conclude that v has no descendant in t_s. Otherwise, $f(w) < l(v)$. Since W_i is closed under the least-common-ancestor operation, we may then conclude that w is the unique first descendant v in W_i. □

Summing up, we have the following result.

PROPOSITION 4.5. CORE-BASE *can be computed in time*

$$O(n) + 2 \max_{\substack{\sum n_i = n, \\ n_i \in \{1, \ldots, \lfloor n/\kappa \rfloor\}}} \sum \mathtt{time}(\textsc{Trees}(n_i)).$$

Proof. Lemma 4.3 allows us to compute all $\{s, t_s\}$ pairs in $O(n)$ time. Clearly, $\sum_s |\mathtt{A}(s)| = n$, and for each s, $|\mathtt{A}(s)| < n/\kappa$. We can compute $\textsc{Trees}\{s, t_s\}$ for all s in time bounded by

$$2 \max_{\substack{\sum n_i = n, \\ n_i \in \{1, \ldots, \lfloor n/\kappa \rfloor\}}} \sum \mathtt{time}(\textsc{Trees}(n_i)).$$

By Lemma 4.2, we are now done with the side values. Concerning the base of the core trees, now preprocess for any queries of the form $\mathtt{mast}\{\mathtt{r}(s), v\}$, where s is a side tree and v is any opposing vertex. For simplicity, we assume that s is a side tree of $\mathcal{T}_0$ and v is a vertex of $\mathcal{T}_1$. The case where s is a side tree of $\mathcal{T}_1$ and v is a vertex of $\mathcal{T}_0$ is symmetric.

By Lemma 4.1, our problem is to decide if v has a descendant in t_s and, if so, to return the first such descendant of v in t_s. Such a query can be answered in time $O(\log |\mathtt{V}(t_s)|) = O(\log n)$ if we first apply the preprocessing of Lemma 4.4 to all the sets $\mathtt{V}(t_s)$. The time for this preprocessing is

$$O\left(|\mathtt{V}(\mathcal{T}_0)| + \sum_s |\mathtt{V}(t_s)|\right) = O\left(n + \sum_s (2|\mathtt{A}(s)| - 1)\right) = O(n).$$

Thus all the preprocessing is completed within the desired bounds. □

4.1. A master theorem. The following theorem will be used to bound the overall work for the bounded-degree and general versions of the MAST problem throughout the remainder of the paper. Our goal is to make our result apply, even if the complexity of maximum-weighted matching is improved. Thus we must allow our recurrence to hold for a wide possible set of choices for the complexity of maximum-weighted matching. While general theorems have been proven for solving recurrences (see, e.g., Verma [23]), we know of no results that directly apply. Thus we offer the following "master theorem" for solving the types of recurrence we need.

THEOREM 4.6. *Assume for some monotone function* $C : \mathbb{R}_{\geq 1} \to \mathbb{R}$ *that* CORE-TREES *can be computed in time at most* $C(\kappa^a n)$*, where* a *is a constant independent of our parameter* κ*. Moreover, assume that* $C(x) = x^{1+\varepsilon} f(x)$*, where* $\varepsilon \geq 0$ *is a constant,* $f(x) = O(x^{o(1)})$*,* f *is monotone, and for some constants* b_1 *and* b_2*,* $\forall x, y \geq b_1 : f(xy) \leq b_2 f(x) f(y)$*. If* $\varepsilon = 0$*, there is a constant* c *such that we can compute* MAST *in time* $O(C(n)c^{\sqrt{\log n}})$*. Otherwise, if* $\varepsilon > 0$*, setting* $\kappa = 4^{\sqrt{\log n}}$*, we can compute* MAST *in time* $O(C(n))$*. Thus* MAST *is computable in time* $O(n^{1+o(1)} + C(n))$*.*

Proof. By Proposition 4.5 and the definition of C, using Algorithm B, we compute TREES in time

$$O(n) + 2 \max_{\substack{\sum n_i = n, \\ n_i \in \{1, \ldots, \lfloor n/\kappa \rfloor\}}} \sum \mathtt{time}(\textsc{Trees}(n_i)) + C(k^a n).$$

Thus for any constant $n_0 \geq 1$ (to be fixed later), we may choose a constant k_1 such that

$$\forall n \in \mathbb{N} : 1 \leq n \leq n_0 \Rightarrow \mathtt{time}(\textsc{Trees}(n)) \leq k_1 C(n), \tag{1}$$

$$\forall n \in \mathbb{N} : n_0 < n \Rightarrow \Big\langle \forall \kappa \in \mathbb{R} : 1 \leq \kappa \leq n \Rightarrow$$

$$\mathtt{time}(\textsc{Trees}(n)) \leq k_1 C(\kappa^a n) + 2 \max_{\substack{\sum n_i = n, \\ n_i \in \{1, \ldots, \lfloor n/\kappa \rfloor\}}} \sum \mathtt{time}(\textsc{Trees}(n_i)) \Big\rangle. \tag{2}$$

Inductively from (1) and (2), it follows that the time complexity for TREES is bounded by any monotone function $T : \mathbb{R}_{\geq 1} \to \mathbb{R}$ that satisfies

$$\forall x \in \mathbb{R} : 1 \leq x \leq n_0 \Rightarrow$$

$$T(x) \geq k_1 C(x), \tag{3}$$

$$\forall x \in \mathbb{R} : n_0 < x \Rightarrow \Big\langle \exists \kappa \in \mathbb{R} : 1 \leq \kappa \leq x \wedge$$

$$T(x) \geq k_1 C(\kappa^a x) + 2 \max_{\substack{\sum x_i = n, \\ 1 \leq x_i \leq n/\kappa}} \sum T(n_i) \Big\rangle. \tag{4}$$

In our search for such an adequate function T, we will restrict ourselves to superlinear functions that satisfies

$$\exists \text{monotone } g : \mathbb{R}_{\geq 1} \to \mathbb{R} \; \forall x \in \mathbb{R}_{\geq 1} : T(x) = x g(x). \tag{5}$$

As a convenient consequence, for any $x, x_1, \ldots, x_l \in \mathbb{R}_{\geq 1}$ such that $\sum x_i = x$ and $1 \leq x_i \leq x/\kappa$, we have that $\sum T(x_i) = \sum (x_i g(x_i)) \leq \sum (x_i g(x/\kappa)) = x g(x/\kappa) =$

$\kappa T(x/\kappa)$. Thus (4) follows if

$$(6)\quad \forall x \in \mathbb{R} : x > n_0 \Rightarrow \langle \exists \kappa \in \mathbb{R} : 1 \le \kappa \le x \ \wedge\ T(x) \ge k_1 C(\kappa^a x) + 2\kappa T(x/\kappa)\rangle.$$

The rest of the proof divides into cases depending on ε.

$\varepsilon = 0$. For this case, we let n_0 be the least number such that $n_0 \ge b_1$, $4^a\sqrt{\log n_0} \ge b_1$, and $4\sqrt{\log n_0} \le n_0$. The last inequality is satisfied if and only if $n_0 \ge 16$. Recall that our choice of n_0 affects k_1. Since $\varepsilon = 0$, $C(x) = O(x^{1+o(1)})$, so we may choose a constant $k_2 \ge (2b_2)^{-1}$ such that $\forall x \in \mathbb{R}_{\ge 1} : C(x) \le k_2 x^{1.5}$. For a solution to our problem, we define T such that

$$(7)\qquad \forall x \in \mathbb{R}_{\ge 1} : T(x) = 2b_2 k_1 k_2 c^{\sqrt{\log x}} C(x), \quad \text{where } c = \max\{8^a, 4\}.$$

Clearly, (3) is satisfied since

$$\forall x \in \mathbb{R}_{\ge 1} : T(x) = 2b_2 k_1 k_2 c^{\sqrt{\log x}} C(x) \ge 2b_2 k_1 (2b_2)^{-1} C(x) = k_1 C(x).$$

Also, (5) is satisfied because for the function that maps x to

$$T(x)/x = 2b_2 k_1 k_2 c^{\sqrt{\log x}} C(x)/x = 2b_2 k_1 k_2 c^{\sqrt{\log x}} x f(x)/x = 2b_2 k_1 k_2 c^{\sqrt{\log x}} f(x)$$

is monotone since f is monotone. Hence (4) follows if we can settle (6). Fix $x > n_0$ and set $\kappa = 4^{\sqrt{\log x}}$. Notice that $1 < \kappa < x$, $\kappa^a > b_1$, and $x > b_1$. Now

$$\begin{aligned} k_1 C(\kappa^a x) &\le k_1 b_2 C(\kappa^a) C(x) \le b_2 k_1 k_2 (\kappa^a)^{1.5} C(x) = b_2 k_1 k_2 4^{1.5a\sqrt{\log x}} C(x) \\ &\le b_2 k_1 k_2 c^{\sqrt{\log x}} C(x) \le T(x)/2. \end{aligned}$$

Moreover,

$$\sqrt{\log(x/\kappa)} = \sqrt{\log(x/4^{\sqrt{\log x}})} = \sqrt{\log x - 2\sqrt{\log x}} \le \sqrt{\log x} - 1,$$

and hence

$$\begin{aligned} \kappa T(x/\kappa) &\le \kappa 2b_2 k_1 k_2 c^{\sqrt{\log(x/\kappa)}} C(x/\kappa) \le \kappa 2b_2 k_1 k_2 c^{\sqrt{\log x}-1}(x/\kappa) f(x/\kappa) \\ &\le 2b_2 k_1 k_2 (c^{\sqrt{\log x}}/c) x f(x) \le 2b_2 k_1 k_2 c^{\sqrt{\log x}} C(x)/c \le T(x)/4. \end{aligned}$$

Thus

$$k_1 C(\kappa^a x) + 2\kappa T(x/\kappa) \le T(x)/2 + 2T(x)/4 = T(x),$$

so (6) and hence (4) is satisfied. We may therefore conclude that with $\varepsilon = 0$, we can compute TREES on a problem of size n in time $2b_2 k_1 k_2 c^{\sqrt{\log n}} C(n) = O(C(n) c^{\sqrt{\log n}}) = O(n^{1+o(1)})$, as desired.

$\varepsilon > 0$. In this case, we fix $\kappa = \max\{4^{1/\varepsilon}, \sqrt[a]{b_1}\}$ and set $n_0 = \kappa b_1$. For a solution to our problem, we define T such that

$$(8)\qquad \forall x \in \mathbb{R}_{\ge 1} : T(x) = k_1 k_3 C(x) \quad \text{where } k_3 = \max\{1, 2b_2 C(\kappa^a)\}.$$

Clearly, (3) and (5) are satisfied. Fix $x > n_0$. Now

$$k_1 C(\kappa^a x) \le b_2 C(\kappa^a) k_1 C(x) \le T(x)/2$$

and

$$\kappa T(n/\kappa) = \kappa b_2 k_3 (n/\kappa)^{1+\varepsilon} f(x/\kappa) \leq b_2 k_3 n^{1+\varepsilon} \kappa^{-\varepsilon} f(x) \leq b_2 k_3 n^{1+\varepsilon} f(x)/4 = T(x)/4,$$

so

$$k_1 C(\kappa^a x) + 2\kappa T(x/\kappa) \leq T(x)/2 + 2T(x)/4 = T(x),$$

Thus (6) and hence (4) is satisfied. We may therefore conclude that with $\varepsilon > 0$, we can compute TREES on a problem of size n in time $b_2 k_1 k_3 C(n) = O(C(n))$, completing the proof. □

5. Computing SPINES. To implement MAST for bounded-degree trees, we need only show how to quickly compute SPINES (Procedure 3). To this end, we introduce the concept of intervals. For spine S with c the critical child of the lowest vertex, let $v, w \in \mathtt{V}(S) \cup \{c\}$, where w is an ancestor of v. Then v and w characterize an *interval* I of S, denoted by $]v, w]$. If $I =]v, w]$, we set $\mathtt{c}(I) = v$ and $\mathtt{r}(I) = w$. By $\mathtt{V}(I)$ we denote the set of vertices that are descendants of w and strict ancestors of v. If $v \neq w$, $\mathtt{V}(I)$ induces a segment of S, and we will often identify I with this segment. If $v = w$, the interval I corresponds to the empty segment together with a position. For any interval I, we define $\mathtt{SideT}(I)$ to be the forest of side trees whose roots are children of vertices in I. Let I_0 and I_1 be intervals from opposing core trees. We say that the pair of opposing intervals $\{I_0, I_1\}$ is *interesting* if and only if $\mathtt{A}(\mathtt{SideT}(I_0)) \cap \mathtt{A}(\mathtt{SideT}(I_1)) \neq \emptyset$. A pair which is not interesting is said to be *boring*. We will show how to implement SPINES quickly by giving an efficient method for dealing with boring interval pairs.

For specificity, fix $\{S_0, S_1\}$ to be a pair of opposing spines for which we wish to compute SPINES$\{S_0, S_1\}$. To ease the presentation, we will identify S_0 and S_1 with the intervals with the same vertex sets. Moreover, we set $m_0 = |\mathtt{V}(S_0)|$, $m_1 = |\mathtt{V}(S_1)|$, and $l = |\mathtt{A}(\mathtt{SideT}(S_0)) \cap \mathtt{A}(\mathtt{SideT}(S_1))|$. Notice that the number of maximal boring interval pairs can be as bad as $\Omega(l^2)$. However, we are going to identify $O(m_0 + m_1 + l \log n)$ boring interval pairs together with $O(m_0 + m_1 + l)$ vertex pairs, representing all computations needed to implement SPINES$\{S_0, S_1\}$.

Our algorithm for SPINES will proceed as follows. In section 5.1, we will choose a small subset of interval pairs on which to compute certain values. We will organize these intervals into an *interval tree*. In section 5.2, we will prove the main technical lemmas needed for dealing with boring interval pairs. In section 5.3, we will show how to actually implement SPINES in terms of the interval tree.

5.1. The interval tree. Consider an interval $I =]v, w]$, and let u be the vertex such that $|\mathtt{V}(]u, w])| = \lceil |\mathtt{V}(I)|/2 \rceil$. Then I^l denotes $]v, u]$ and I^r denotes $]u, w]$. Note that $\mathtt{V}(I^l)$ and $\mathtt{V}(I^r)$ are disjoint. Also note that if $I =]v, v]$ then $I^l = I^r = I$.

Given an opposing pair $\{I_0, I_1\}$ of intervals, we call $\mathtt{mast}\{\mathtt{c}(I_0), \mathtt{c}(I_1)\}$ the *floor*, $\mathtt{mast}\{\mathtt{c}(I_0), \mathtt{r}(I_1)\}$ and $\mathtt{mast}\{\mathtt{r}(I_0), \mathtt{c}(I_1)\}$ the *diagonals*, and $\mathtt{mast}\{\mathtt{r}(I_0), \mathtt{r}(I_1)\}$ the *ceiling* of the pair. Thus if I_0 and I_1 are intervals of spines S_0 and S_1, then the diagonals, together with the floor, are exactly the values from the base of $\{\mathtt{V}(I_0), \mathtt{V}(I_1)\}$ that are not in the base of $\{\mathtt{V}(S_0), \mathtt{V}(S_1)\}$.

We are now going to construct a rooted tree $\mathcal{E}$ whose nodes are opposing interval pairs. The root pair is (S_0, S_1). Suppose (I_0, I_1) is a node in $\mathcal{E}$ and that one of the following conditions is satisfied:

(i) $\{I_0, I_1\}$ is interesting and one of I_0 and I_1 contains more than one vertex.

(ii) One of I_0 and I_1 has the spine root as root, and the other contains more than one vertex.

Then $\{I_0, I_1\}$ has four children: (I_0^l, I_1^l), (I_0^l, I_1^r), (I_0^r, I_1^l), and (I_0^r, I_1^r); otherwise, $\{I_0, I_1\}$ is a leaf.

We are going to compute the ceiling and the diagonals of all the pairs in $\mathcal{E}$. Clearly, we will thereby implement SPINES since if r is the root of one of the spines and v is a vertex from the other spine, then the second condition defining $\mathcal{E}$ ensures that $\mathtt{mast}\{r, v\}$ is the ceiling of some pair in $\mathcal{E}$.

PROPOSITION 5.1. *The size and construction time for $\mathcal{E}$ is $O(m_0+m_1+l\log n)$.*

Proof. In our argument, we will focus on the size of $\mathcal{E}$. However, a corresponding construction will be indicated on the side. The depth of $\mathcal{E}$ is no more than $\max\{\lceil\log_2|\mathtt{V}(S_0)|\rceil, \lceil\log_2|\mathtt{V}(S_1)|\rceil\}$. Clearly, we only need to bound the number of internal nodes since the number of leaves is less than a factor of four larger. We will separately count the number of nodes made internal by the two conditions. Thereby, we accept a certain overlap where a node $\{I_0, I_1\}$ satisfies both conditions.

Concerning condition (i), by induction starting at the root, for each $a \in \mathtt{A}(\mathtt{SideT}(S_0)) \cap \mathtt{A}(\mathtt{SideT}(S_1)$, there is exactly one node $\{I_0, I_1\}$ at each level in $\mathcal{E}$ such that $a \in \mathtt{A}(\mathtt{SideT}(I_0)) \cap \mathtt{A}(\mathtt{SideT}(I_1)$. Thus at each level, we have at most $l = |\mathtt{A}(\mathtt{SideT}(S_0))\cap\mathtt{A}(\mathtt{SideT}(S_1)|$ satisfying condition (i), so in total we have at most $l\log n$ nodes in $\mathcal{E}$ satisfying (i). In order to find these nodes, starting from the root, with each node $\{I_0, I_1\}$ we store the set $\mathtt{A}(\mathtt{SideT}(I_0))\cap\mathtt{A}(\mathtt{SideT}(I_1))$—$\{I_0, I_1\}$ is interesting if the set is nonempty. If condition (i) is satisfied, in time $O(|\mathtt{A}(\mathtt{SideT}(I_0)) \cap \mathtt{A}(\mathtt{SideT}(I_1)|)$, we partition $\mathtt{A}(\mathtt{SideT}(I_0))\cap\mathtt{A}(\mathtt{SideT}(I_1)$ into $\mathtt{A}(\mathtt{SideT}(I_0^l))\cap\mathtt{A}(\mathtt{SideT}(I_1^l)$, $\mathtt{A}(\mathtt{SideT}(I_0^r)\cap \mathtt{A}(\mathtt{SideT}(I_1^l)$, and $\mathtt{A}(\mathtt{SideT}(I_0^r))\cap\mathtt{A}(\mathtt{SideT}(I_1^r)$ for the children. For each of the $O(\log n)$ levels, the total size of these sets is no more than l, so the construction time is $O(l\log n)$.

Concerning the internal nodes satisfying (ii), ignoring the symmetric case, we restrict (ii) to

(ii)$_0$ $\mathtt{r}(I_0) = \mathtt{r}(S_0)$ and I_1 contains more than one vertex.

Consider a node (I_0, I_1) of $\mathcal{E}$ satisfying (ii)$_0$. Then due to the first part of the condition, the only children of (I_0, I_1) that can satisfy (ii)$_0$ are (I_0^r, I_1^l) and (I_0^r, I_1^r). Since there is no freedom in choosing first coordinate of the children, the number of internal nodes in $\mathcal{E}$ satisfying (ii)$_0$ is

$$\begin{aligned}|\{S_1^\alpha : \alpha \in \{l,r\}^*, |\mathtt{V}(S_1^\alpha)| > 1\}| &< |\{S_1^\alpha : \alpha \in \{l,r\}^*, |\mathtt{V}(S_1^\alpha)| = 1\}| \\ &= |\mathtt{V}(S_1)| \;\le\; |\mathtt{A}(\mathtt{SideT}(S_1))| \;=\; m_1.\end{aligned}$$

The first inequality follows from the fact that the number of internal nodes of a binary tree is smaller than the number of leaves.

Equivalently, for the symmetric case (ii)$_1$, where $\mathtt{r}(I_1) = \mathtt{r}(S_1)$ and I_0 contains more than one vertex, we get at most m_0 internal nodes in $\mathcal{E}$. Thus there are at most m_0+m_1 internal nodes in $\mathcal{E}$ satisfying (ii) [$=$ (ii)$_0 \vee$ (ii)$_1$], so we conclude that in total there are at most $O(m_0+m_1+l\log n)$ nodes in $\mathcal{E}$. Moreover, $\mathcal{E}$ can be generated from the root (S_0, S_1), adding each new node in constant time. □

We are going to compute the ceiling and the diagonals of all the pairs in $\mathcal{E}$ respecting the linear order $<$ such that for each internal pair (I_0, I_1) of $\mathcal{E}$, we have $(I_0^l, I_1^l) < (I_0^l, I_1^r) < (I_0^r, I_1^l) < (I_0^r, I_1^r) < (I_0, I_1)$. Thus whenever we compute a pair, we can assume that all previous pairs have been computed.

OBSERVATION 5.2. *For any internal pair of $\mathcal{E}$, the ceiling and diagonals are the ceiling and diagonals of its children. The floor of any pair in $\mathcal{E}$ is the ceiling or diagonal of some preceding pair.* □

Thus we need only concern ourselves with computations at the leaves, which are either boring interval pairs or interesting node pairs.

5.2. Handling boring interval pairs.

LEMMA 5.3. *If the interval pair* $\{I_0, I_1\}$ *is boring, then*

$$\texttt{mast}\{\texttt{r}(I_0), \texttt{r}(I_1)\} = \max\left\{ \begin{array}{l} \texttt{mast}\{\texttt{c}(I_0), \texttt{r}(I_1)\}, \texttt{mast}\{\texttt{c}(I_1), \texttt{r}(I_0)\}, \\ \texttt{side-mast}\{\texttt{c}(I_0), I_1\} + \texttt{side-mast}\{\texttt{c}(I_1), I_0\} \end{array} \right\}.$$

Here $\texttt{side-mast}\{v, I\} = \max\{\texttt{mast}\{v, \texttt{r}(t)\} | t \in \texttt{SideT}(I)\}$.

Proof. Let lhs and rhs denote the left- and right-hand side of the equality of the lemma. First we prove lhs $\geq$ rhs. Clearly, $\texttt{mast}\{\texttt{r}(I_0), \texttt{r}(I_1)\} \geq \max\{\texttt{mast}\{\texttt{c}(I_0), \texttt{r}(I_1\}, \texttt{mast}\{\texttt{c}(I_1), \texttt{r}(I_0)\}\}$. For $i = 0, 1$, let $t_{\bar{i}}$ be a side tree in $\texttt{SideT}(I_{\bar{i}})$ such that $\texttt{mast}\{\texttt{c}(I_i), \texttt{r}(t_{\bar{i}})\} = \texttt{side-mast}\{\texttt{c}(I_i), I_{\bar{i}}\}$. Let p_i and c_i denote the parent and core sibling of $\texttt{r}(t_{\bar{i}})$. By application of Lemma 3.1, we get the following inequalities:

$$\begin{aligned} \texttt{mast}\{c_i, \texttt{r}(t_{\bar{i}})\} &\geq \texttt{mast}\{\texttt{c}(I_i), \texttt{r}(t_{\bar{i}})\} \text{ for } i = 0, 1, \\ \texttt{mast}\{f_0, f_1\} &\geq \texttt{mast}\{c_0, \texttt{r}(t_1)\} + \texttt{mast}\{\texttt{r}(t_0), c_1\}, \\ \texttt{mast}\{\texttt{r}(I_0), \texttt{r}(I_1)\} &\geq \texttt{mast}\{f_0, f_1\}. \end{aligned}$$

Hence it follows that $\texttt{mast}\{\texttt{r}(I_0), \texttt{r}(I_1\} \geq \texttt{side-mast}\{\texttt{c}(I_0), I_1\} + \texttt{side-mast}\{\texttt{c}(I_1), I_0\}$, and we may therefore conclude that lhs $\geq$ rhs.

We show that lhs $\leq$ rhs by contradiction. Assume that there is a pair $(f_0, f_1) \in \texttt{V}(I_0) \times \texttt{V}(I_1)$ such that $\texttt{mast}\{f_0, f_1\} >$ rhs and fix (f_0, f_1) to be a minimal such pair. First, we observe that the strict inequality implies that $s_0 \neq \texttt{c}(I_0)$ and $s_1 \neq \texttt{c}(I_1)$. For $i = 0, 1$, let c_i denote the core child of f_i. By the minimality of (f_0, f_1), we cannot have $\texttt{mast}\{f_i, f_{\bar{i}}\} = \texttt{mast}\{c_i, f_{\bar{i}}\}$. Also by minimality, we cannot have $\texttt{mast}\{f_i, f_{\bar{i}}\} = \texttt{mast}\{s_i, f_{\bar{i}}\}$, where s_i is a side child of f_i, because since our pair of intervals is boring, $\texttt{mast}\{s_i, f_{\bar{i}}\} = \texttt{mast}\{s_i, c_{\bar{i}}\} \leq \texttt{mast}\{f_i, c_{\bar{i}}\}$. Thus by Lemma 3.1, we have $\texttt{mast}\{f_0, f_1\} = \texttt{match}\{f_0, f_1\}$. Let M be a minimal matching in $\texttt{C}(f_0) \times \texttt{C}(f_1)$ such that $\texttt{match}\{f_0, f_1\} = \sum_{(v_0, v_1) \in M} \texttt{mast}\{v_0, v_1\}$. Since our interval pair is boring, we can only have an edge in M if either its head or its tail is core. By the minimality of (f_0, f_1), we cannot have $M = \{(c_0, c_1)\}$. Thus we have $M \subseteq \{(c_0, s_1), (c_1, s_0)\}$, where s_i is a specific side child of f_i. Putting everything together, we get

$$\begin{aligned} \text{rhs} &< \texttt{match}\{f_0, f_1\} \\ &= \sum_{(v_0, v_1) \in M} \texttt{mast}\{v_0, v_1\} \\ &= \texttt{mast}\{c_0, s_1\} + \texttt{mast}\{c_1, s_0\} \\ &= \texttt{mast}\{\texttt{c}(I_0), s_1\} + \texttt{mast}\{\texttt{c}(I_1), s_0\} \\ &\leq \texttt{side-mast}\{\texttt{c}(I_0), I_1\} + \texttt{side-mast}\{\texttt{c}(I_1), s_0\} \\ &\leq \text{rhs}, \end{aligned}$$

and hence we have the desired contradiction. □

What makes this useful is the following technical lemma.

LEMMA 5.4. *After an* $O(n \log n)$ *preprocessing based on the base of the core trees, given any pair of a vertex* v *from one core tree and an interval* I *from the other core tree, we can compute* $\texttt{side-mast}\{v, I\}$ *in time* $O(\log^2 n)$.

Proof. For simplicity, we assume that v is from $\mathcal{T}_0$ and I is from $\mathcal{T}_1$. A symmetric preprocessing is needed for the opposite case where I is from $\mathcal{T}_0$ and v is from $\mathcal{T}_1$. First, for each spine S in $\mathcal{T}_1$, we construct a balanced binary tree $\mathcal{I}_S$ over some nonempty intervals of S. The root of $\mathcal{I}_S$ is S itself, and given any node I in $\mathcal{I}_S$, I is a leaf if it consists of a single vertex; otherwise, I has children I^l and I^r. $\mathcal{I}_S$ has depth

$\lceil \log_2 |\mathtt{V}(S)| \rceil$. Denote by $\mathcal{I}$ the forest of the $\mathcal{I}_S$'s. Now the spines of $\mathcal{T}$ constitute the top level of $\mathcal{I}$. Generally, we have that all intervals at any specific level are mutually disjoint.

Our goal is to find an $O(n \log n)$ preprocessing such that given any vertex from $\mathcal{T}_0$ and interval I from $\mathcal{I}$, we can derive $\mathtt{side\text{-}mast}(v, I)$ in time $O(\log n)$. Assume that this is done. Then any interval I from $\mathcal{T}_1$ is the concatenation $I_1 \cdots I_l$ of at most $2 \log n$ intervals from $\mathcal{I}$, and then $\mathtt{side\text{-}mast}(v, I) = \max_i\{\mathtt{side\text{-}mast}(v, I_i)\}$. Thus such a preprocessing allows us to compute $\mathtt{side\text{-}mast}$ in time $O(\log^2 n)$, as desired.

For $i = 0, 1$ and for every label a, let $\mathtt{sr}_i(a)$ denote the root of the side tree of $\mathcal{T}_i$ containing the leaf with label a. By contraction of the side trees, we precompute all values of $\mathtt{sr}$ in time $O(n)$.

The remaining preprocessing is divided into $O(\log n)$ separate $O(n)$ preprocessings: one for each level in $\mathcal{I}$. Let $I_1, \ldots, I_k$ be all the intervals at some specific level in $\mathcal{I}$. Then $I_1, \ldots, I_k$ are mutually disjoint, so, in particular, $\mathtt{A}(\mathtt{SideT}(I_1)), \ldots, \mathtt{A}(\mathtt{SideT}(I_k))$ are mutually disjoint.

Consider an arbitrary fixed interval I of $\mathcal{T}_0$, and note the following recursion formula for $\mathtt{side\text{-}mast}(\cdot, I)$:

$$\mathtt{side\text{-}mast}(v, I) = \max\{$$

$$\max\{\mathtt{mast}(v, \mathtt{r}(t)) | t \in \mathtt{SideT}(I)\}, \tag{9}$$

$$\max\{\mathtt{side\text{-}mast}(w, I) | w \text{ is a descendant } v\}\} \tag{10}$$

Set $V_I = \{\mathtt{p}(\mathtt{sr}_0(a)) | a \in \mathtt{A}(\mathtt{SideT}(I))\}$ and $W_I = \mathtt{LCA}(V_I)$. Then V_I contains all core vertices v for which (9) is relevant. Moreover, W_I contains all the values of w that are relevant for (10). Let t_I denote the topological subtree of $\mathcal{T}_0$ with vertex W_I. All of the sets V_{I_i} are easily computed in time $O(n)$, and by Lemma 4.3, in time $O(n)$ we can compute all the t_{I_i}'s.

Consider any specific t_{I_i}. We want to compute $\mathtt{side\text{-}mast}(v, I_i)$ for all $v \in \mathtt{V}(t_{I_i}) = W_{I_i}$. For this purpose, we introduce an variable $sm(v)$ for each $v \in W_{I_i}$. Initially $sm(v) := 0$ for all $v \in W_{I_i}$. Now, corresponding to (9), for all $a \in \mathtt{A}(\mathtt{SideT}(I_i))$, set

$$sm(\mathtt{p}(\mathtt{sr}_0(a))) := \max\{sm(\mathtt{p}(\mathtt{sr}_0(a))), \mathtt{mast}\{\mathtt{p}(\mathtt{sr}_0(a))), \mathtt{sr}_1(a)\}.$$

Next, corresponding to (10), go through the vertices of t_{I_i} in post-order. When visiting the vertex v, set

$$v := \max\{sm(v), \max\{sm(w) | w \text{ is a child of } v \text{ in } t_{I_i}\}\}.$$

When the computation is finished $sm(v) = \mathtt{side\text{-}mast}(v, I_i)$ for all $v \in \mathtt{V}(t_{I_i}) = W_{I_i}$. The time of the computation is $O(|\mathtt{A}(\mathtt{SideT}(I_i))| + |W_{I_i}|) = O(|\mathtt{A}(\mathtt{SideT}(I_i))|)$. Thus we can precompute $\mathtt{side\text{-}mast}(v, I_i)$ for all $I_i, w \in W_{I_i}$ in time $O(\sum |\mathtt{A}(\mathtt{SideT}(I_i))|) = O(n)$.

Now consider any query $\mathtt{side\text{-}mast}(v, I_i)$ where $v \notin W_{I_i}$. Then $\mathtt{side\text{-}mast}(v, I) = \mathtt{side\text{-}mast}(w, I)$, where w is the nearest descendant of v in W_{I_i}, if any; otherwise, $\mathtt{side\text{-}mast}(v, I) = 0$. We solve the nearest-descendant query in $O(\log n)$ time by first applying the preprocessing from Lemma 4.4 to all the W_{I_i}s. This preprocessing takes time $O(\sum_{I_i} |W_{I_i}|)$, where

$$\sum_{I_i} |W_{I_i}| < 2 \sum_{I_i} |V_{I_i}| \leq 2 \sum_{I_i} |\mathtt{A}(\mathtt{SideT}(I_i))| = 2n.$$

Thus all preprocessing of $I_1, \ldots, I_k$ is done in time $O(n)$, so the total preprocessing is done in time $O(n \log n)$, as desired. □

5.3. Using the interval tree.

PROPOSITION 5.5. *Any diagonal of a pair in $\mathcal{E}$ can be computed in time $O(\log^2 n)$.*

Proof. Let (I_0, I_1) be any pair in $\mathcal{E}$. By symmetry, it is sufficient to show the computation of $\mathtt{mast}\{\mathtt{c}(I_0), \mathtt{r}(I_1)\}$. Suppose there is a vertex $v_0 \in \mathtt{V}(S_0)$ below or equal to $\mathtt{c}(I_0)$ which is interesting with respect to some vertex in I_1. Fix v_0' to be the highest such vertex, i.e., the one closest to $\mathtt{c}(I_0)$. Let I_0' be the interval in $\mathcal{I}_0$ which contains v_0 and which is on the same level as I_0 and hence as I_1 in $\mathcal{I}_1$. Thus I_0' is interesting with respect to I_1, and since they are on the same level in their respective interval trees, we can conclude that $(I_0', I_1) \in \mathcal{E}$. Trivially, $(I_0', I_1) < (I_0, I_1)$, so we can assume that the ceiling $\mathtt{mast}\{\mathtt{r}(I_0'), \mathtt{r}(I_1)\}$ is computed.

Set $I_0'' =]\mathtt{r}(I_0'), \mathtt{c}(I_0)]$. By choice of v_0', the pair (I_0'', I_1) is boring—but typically not in $\mathcal{E}$. The diagonal $\mathtt{mast}\{\mathtt{c}(I_0''), \mathtt{r}(I_1)\}$ is exactly the ceiling of (I_0', I_1) which we saw was computed, and the diagonal $\mathtt{mast}\{\mathtt{r}(I_0'), \mathtt{c}(I_1)\}$ is the floor of (I_0, I_1) which is computed by Lemma 5.2. Thus by Lemmas 5.3 and 5.4, we can compute the ceiling $\mathtt{mast}\{\mathtt{r}(I_0''), \mathtt{r}(I_1)\}$ in time $O(\log^2 n)$. However, this ceiling is exactly the desired diagonal of (I_0, I_1). □

COROLLARY 5.6. *Any boring pair in $\mathcal{E}$ can be computed in time $O(\log^2 n)$.*

Proof. By Proposition 5.5, we can compute the diagonals in time $O(\log^2 n)$, but then we can apply Lemmas 5.3 and 5.4 to get the ceiling in time $O(\log^2 n)$. □

For any opposing pair $\{v_0, v_1\}$, by $\mathtt{time}(\mathtt{mast}\{v_0, v_1\})$ we refer to the time it takes to compute $\mathtt{mast}\{v_0, v_1\}$ assuming that every value in the base of $\{v_0, v_1\}$ is available in time $O(\log n)$.

COROLLARY 5.7. *Any interesting leaf pair (I_0, I_1) in $\mathcal{E}$ can be computed in time $O(\log^2 n + \mathtt{time}(\mathtt{mast}\{\mathtt{r}(I_0), \mathtt{r}(I_1)\})$.*

Proof. Again, the diagonals are computed by Proposition 5.5, and the floor follows by Lemma 5.2. Hence we have the whole basis of $\{\mathtt{V}(I_0), \mathtt{V}(I_1)\}$ available. Since (I_0, I_1) is an interesting leaf pair, it follows that $\mathtt{V}(I_i) = \{\mathtt{r}(I_i)\}$ for $i = 0, 1$. Thus, in fact, we have the basis of $\{\mathtt{r}(I_0), \mathtt{r}(I_1)\}$ available, so the ceiling $\mathtt{mast}\{\mathtt{r}(I_0), \mathtt{r}(I_1)\}$ can be computed directly. □

Summing up, we conclude as following.

PROPOSITION 5.8. SPINES$\{S_0, S_1\}$ (*Procedure* 3) *can be computed in time $O((m_0 + m_1 + l \log n) \log^2 n + (\sum_{\{v_0, v_1\} \in F} \mathtt{time}(\mathtt{mast}\{v_0, v_1\}))$, where $m_0 = |\mathtt{V}(S_0)|$, $m_1 = |\mathtt{V}(S_1)|$, $l = |\mathtt{A}(\mathtt{SideT}(S_0)) \cap \mathtt{A}(\mathtt{SideT}(S_1))|$, and $F \subseteq \mathtt{V}(S_0) \times \mathtt{V}(S_1)$ contains at most l pairs.*

Proof. We observed above that, indeed, computing all the ceilings of pairs in $\mathcal{E}$ is sufficient for implementing SPINES; and now, from Proposition 5.1, Lemma 5.2, and Corollaries 5.6 and 5.7 together with the observation that there are at most l interesting leaf pairs, it follows that we can compute all ceilings and diagonals of the pairs in $\mathcal{E}$ within the desired time bound. □

THEOREM 5.9. CORE-TREES (*Procedure* 2) *can be computed in time $O(\kappa n \log^3 n + \sum_{\{v_0, v_1\} \in E} \mathtt{time}(\mathtt{mast}\{v_0, v_1\}))$. Here E divides into two sets E_1 and E_2. The set E_1 contains all the $O(\kappa n)$ pairs of core vertices where one is critical. The set E_2 contains the at most n interesting pairs of spine vertices.*

Proof. Let S denote the set of pairs of spines from opposing core trees. Since there can be at most κ spines in each core tree, we get $(\sum_{\{S_0, S_1\} \in S} |\mathtt{V}(S_0)| + |\mathtt{V}(S_1)|) \leq \kappa n$. Moreover, by Fact 3.2, we get that $(\sum_{\{S_0, S_1\} \in S} |\mathtt{A}(\mathtt{SideT}(S_0)) \cap \mathtt{A}(\mathtt{SideT}(S_1))|) \leq n$. Thus the result follows directly from Observation 3.3 together with Proposition 5.8. □

COROLLARY 5.10. *For trees T_0 and T_1 with bounded degree,* TREES$\{T_0, T_1\}$ *can be computed in time $O(nc^{\sqrt{\log n}})$.*

Proof. For bounded degrees, $\texttt{time}(\texttt{mast}\{v_0, v_1\})) = O(\log n)$, so the result follows from Theorem 4.6 with $\kappa = 4^{\sqrt{\log n}}$ and Theorem 5.9. □

6. Computing the matchings. The aim of this section is to reduce the work of computing the $O(\kappa n)$ mast values of the set E specified in Theorem 5.9 when the trees given have unbounded degree. Recall that $\texttt{mast}\{v, w\}$ is found as the maximum value over $\texttt{Diag}\{v, w\}$ and $\texttt{match}\{v, w\}$. The maximum diagonal of the $O(\kappa n)$ pairs in E can easily be computed in $O(\kappa n \log n)$ time by a bottom-up dynamic program, so our problem is to bound the work on matchings. Recall that the matching associated with a pair $\{v, w\}$ is on the weighted bipartite graph whose vertex sets are $\texttt{C}(v)$ and $\texttt{C}(w)$ and whose edges (u, v) are weighted by $\texttt{mast}\{u, v\}$. We denote this graph by G_{vw}. For any weighted bipartite graph G, let $\texttt{match}(G)$ be the value of the maximal weighted bipartite matching on G. Thus $\texttt{match}\{v, w\} = \texttt{match}(G_{vw})$ for all opposing vertex pairs $\{v, w\}$. We will reduce the size of the matchings by reducing the number of nodes and edges and the total sum of the edge weights involved. Initially, these values are $O(n^2)$, $O(n^2)$, and $O(n^3)$, respectively. Our goal is to reduce them to $O(\kappa n)$, $O(\kappa n)$, and $O(\kappa^2 n)$. We will do this by deleting some edges and reducing the weights of others. In general, we will be building a set of matching graphs H_{vw} from the original matching graphs G_{vw} such that the maximum weight of a matching in G_{vw} can be deduced from the maximum weight of a matching in H_{vw}. Note that we will not explicitly build the G_{vw} since their total size can be as large as $\Omega(n^2)$.

The vertices will be bounded by the number of edges. Recall that nodes come in three types: side, critical, and spine. All edges in the matching are between children of core nodes. Let a *side–side* edge be a nonzero edge between opposing side children. For matching graph G_{vw}, we will let l_{vw} be the number of side–side edges in the graph. By Fact 3.2, there can be no more than n side–side edges, and the sum of their weights is also bounded by n.

Recall that the opposing pairs in E from Theorem 5.9 come in two varieties: E_1 contains all the pairs of core vertices where one is critical; E_2 contains the at most n interesting pairs of spine vertices. We will bound the size of the E_1 and E_2 matchings separately.

LEMMA 6.1. *There are graphs H_{vw} for all $\{v, w\} \in E_1$ such that $\texttt{match}(H_{vw}) = \texttt{match}(G_{vw})$ and such that $\sum_{\{v,w\}\in E_1} |\texttt{E}(H_{vw})| = O(\kappa n)$.*

Proof. As note above, there are at most n side–side edges. All other edges are incident on a core child of a critical node. There are a total of $O(\kappa)$ such core children, and no node can be involved in more than $O(n)$ matching edges. Thus just by consideration of nonzero edges, we see that in total we need only $O(\kappa n)$ edges for the H_{vw}. □

LEMMA 6.2. *There are graphs H_{vw} for all $\{v, w\} \in E_2$ such that $\texttt{match}(H_{vw}) = \texttt{match}(G_{vw})$, $|\texttt{E}(H_{vw})| \le 3l_{vw} + 3$ and $\sum_{\{v,w\}\in E_2} |\texttt{E}(H_{vw})| = O(n)$.*

Proof. First of all, our reduced matching graph H_{vw} contains all side–side edges and the edge between the two opposing core nodes. This gives at most $l_{vw} + 1$ edges. The question is which edges we need to include between the two core nodes and their opposing side nodes.

Let c be one of the core nodes. From c to the opposing side nodes, we will include all of the at most l_{vw} edges to side nodes incident with side–side edges. Moreover, we will include one of the maximum-weight remaining edges to side nodes. Let this edge be $\{c, c^s\}$. Thus H_{vw} contains a total of at most $l_{vw} + 1 + 2(l_{vw} + 1) = 3l_{vw} + 3$ edges, as required.

We need to prove that that $\texttt{match}(H_{vw}) = \texttt{match}(G_{vw})$. Consider an arbitrary maximal matching M in G_{vw}. We assume that M has no zero-weight edges. Suppose

that M contains an edge outside H_{vw}. Then this edge must be between a core node c and an opposing side node u which is not incident on a side–side edge and which is different from c^s. Then c^s cannot be matched in M, so we get a new matching M' in G_{vw} if we replace $\{c, u\}$ by the edge $\{c, c^s\}$ from H_{vw}. Moreover, from our choice of c^s, it follows that the weight of M' is at least that of M. We may therefore conclude that one of the maximal matchings in G_{vw} is a maximal matching in H_{vw}.

Finally, we must bound the summation $\sum_{\{v,w\}\in E_2} 3l_{vw} + 3$. However, $|E_2| \leq n$ and $\sum l_{vw} \leq n$ by Fact 3.2. □

LEMMA 6.3. *We can compute all of the H_{vw}'s in $O(n \log^3 n)$ time.*

Proof. The matching graphs H_{vw} from Lemma 6.1 come automatically from only considering nonzero-weight matching edges. In order to sparsify matching graphs of Lemma 6.2, we first choose an arbitrary ordering of the side children of each vertex. It is now meaningful to talk about intervals of side children. With the same technique that was used in the proof of Lemma 5.4, we can make an $O(n \log n)$ preprocessing such that given any vertex v and interval I of side children of some opposing vertex, we can compute $\max\{\texttt{mast}\{v, w\} | w \in \texttt{V}(I)\}$ in time $O(\log^2 n)$.

Recall our problem: we are given a vertex v together with a vertex w and a subset S of the side children which already participate in the matching since they share labels with the side trees of w. Let w^c be the core children of w. We want to find the side child v^s of v outside S which has the MAST with w^c.

This is done in time $O(|S| \log^2 n)$ because removing the vertices from S leaves us with no more than $|S| + 1$ intervals, each of which we can deal with in time $O(\log^2 n)$, and afterwards we just need to find the maximum, which is done in time $O(|S|)$. □

Having reduced the number of edges in our matching to $O(\kappa n)$, we can trivially conclude that the total weight is (κn^2). We will now show that we can reduce some edge weights before applying the matching algorithm so that the total becomes $O(\kappa^2 n)$. We will further show that the weight reductions are reversible in that we will easily be able to retrieve the maximum-weighted matching on the original graph from the matching on the weight-reduced graph. With the current best algorithm for (unary) weighted bipartite matching [12], this reduction of the weights has no significance because the weights only effect the running time by a logarithmic factor. However, the point of this paper is to show an equivalence to unary weighted bipartite matching which holds even if more weight-sensitive algorithms for unary weighted bipartite matching are found. For the weight reduction, we will use the following technical observation.

OBSERVATION 6.4. *Let $G = (V_0 \cup V_1, E, W)$ be a weighted bipartite graph, and let $v \in V_0$. For all edges $\{v, w\}$, set $\Delta\{v, w\} = W\{v, w\} - \max(\{0\} \cup \{W\{v', w\} | v' \in V_0 \setminus \{v\})$. Moreover, set $\Delta(v) = \max\{\Delta\{v, w\} | \{v, w\} \in E\}$.*

If $\Delta(v) > 1$, let G' be the weighted bipartite graph obtained by reducing the weights of all edges incident with v by $\Delta(v) - 1$. Then $\Delta'(v) = 1$, and then the maximal weight matchings in G' are the same as those in G and their weights are exactly $\Delta(v) - 1$ smaller.

Proof. The observation follows directly from the fact that v has to be in any maximal matching if $\Delta(v) \geq 1$. □

Clearly, we can find $\Delta(v)$ in time linear in the number of edges. Thus for each of our matchings, we may choose a constant number of vertices v that we *reduce*, getting $\Delta'(v) \leq 1$, before we apply a matching procedure.

PROPOSITION 6.5. *The total weight of matching edges can be reduced to $O(\kappa^2 n)$ in time $O(\kappa n)$.*

Proof. The total weight of the side–side edges is at most n, so if for each matching

based on spine nodes we apply the reduction to their two core children, the total sum of their matching weights becomes $O(n)$, and if for each matching based on a spine node and a critical node we apply the reduction to the core child of the spine node, the total sum of their matching weights becomes $O(\kappa n)$. With regards to the $O(\kappa^2)$ matchings based on two critical nodes, their sum cannot exceed $O(\kappa^2 n)$ in total weight. Thus, since we have a total of $O(\kappa n)$ edges involved in the matchings, in time $O(\kappa n)$, we can reduce the total sum of the matching weights to $O(\kappa^2 n)$. □

THEOREM 6.6. *Let $M : \mathbb{R}_{\geq 1} \to \mathbb{R}$ be a monotone function bounding the time complexity* UWBM. *Moreover, let M satisfy that $M(x) = x^{1+\varepsilon} f(x)$, where $\varepsilon \geq 0$ is a constant, $f(x) = O(x^{o(1)})$, f is monotone, and for some constants b_1 and b_2, $\forall x, y \geq b_1 : f(xy) \leq b_2 f(x) f(y)$. Then, with $\kappa = \sqrt[\varepsilon]{4}$,* MAST *is computable in time $O(n^{1+o(1)} + M(n))$.*

Proof. We spend $O(n \text{ polylog } n + \texttt{time}(\texttt{UWBM}(\kappa^2 n)))$ on the matchings. Therefore, by Theorem 5.9, we have that CORE-TREES can be computed in time $O(n \text{ polylog } n + \texttt{time}(\texttt{UWBM}(\kappa^2 n)))$. Applying Theorem 4.6 gives the desired complexity. □

Inserting the best known bounds for unary weighted bipartite matching [12], with $\kappa = \sqrt[1/2]{4} = 16$, we get the following result.

COROLLARY 6.7. MAST *is computable in time $O(n^{1.5} \log n)$.* □

As a general remark, we note that we can get somewhat tighter bounds as follows. Let $\texttt{UWBM}(n, b)$ be the unary weighted bipartite matching problem in which the independent sets can be no larger than b and the sum of the weights in bounded by n. Let the $\texttt{MAST}(n, b)$ problem be the MAST problem on two n leaf trees with degree bound b. We can generalize our results to show that $\texttt{UWBM}(n, b)$ reduces linearly to $\texttt{MAST}(n, b)$. We cannot, in general, bound the work on $\texttt{MAST}(n, b)$ by $O(n^{1+o(1)} + \texttt{time}(\texttt{UWBM}(n, b)))$, but we note that using the Gabow–Tarjan algorithms gives a time of $O(n\sqrt{b} \log n)$ for $\texttt{UWBM}(n, b)$. In this case, we can bound the total work for $\texttt{MAST}(n, b)$ by $O(n^{1+o(1)} + n\sqrt{b} \log n)$, thus unifying the complexities of the bounded and general cases.

Acknowledgments. We thank the referees for their careful reading and helpful comments.

REFERENCES

[1] R. AGARWALA AND D. FERNANDEZ-BACA, *A polynomial-time algorithm for the phylogeny problem when the number of character states is fixed*, in Proc. 34th IEEE Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 140–147.

[2] S. F. ALTSCHUL, W. GISH, W. MILLER, E. W. MYERS, AND D. J. LIPMAN, *Basic local alignment search tool*, J. Molecular Biol., 215 (1990), pp. 403–410.

[3] A. AMIR AND D. KESELMAN, *Maximum agreement subtrees in multiple evolutionary trees*, in Proc. 35th IEEE Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 758–769.

[4] M. J. CHUNG, *$O(n^{2.5})$ time algorithms for the subgraph homeomorphism problem on trees*, J. Algorithms, 8 (1987), pp. 106–112.

[5] W. H. E. DAY, *Foreward: Comparison and consensus of classifications*, J. Classification, 3 (1986), pp. 183–185.

[6] M. FARACH, S. KANNAN, AND T. WARNOW, *A robust model for finding optimal evolutionary trees*, Algorithmica, 13 (1995), pp. 155–179.

[7] M. FARACH AND M. THORUP, *Fast comparison of evolutionary trees* (extended abstract), in Proc. 5th Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1994, pp. 481–488.

[8] J. S. FARRIS, *Estimating phylogenetic trees from distance matrices*, Amer. Naturalist, 106 (1972), pp. 645–668.

[9] J. Felsenstein, *Phylogenies from molecular sequences: Inference and reliability*, Annual Rev. Genetics, 22 (1988), pp. 521–565.
[10] C. R. Finden and A. D. Gordon, *Obtaining common pruned trees*, J. Classification, 2 (1985), pp. 255–276.
[11] W. M. Fitch and E. Margoliash, *The construction of phylogenetic trees*, Science, 155 (1976), pp. 29–94.
[12] H. Gabow and R. Tarjan, *Faster scaling algorithms for network problems*, SIAM J. Comput., 18 (1989), pp. 1013–1036.
[13] D. M. Hillis, *Molecular vs. morphological approaches to systematics*, Annual Rev. Systematics and Ecology, 18 (1987), pp. 23–42.
[14] J. Hunt and T. Szymanski, *A fast algortihm for computing longest common subsequences*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 350–353.
[15] F. K. Hwang and D. S. Richards, *Steiner tree problems*, Networks, 22 (1992), pp. 55–89.
[16] S. Kannan, E. Lawler, and T. Warnow, *Determining the evolutionary tree*, in Proc. 1st Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1990, pp. 475–484.
[17] E. Kubicka, G. Kubicki, and F. R. McMorris, *An algorithm to find agreement subtrees*, J. Classification, 12 (1995), pp. 91–100.
[18] G. J. Olsen. *Earliest phylogenetic branchings: Comparing rRNA-based evolutionary trees inferred with various techniques*, Cold Spring Harbor Sympos. on Quantitative Biol., 52 (1987), pp. 825–837.
[19] N. Saitou and M. Nei, *The neighbor-joining method: A new method for reconstructing phylogentic trees*, Molecular Biol. Evol., 4 (1987), pp. 406–424.
[20] P. H. A. Sneath and R. R. Sokal, *Numerical Taxonomy*, W. H. Freeman, San Francisco, 1973.
[21] M. Steel and T. Warnow, *Kaikoura tree theorems: Computing the maximum agreement subtree*, Inform. Process. Lett., 48 (1993), pp. 77–82.
[22] D. L. Swofford and G. J. Olsen, *Phylogeny reconstruction*, in Molecular Systematics, D. M. Hillis and C. Moritz, eds., Sinauer Associates Inc., Sunderland, MA, 1990, pp. 411–501.
[23] R. M. Verma, *General techniques for analyzing recursive algorithms with applications*, Technical report, Computer Science Department, University of Houston, Houston, TX, 1992.
[24] H. T. Wareham, *On the computational complexity of inferring evolutionary trees*, Master's thesis, Technical report 9301, Department of Computer Science, Memorial University of Newfoundland, St. John's, NF, Canada, 1993.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 231–242, February 1997

013

TOTAL PROTECTION OF ANALYTIC-INVARIANT INFORMATION IN CROSS-TABULATED TABLES*

MING-YANG KAO†

Abstract. To protect sensitive information in a cross-tabulated table, it is a common practice to suppress some of the cells in the table. An *analytic invariant* is a power series in terms of the suppressed cells that has a unique feasible value and a convergence radius equal to $+\infty$. Intuitively, the information contained in an invariant is not protected even though the values of the suppressed cells are not disclosed. This paper gives an optimal linear-time algorithm for testing whether there exist nontrivial analytic invariants in terms of the suppressed cells in a given set of suppressed cells. This paper also presents NP-completeness results and an almost linear-time algorithm for the problem of suppressing the minimum number of cells in addition to the sensitive ones so that the resulting table does not leak analytic-invariant information about a given set of suppressed cells.

Key words. statistical tables, data security, analytic invariants, mathematical analysis, mixed graph connectivity, graph augmentation

AMS subject classifications. 68Q22, 62A99, 05C99, 54C30

PII. S0097539793253589

1. Introduction.

Cross-tabulated tables are used in a wide variety of documents to organize and exhibit information, often with the values of some cells suppressed in order to conceal sensitive information. Concerned with the effectiveness of the practice of cell suppression [12], statisticians have raised two fundamental issues and developed computational heuristics to various related problems [5, 7, 8, 9, 10, 11, 28, 29, 30, 31]. The *detection* issue is whether an adversary can deduce significant information about the suppressed cells from the published data of a table. The *protection* issue is how a table maker can suppress a small number of cells in addition to the sensitive ones so that the resulting table does not leak significant information.

This paper investigates the complexity of how to protect a broad class of information contained in a two-dimensional table that publishes (1) the values of all cells except a set of sensitive ones, which are *suppressed*, (2) an upper bound and a lower bound for each cell, and (3) all row sums and column sums of the complete set of cells. The cells may have real or integer values. They may have different bounds, and the bounds may be finite or infinite. The upper bound of a cell should be strictly greater than its lower bound; otherwise, the value of that cell is immediately known even if that cell is suppressed. The cells that are not suppressed also have upper and lower bounds. These bounds are necessary because some of the unsuppressed cells may later be suppressed to protect the information in the sensitive cells. (See Tables 1 and 2 for an example of a complete table and its published version.)

An *unbounded feasible assignment* to a table is an assignment of values to the suppressed cells such that each row or column adds up to its published sum. A *bounded* feasible assignment is an unbounded one that also obeys the bounds of the suppressed

* A preliminary version of this work appeared in *Proc. 11th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Comput. Sci. 775, Springer-Verlag, Berlin, 1994, pp. 723–734. This research was supported in part by NSF grants MCS-8116678, DCR-8405478, and CCR-9101385.

http://www.siam.org/journals/sicomp/26-1/25358.html

† Department of Computer Science, Duke University, Durham, NC 27708 (kao@cs.duke.edu). Part of this work was done while the author was at the Department of Computer Science, Yale University, New Haven, CT 06520.

TABLE 1
A complete table.

row column index	a	b	c	d	e	f	g	h	i	row sum
1	9.5	4.5	1.5	7	1.5	1.5	5.5	2	3	36.0
2	4.5	9.5	9.5	4.5	4.5	9.5	9.5	9.5	4.5	65.5
3	6	1.5	9.5	0	9.5	6	5.5	2	5.5	45.5
4	2	1.5	4	7	1.5	4.5	9.5	5.5	2	37.5
5	1.5	5.5	4	6	5.5	0	0	4.5	9.5	36.5
6	2	3	3	4	6	5.5	2	2	9.5	37.0
column sum	25.5	25.5	31.5	28.5	28.5	27.0	32.0	25.5	34.0	

TABLE 2
A published table.

row column index	a	b	c	d	e	f	g	h	i	row sum
1			1.5	7	1.5	1.5	5.5	2	3	36.0
2										65.5
3	6	1.5				6	5.5	2	5.5	45.5
4	2	1.5	4	7	1.5			5.5	2	37.5
5	1.5	5.5	4	6	5.5					36.5
6	2	3	3	4	6	5.5	2	2		37.0
column sum	25.5	25.5	31.5	28.5	28.5	27.0	32.0	25.5	34.0	

Note: Let $X_{p,q}$ denote the cell at row p and column q. The lower and upper bounds for all suppressed cells except $X_{2,c}$ and $X_{3,c}$ are $-\infty$ and $+\infty$. The lower and upper bounds for $X_{2,c}$ and $X_{3,c}$ are 0 and 9.5.

cells. An *analytic function* of a table is a power series of the suppressed cells, each regarded as a variable, such that the convergence radius is ∞ [1, 4, 21, 22, 26, 27]. An *analytic invariant* is an analytic function that has a unique value at all the bounded feasible assignments. If an analytic invariant is formed by a linear combination of the suppressed cells, then it is called a *linear invariant* [17, 19]. Similarly, a suppressed cell is called an *invariant cell* [14, 15] if it is an invariant by itself. For instance, in Table

2, a published table, let $X_{p,q}$ be the cell at row p and column q. $X_{6,i}$ is an invariant because it is the only suppressed cell in row 6. $X_{2,c}$ and $X_{3,c}$ are invariant cells because their values are between 0 and 9.5, their sum is 19, and both cells are forced to have the same unique value 9.5. Consequently, $(X_{3,c} \cdot X_{2,c} + 0.5 \cdot X_{2,c} - 95)^2 \cdot X_{1,b} + \sin(X_{2,c} \cdot X_{2,a} - 9.5 \cdot X_{2,a})$ is also an invariant.

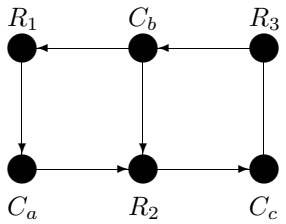
Intuitively, the information contained in an analytic invariant is unprotected because its value can be uniquely deduced from the published data. In this paper, a set of suppressed cells is *totally protected* if there exists no analytic invariant in terms of the suppressed cells in the given set, except the trivial invariant that contains no nonzero terms. Since the analytic power series form a very broad family of mathematical functions, total protection conceals from the adversary a very large class of information. This paper gives a very simple algorithm for testing whether a given set of suppressed cells is totally protected. When a graph representation, called the *suppressed graph*, of a table is given as input, this algorithm runs in optimal $O(m+n)$ time, where m is the number of suppressed cells and n is the total number of rows and columns. This paper also considers the problem of computing and suppressing the minimum number of additional cells so that a given set of original suppressed cells becomes totally protected. This problem is shown to be NP-complete. For a large class of tables, this optimal-suppression problem can be solved in $O((m+n) \cdot \alpha(n, m+n))$ time, where α is an Ackerman's inverse function and its value is practically a small constant [2, 3, 6, 16]. Moreover, for this class of tables, every optimal set of cells for additional suppression forms a spanning forest of some sort. As a consequence, at most $n-1$ additional cells need to be suppressed to achieve the total protection of a given set of original suppressed cells. Since the size of a table may grow quadratically in n, the suppression of $n-1$ additional cells is a negligible price to pay for total protection for a reasonably large table.

Previously, four other levels of data security have been considered that protect information contained, respectively, in individual suppressed cells [14, 15], in a row or column as a whole, in a set of k rows or k columns as a whole, and in a table as a whole [18]. These four levels of data security and total protection differ in two major aspects. First, these four levels of data security primarily protect information expressible as linear invariants, whereas total protection protects the much broader class of analytic-invariant information. Second, these four levels of data security emphasize protecting regular regions of a table, whereas total protection protects any given set of suppressed cells and is more flexible. These four levels of data security and total protection share some interesting similarities. As total protection corresponds to spanning forests in suppressed graphs, these four levels of data security are equivalent to some forms of 2-edge connectivity [14, 15], 2-vertex connectivity, k-vertex connectivity, and graph completeness [18]. In this paper, the NP-completeness results and efficient algorithms for total protection rely heavily on its graph characterizations. Similarly, the equivalence characterizations of these four levels of data security have been key in obtaining efficient algorithms [14, 15, 18] and NP-completeness proofs [18] for various detection and protection problems.

Section 2 discusses basic concepts. Section 3 formally defines the notion of total protection and gives a linear-time algorithm to test for this notion. Sections 4 and 5 give NP-completeness results and efficient algorithms for optimal suppression problems of total protection. Section 6 concludes this paper with discussions.

2. Basics of two-dimensional tables. This section discusses basic relationships between tables and graphs.

row column index	a	b	c	row sum
1	[0]	[9]	1	10
2	[9]	[9]	[0]	18
3	6	[0]	[5]	11
column sum	15	18	6	



In the above 3×3 table, the number in each cell is the value of that cell. A cell with a box is a suppressed cell. The lower and upper bounds of the suppressed cells are 0 and 9. The graph below the table is the suppressed graph of the table. Vertex R_p corresponds to row p, and vertex C_q to column q.

FIG. 1. *A table and its suppressed graph.*

A *mixed* graph is one that may contain both undirected and directed edges. A *traversable* cycle or path in a mixed graph is one that can be traversed along the directions of its edges. A *direction-blind* cycle or path is one that can be traversed if the directions of its edges are disregarded. The word direction-blind is often omitted for brevity. A mixed graph is *connected* (respectively, *strongly connected*) if each pair of vertices are contained in a direction-blind path (respectively, traversable cycle). A *connected component* (respectively, *strongly connected component*) of a mixed graph is a maximal subgraph that is connected (respectively, strongly connected). A set of edges in a mixed graph is an *edge cut* if its removal disconnects one or more connected components of that graph. An edge cut is a *minimal* one if it has no proper subset that is also an edge cut.

Henceforth, let $\mathcal{T}$ be a table, and let $\mathcal{H}' = (A, B, E')$ and $\mathcal{H} = (A, B, E)$ be the bipartite mixed graphs constructed below. $\mathcal{H}'$ and $\mathcal{H}$ are called the *total graph* and the *suppressed graph* of $\mathcal{T}$, respectively [15]. For each row (respectively, column) of $\mathcal{T}$, there is a unique vertex in A (respectively, B). This vertex is called a *row* (respectively, *column*) vertex. For each cell $X_{i,j}$ at row i and column j in $\mathcal{T}$, there is a unique edge e in E between the vertices of row i and column j. If the value of $X_{i,j}$ is strictly between its bounds, then e is undirected. Otherwise, if the value is equal to the lower (respectively, upper) bound, then e is directed towards to its column (respectively, row) endpoint. Note that $\mathcal{H}'$ is a *complete* bipartite mixed graph, i.e., there is exactly one edge between each pair of vertices from the two vertex sets of the graph. The graph $\mathcal{H}$ is the subgraph of $\mathcal{H}'$ whose edge set consists of only those corresponding to the suppressed cells of $\mathcal{T}$. Figure 1 illustrates a table and its suppressed graph. For

convenience, a row or column of $\mathcal{T}$ will be regarded as a vertex in $\mathcal{H}$ and a cell as an edge, and vice versa.

THEOREM 2.1 (see [15]). *A suppressed cell of $\mathcal{T}$ is an invariant cell if and only if it is not in an edge-simple traversable cycle of $\mathcal{H}$.*

The *effective area* of an analytic function F of $\mathcal{T}$, denoted by $EA(F)$, is the set of variables in the nonzero terms of F. The function F is called *nonzero* if $EA(F) \neq \emptyset$. Note that because the convergence radius of F is ∞, $EA(F)$ is independent of the point at which F is expanded into a power series.

THEOREM 2.2 (see [17]). *For every minimal edge cut Y of a strongly connected component of $\mathcal{H}$, $\mathcal{T}$ has a linear invariant F with $EA(F) = Y$.*

The *bounded kernel* (respectively, *unbounded kernel*) of $\mathcal{T}$, denoted by $BK(\mathcal{T})$ (respectively, $UK(\mathcal{T})$), is the real vector space consisting of all linear combinations of $x - y$, where x and y are arbitrary bounded (respectively, unbounded) feasible assignments of $\mathcal{T}$.

Because $\mathcal{H}$ is bipartite, every cycle of $\mathcal{H}$ is of even length. Thus the edges of an edge-simple direction-blind cycle of $\mathcal{H}$ can be alternately labeled with $+1$ and -1. Such a labeling is called a *direction-blind labeling*. A direction-blindly labeled cycle is regarded as an assignment to the suppressed cells of $\mathcal{T}$. If the corresponding edge of a suppressed cell is in the given cycle, then the value assigned to that cell is the label of that edge; otherwise, the value is 0. Note that this assignment needs not be an unbounded feasible assignment of $\mathcal{T}$.

THEOREM 2.3 (see [19]).

1. *$UK(\mathcal{T}) = BK(\mathcal{T})$ if every connected component of $\mathcal{H}$ is strongly connected.*
2. *Every direction-blindly labeled cycle of $\mathcal{H}$ is a vector in $UK(\mathcal{T})$.*

3. Total protection. A set Q of suppressed cells of $\mathcal{T}$ is *totally protected* in $\mathcal{T}$ if there is no nonzero analytic invariant F of $\mathcal{T}$ with $EA(F) \subseteq Q$. The goal of total protection can be better understood by considering Q as the set of suppressed cells that contain sensitive data. The total protection of Q means that no precise analytic information about these data, not even their row and column sums, can be deduced from the published data of $\mathcal{T}$. Since analytic power series form a very large class of functions in mathematical sciences, this notion of protection requires a large class of information about Q to be concealed from the adversary.

The next lemma and theorem characterize the notion of total protection in graph concepts.

LEMMA 3.1. *If F is a nonzero analytic invariant of $\mathcal{T}$ such that the edges in $EA(F)$ are contained in the strongly connected components of $\mathcal{H}$, then for some strongly connected component D of $\mathcal{H}$, $EA(F) \cap D$ is an edge cut of D.*

Remark. The converse of this lemma is not true; for a counterexample, consider the linear combination $X_{1,a} + 2{\cdot}X_{1,b}$ for the table in Figure 1. Also, if F is a nonzero linear invariant, then for every strongly connected component D of $\mathcal{H}$, the set $D \cap EA(F)$ is either empty or is an edge cut of D [17].

Proof. Let $\mathcal{T}_s$ be the table constructed from $\mathcal{T}$ by also publishing the suppressed cells that are not in the strongly connected components of $\mathcal{H}$. By Theorem 2.1, F remains a nonzero analytic function of $\mathcal{T}_s$. Also, the connected components of the suppressed graph $\mathcal{H}_s$ of $\mathcal{T}_s$ are the strongly connected components of $\mathcal{H}$. Thus to prove the lemma, it suffices to prove it for $\mathcal{T}_s$, $\mathcal{H}_s$, and F.

Let x_0 be a fixed bounded feasible assignment of $\mathcal{T}_s$. Let $K = \{x - x_0 | x$ is a bounded feasible assignment of $\mathcal{T}_s\}$. Since F is an analytic invariant of $\mathcal{T}_s$, the function $G(x) = F(x) - F(x_0)$ is an analytic invariant of $\mathcal{T}_s$ with $EA(G) = EA(F)$ and its value

is zero over x_0+K. Because K contains a nonempty open subset of $BK(\mathcal{T}_s)$, G is zero over $x_0 + BK(\mathcal{T}_s)$. By Theorem 2.3(1) and the strong connectivity of the connected components of $\mathcal{H}_s$, $BK(\mathcal{T}_s) = UK(\mathcal{T}_s)$ and G is zero over $x_0+UK(\mathcal{T}_s)$. Thus it suffices to show that if $D - EA(F)$ is connected for all connected components D of $\mathcal{H}_s$, then $G(x_0 + z_0) \neq 0$ for some $z_0 \in UK(\mathcal{T}_s)$. To construct z_0, let $EA(G) = \{e_1, \ldots, e_k\}$. Let D_i be the connected component of $\mathcal{H}_s$ that contains e_i. By the connectivity of $D_i - EA(F)$, there is a vertex-simple path P_i in $D_i - EA(F)$ between the endpoints of e_i. Let C_i be the vertex-simple cycle formed by e_i and P_i. Next, direction-blindly label C_i with e_i labeled $+1$. Since G is a nonzero power series, $G(x_0+y_0) \neq 0$ for some vector y_0. Note that y_0 is not necessarily in $UK(\mathcal{T}_s)$. Therefore, let $z_0 = \sum_{i=1}^{k} h_i \cdot C_i$, where h_i is the component of y_0 at variable e_i. Then by Theorem 2.3(2), $z_0 \in UK(\mathcal{T}_s)$. Because P_i is in $\mathcal{H}_s - EA(F)$, e_i appears only in the term C_i in $\sum_{i=1}^{k} h_i \cdot C_i$. Thus z_0 and y_0 have the same component values at the variables in $EA(G)$. Since the variables not in $EA(G)$ do not appear in any expansion of G, $G(x_0 + z_0) = G(x_0 + y_0) \neq 0$, proving the lemma. □

THEOREM 3.2. *A set Q of suppressed cells is totally protected in $\mathcal{T}$ if and only if the two statements below are both true:*

1. *The edges in Q are contained in the strongly connected components of $\mathcal{H}$.*
2. *For each strongly connected component D of $\mathcal{H}$, the graph $D-Q$ is connected.*

Proof. It is equivalent to show that Q is not totally protected if and only if Q contains some edges not in the strongly connected components of $\mathcal{H}$ or for some strongly connected component D of $\mathcal{H}$, the graph $D - Q$ is not connected. The $\Rightarrow$ direction follows from Lemma 3.1. As for the $\Leftarrow$ direction, if Q contains some edges not in the strongly connected components of $\mathcal{H}$, then by Theorem 2.1, Q contains some invariant cells of $\mathcal{T}$ and thus cannot be totally protected. If for some strongly connected component D of $\mathcal{H}$, the graph $D - Q$ is not connected, then some subset Y of Q is a minimal edge cut of D. By Theorem 2.2, $\mathcal{T}$ has a linear invariant F with $EA(F) = Y$ and thus Q is not totally protected. □

This paper investigates the following two problems concerning how to achieve total protection.

PROBLEM 1 (protection test).

- *Input*: The suppressed graph $\mathcal{H}$ and a set Q of suppressed cells of a table $\mathcal{T}$.
- *Output*: Is Q totally protected in $\mathcal{T}$?

THEOREM 3.3. *Problem 1 can be solved in linear time in the size of $\mathcal{H}$.*

Proof. This problem can be solved within the desired time bound by means of Theorem 3.2 and linear-time algorithms for computing connected components and strongly connected components [2, 3, 6, 16]. □

PROBLEM 2 (optimal suppression).

- *Input*: A table $\mathcal{T}$, a subset Q of E, and an integer $p \geq 0$, where E is the set of all suppressed cells in $\mathcal{T}$.
- *Output*: Is there a set P consisting of at most p published cells of $\mathcal{T}$ such that Q is totally protected in the table $\overline{\mathcal{T}}$ formed by $\mathcal{T}$ with the cells in P also suppressed?

This problem is clearly in NP. Section 4 shows that this problem with $Q = E$ is NP-complete. In contrast, section 5 proves that if the total graph of $\mathcal{T}$ is undirected, then this problem with general Q can be solved in almost linear time.

4. NP-completeness of optimal suppression. Throughout this section, the total graph of $\mathcal{T}$ may or may not be undirected.

THEOREM 4.1. *Problem* 2 *with* $Q = E$ *is NP-complete.*

To prove this theorem, the idea is to first transform Problem 2 with $Q = E$ to the following graph problem and then prove the NP-completeness of the graph problem.

PROBLEM 3.

- *Input*: A complete bipartite mixed graph $\mathcal{H}' = (A, B, E')$, a subgraph $\mathcal{H} = (A, B, E)$, and an integer $p \geq 0$.
- *Output*: Does any set P of at most p edges in $E' - E$ hold the following two properties?

  Property N1: Every connected component of $(A, B, E \cup P)$ is strongly connected.

  Property N2: The vertices of each connected component of $\mathcal{H}$ are connected in (A, B, P), i.e., contained in a connected component in (A, B, P).

LEMMA 4.2. *Problem* 2 *with* $Q = E$ *and Problem* 3 *can be reduced to each other in linear time.*

Proof. Given an instance $\mathcal{T}$ and p of Problem 2 with $Q = E$, the desired instance of Problem 3 is the total graph $\mathcal{H}' = (A, B, E')$, the suppressed graph $\mathcal{H} = (A, B, E)$ of $\mathcal{T}$, and p itself. This transformation can easily be computed in linear time. There are two directions to show that it reduces Problem 2 to Problem 3. Assume that P is a desired set for Problem 3. By Property N1, statement 1 in Theorem 3.2 is true. Also, every strongly connected component of $(A, B, E \cup P)$ is a union of edge-disjoint connected components in $\mathcal{H}$ and (A, B, P). Therefore, by Property N2, statement 2 of Theorem 3.2 holds. As a result, P itself is a desired set for Problem 2. On the other hand, assume that P is a desired set for Problem 2. Let P' be the set of all edges in P that are also in the strongly connected components of $(A, B, E \cup P)$. By statement 1 of Theorem 3.2 and the total protection of E in $\overline{\mathcal{T}}$, the connected components of $(A, B, E \cup P')$ are the strongly connected components of $(A, B, E \cup P)$. Thus P' holds Property N1. Next, because a connected component of $\mathcal{H}$ is included in a strongly connected component of $(A, B, E \cup P')$, by statement 2 of Theorem 3.2, P' also holds Property N2 and thus is a desired set for Problem 3.

Given an instance $\mathcal{H}'$, $\mathcal{H}$, and p of Problem 3, the desired instance of Problem 2 with $Q = E$ is p itself and the table defined as follows. For each vertex in A (respectively, B), there is a row (respectively, column). The upper and lower bounds for each cell are 2 and 0. For each edge e in E', its corresponding cell is at the row and column corresponding to its endpoints. The value of that cell is 1 (respectively, 0 and 2) if e is undirected (respectively, directed from A to B or directed from B to A). For each edge e in $\mathcal{H}$, its corresponding cell is suppressed. Note that the total and suppressed graphs of this table are $\mathcal{H}'$ and $\mathcal{H}$ themselves. Thus the remaining proof details for this reduction are essentially the same as for the other reduction. □

Both Problem 2 with $Q = E$ and Problem 3 are clearly in NP. To prove their completeness in NP, by Lemma 4.2, it suffices to reduce the following NP-complete problem to Problem 3.

PROBLEM 4 (hitting set [13]).

- *Input*: A finite set S, a nonempty family W of subsets of S, and an integer $h \geq 0$.
- *Output*: Is there a subset S' of S such that $|S'| \leq h$ and S' contains at least one element in each set in W?

Given an instance $S = \{s_1, \ldots, s_q\}$, $W = \{S_1, \ldots, S_r\}$, h of Problem 4, an instance $\mathcal{H}' = (A, B, E')$, $\mathcal{H} = (A, B, E)$, p of Problem 3 is constructed as follows:

- *Rule* 1: Let $A = \{a_0, a_1, \ldots, a_q\}$. The vertices $a_1, \ldots, a_q$ correspond to

$s_1, \ldots, s_q$, but a_0 corresponds to no s_i.

- *Rule* 2: Let $B = \{b_0, b_1, \ldots, b_r\}$. The vertices $b_1, \ldots, b_r$ correspond to $S_1, \ldots, S_r$ of S, but b_0 corresponds to no S_j.
- *Rule* 3: Let E' be the union of the following sets of edges:
  1. $\{b_0 \to a_0\}$;
  2. $\{a_0 \to b_j \mid \forall\ j \text{ with } 1 \le j \le r\}$;
  3. $\{a_i \to b_0 \mid \forall\ i \text{ with } 1 \le i \le q\}$;
  4. $\{b_j \to a_i \mid \forall\ s_i \text{ and } S_j \text{ with } s_i \in S_j\}$;
  5. $\{a_i \to b_j \mid \forall\ s_i \text{ and } S_j \text{ with } s_i \notin S_j\}$.
- *Rule* 4: Let $E = \{a_0 \to b_1, \ldots, a_0 \to b_r\}$.
- *Rule* 5: Let $p = h + r + 1$.

The above construction can easily be computed in polynomial time. The next two lemmas show that it is indeed a desired reduction.

LEMMA 4.3. *If some set $S' \subseteq S$ with $|S'| \le h$ contains at least one element in each S_j, then there is a set $P \subseteq E' - E$ consisting of at most p edges that holds Properties* N1 *and* N2.

Proof. For each S_j, let s_{i_j} be an element in $S' \cap S_j$; by the assumption of this lemma, these elements exist. Next, let $P_1 = \{b_1 \to a_{i_1}, \ldots, b_r \to a_{i_r}\}$ and $P_2 = \{a_{i_1} \to b_0, \ldots, a_{i_r} \to b_0\}$; by Rule 3, these two sets exist. Now let $P = P_1 \cup P_2 \cup \{b_0 \to a_0\}$. Note that $P \subseteq E' - E$. Since P_1 consists of r edges and P_2 consists of at most $|S'|$ edges, P has at most p edges. P holds Property N1 because $E \cup P$ consists of the edges in the traversable cycles $b_0 \to a_0$, $a_0 \to b_j$, $b_j \to a_{i_j}$, and $a_{i_j} \to b_0$. Property N2 of P follows from the fact that P connects $\{a_0, b_1, \ldots, b_r\}$, which forms the only connected component of $\mathcal{H}$ with more than one vertex. □

LEMMA 4.4. *If some set $P \subseteq E' - E$ consisting of at most p edges holds Properties* N1 *and* N2, *then there exists a set $S' \subseteq S$ with $|S'| \le h$ that contains at least one element in each S_j.*

Proof. By Property N1, P must contain some edge $b_j \to a_{i_j}$ for each j with $1 \le j \le r$. By Rule 3(4), $s_{i_j} \in S_j$. Now let $S' = \{s_{i_1}, \ldots, s_{i_r}\}$. To calculate the size of S', note that by Property N1, P must also contain $b_0 \to a_0$ and at least one edge leaving a_{i_j} for each j. Thus $|P| \ge |S'| + r + 1$. Then $|S'| \le h$ because $|P| \le p = r + h + 1$. □

The above lemma completes the proof of Theorem 4.1.

5. Optimal suppression in almost linear time. Under the assumption that the total graph of $\mathcal{T}$ is undirected, this section considers the following optimization version of Problem 2.

PROBLEM 5 (optimal suppression).

- *Input*: The suppressed graph $\mathcal{H} = (A, B, E)$ of a table $\mathcal{T}$ and a subset Q of E.
- *Output*: A set P consisting of the smallest number of published cells in $\mathcal{T}$ such that Q is totally protected in the table $\overline{\mathcal{T}}$ formed by $\mathcal{T}$ with the cells in P also suppressed.

For all positive integers n and m, let α denote the best known function such that $m + n$ unions and finds of disjoint subsets of an n-element set can be performed in $O((m+n)\cdot\alpha(n, m+n))$ time [2, 3, 6, 16].

THEOREM 5.1. *Problem* 5 *can be solved in $O((m+n)\cdot\alpha(n, m+n))$ time, where m is the number of suppressed cells and n is the total number of rows and columns in $\mathcal{T}$.*

To prove Theorem 5.1, Problem 5 is first converted to the next problem.

PROBLEM 6.

- *Input*: An undirected bipartite graph $\mathcal{H} = (A, B, E)$ and a subset Q of E.
- *Output*: A forest P formed by the smallest number of undirected edges between A and B but not in E such that the vertices of each connected component of (A, B, Q) are connected in $(A, B, (E - Q) \cup P)$, i.e., contained in a connected component of $(A, B, (E - Q) \cup P)$.

LEMMA 5.2. *Problems* 5 *and* 6 *can be reduced to each other in linear time.*

Proof. The proof uses arguments similar to those in the proof of Lemma 4.2. The strong connectivity properties in Problem 3 and Theorem 3.2 can be ignored because this section assumes that the total graph of $\mathcal{T}$ is undirected. The forest structure of P follows from its minimality. □

Note that because $Q \subseteq E$, the vertices of each connected component of (A, B, Q) are connected in $(A, B, (E - Q) \cup P)$ if and only if the vertices of each connected component of $\mathcal{H}$ are connected in $(A, B, (E - Q) \cup P)$. Using this equivalence, the next stage of the proof of Theorem 5.1 further reduces Problem 6 to another graph problem with the steps below:

M1. Compute the connected components $D_1, \ldots, D_r$ of $\mathcal{H}$.

M2. For each D_i, compute a maximal forest K_i over the vertices of D_i using only the edges in $E - Q$.

M3. For each D_i, extend K_i to a maximal forest L_i over the vertices of D_i using additional edges only from the complement graph D_i^c of D_i.

M4. Construct a graph $\hat{\mathcal{H}}$ from $\mathcal{H}$ by contracting each tree in each L_i into a single vertex.

M5. For each D_i, compute its contracted version $\hat{D}_i$ in $\hat{\mathcal{H}}$.

M6. Divide the vertices of $\hat{\mathcal{H}}$ into three sets, V_A, V_B, and V_{AB}, where a vertex in V_A (respectively, V_B) consists of a single vertex from A (respectively, B) and a vertex in V_{AB} contains at least two vertices (thus with at least one from each of A and B).

A set of undirected edges between vertices in V_A, V_B, and V_{AB} is called *semitripartite* if every edge in that set is between two of the three sets or is between two vertices in V_{AB}. Note that the set of edges in $\hat{\mathcal{H}}$ is semitripartite.

PROBLEM 7.

- *Input*: Three disjoint finite sets V_A, V_B, and V_{AB} and a partition $\hat{D}_1, \ldots, \hat{D}_r$ of $V_A \cup V_B \cup V_{AB}$.
- *Output*: A semitripartite set $\hat{P}$ consisting of the smallest number of edges such that no edge in $\hat{P}$ connects two vertices in the same D_i and the vertices in each D_i are connected in the graph formed by is $\hat{P}$.

LEMMA 5.3. *Problem* 6 *can be reduced to Problem* 7 *in* $O((m + n) \cdot \alpha(n, m + n))$ *time, where* m *is the number of edges and* n *is the number of vertices in* $\mathcal{H}$.

Proof. The key idea is that an optimal P for Problem 6 can be obtained by connecting the vertices of each D_i first with edges in $E - Q$, which can be used for free, next with edges in D_i^c, and then with edges outside $D_i \cup D_i^c$. Let P' be a set of $|\hat{P}|$ edges in the complement of $\mathcal{H}$ that becomes $\hat{P}$ after step M4. Then $P' \cup (L_1 - K_1) \cup \cdots \cup (L_r - K_r)$ is a desired output P for Problem 6, showing that steps M1–M6 can indeed reduce Problem 6 to Problem 7. Step M3 is the only step that requires more than linear time. It is important to avoid directly computing D_i^c at step M3. Computing these complement graphs takes $\Theta(|A| \cdot |B|)$ time if some D_i contains a constant fraction of the vertices in $\mathcal{H}$. In such a case, if $\mathcal{H}$ is sparse, then the time spent on computing D_i^c alone is far greater than the desired complexity. Instead of this naïve approach, step M3 uses efficient techniques recently developed

for complement-graph problems [20] and takes the desired $O((m+n)\cdot\alpha(n,m+n))$ time. □

The last stage of the proof of Theorem 5.1 is to give a linear-time algorithm for Problem 7. A component $\hat{D}_i$ is *good* if it has at least two vertices with at least one from V_{AB}; it is *bad* if it has at least two vertices with none from V_{AB} (and thus with at least one from each of V_A and V_B). The goal is to use as few edges as possible to connect the vertices in each of these components. Let w_g and w_b be the numbers of good and bad components, respectively. There are three cases based on the value of w_g.

Case 1: $w_g = 0$. If $w_b = 0$, then let $\hat{P} = \emptyset$ because no $\hat{D}_i$ needs to be connected. If $w_b > 0$ and $|V_{AB}| > 0$, then include in $\hat{P}$ an edge between each vertex in the bad components and an arbitrary vertex in V_{AB}. If $w_b > 0$ and $|V_{AB}| = 0$, then there does not exist a desired $\hat{P}$ and the given instance of Problem 7 has no solution.

Case 2: $w_g = 1$. Let $\hat{D}_j$ be the unique good component.

If $w_b > 0$, then find a bad component $\hat{D}_k$ and three vertices $u \in V_{AB} \cap \hat{D}_j$, $v_1 \in V_A \cap \hat{D}_k$, and $v_2 \in V_B \cap \hat{D}_k$. Next, include in $\hat{P}$ an edge between v_2 and each vertex in $(\hat{D}_j \cap (V_A \cup V_{AB})) - \{u\}$, an edge between v_1 and each vertex in $\hat{D}_j \cap V_B$, and an edge between u and each vertex in the bad components.

If $w_b = 0$ and $V_{AB} - \hat{D}_j \neq \emptyset$, then include in $\hat{P}$ an edge between every vertex in $\hat{D}_j$ and an arbitrary vertex in $V_{AB} - \hat{D}_j$.

If $w_b = 0$ and $V_{AB} - \hat{D}_j = \emptyset$, then there are sixteen subcases depending on whether $V_A \cap \hat{D}_j = \emptyset$, $V_A - \hat{D}_j = \emptyset$, and $V_B \cap \hat{D}_j = \emptyset$, $V_B - \hat{D}_j = \emptyset$. If $V_A \cap \hat{D}_j \neq \emptyset$, $V_A - \hat{D}_j \neq \emptyset$, $V_B \cap \hat{D}_j \neq \emptyset$, and $V_B - \hat{D}_j \neq \emptyset$, then include in $\hat{P}$ an edge between each vertex in $V_A \cap \hat{D}_j$ and a vertex $v_2 \in V_B - \hat{D}_j$, an edge between each vertex in $V_B \cap \hat{D}_j$ and a vertex $v_1 \in V_A - \hat{D}_j$, and an edge between v_1 and each vertex in $V_{AB} \cup \{v_2\}$. The other fifteen subcases are handled similarly.

Case 3: $w_g \geq 2$. Let d be the total number of vertices in the good and bad components. Let w' be the number of connected components in $\hat{P}$ that contain the vertices of at least one good or bad $\hat{D}_i$; let d' be the number of vertices in these connected components of $\hat{P}$ that are not in any good or bad $\hat{D}_i$. By its minimality, $\hat{P}$ forms a forest and $|\hat{P}| = d' + d - w'$. The techniques for Cases 1 and 2 can be used to show that there exists an optimal $\hat{P}$ with $d' = 0$. Thus to minimize $|\hat{P}|$ is to maximize w'. Because two bad components cannot be connected by edges between them alone, the strategy for maximizing w' is to pair a good component with a bad one, whenever possible, and include in $\hat{P}$ edges between them to connect their vertices into a tree. After this step, if there remain unconnected bad components but no unconnected good ones, then add to P an edge between each vertex in the remaining bad components and an arbitrary vertex in the intersection of V_{AB} and a good component. On the other hand, if there remain good components but no bad ones, then pair up these good components similarly. After this step, if there remains a good component, then add to $\hat{P}$ an edge between each vertex in this last good component and an arbitrary vertex in the intersection of V_{AB} and another good component. (As a result, if $w_g \leq w_b$, then $|\hat{P}| = d - w_g$; otherwise, $|\hat{P}| = d - \lfloor \frac{w_g + w_b}{2} \rfloor$.)

The above discussion yields a linear-time algorithm for Problem 7 in a straightforward manner. This finishes the proof of Theorem 5.1.

6. Discussions. Lemma 5.2 has several significant implications. Since P is a forest, it has at most $n-1$ edges. Thus for a table with an undirected total graph, no more than $n-1$ additional cells need to be suppressed to achieve total protection. This

is a small number compared to the size of the table, which may grow quadratically in n. Moreover, when $\mathcal{H}$ is connected and $E = Q$, (A, B, P) is a spanning tree. In this case, many well-studied tree-related computational concepts and tools, such as minimum-cost spanning trees, can be applied to consider other optimal suppression problems for total protection.

Acknowledgments. The author is deeply grateful to Dan Gusfield for his constant encouragement and help. The author wishes to thank an anonymous referee for very helpful and thorough comments. The referee has also pointed out that some very interesting materials related to Theorems 2.2 and 2.3 have been developed in the context of protecting sums of suppressed cells [23, 24, 25].

REFERENCES

[1] L. Ahlfors, *Complex Analysis*, McGraw–Hill, New York, 1979.
[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.
[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison–Wesley, Reading, MA, 1983.
[4] T. M. Apostol, *Mathematical Analysis*, Addison–Wesley, Reading, MA, 1974.
[5] G. J. Brackstone, L. Chapman, and G. Sande, *Protecting the confidentiality of individual statistical records in Canada*, in Proc. Conference of the European Statisticians 31st Plenary Session, Conference of European Statistics, Geneva, 1983.
[6] T. H. Cormen, C. L. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1991.
[7] L. H. Cox, *Disclosure analysis and cell suppression*, in Proceedings of the American Statistical Association, Social Statistics Section, American Statistical Association, Alexandria, VA, 1975, pp. 380–382.
[8] L. H. Cox, *Suppression methodology in statistics disclosure*, in Proc. American Statistical Association, Social Statistics Section, American Statistical Association, Alexandria, VA, 1977, pp. 750–755.
[9] L. H. Cox, *Automated statistical disclosure control*, in Proc. American Statistical Association, Survey Research Method Section, American Statistical Association, Alexandria, VA, 1978, pp. 177–182.
[10] L. H. Cox, *Suppression methodology and statistical disclosure control*, J. Amer. Statist. Assoc., 75 (1980), pp. 377–385.
[11] L. H. Cox and G. Sande, *Techniques for preserving statistical confidentiality*, in Proc. 42nd Session of the International Statistical Institute, the International Association of Survey Statisticians, Voorburg, The Netherlands, 1979.
[12] D. Denning, *Cryptography and Data Security*, Addison–Wesley, Reading, MA, 1982.
[13] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.
[14] D. Gusfield, *Optimal mixed graph augmentation*, SIAM J. Comput., 16 (1987), pp. 599–612.
[15] D. Gusfield, *A graph theoretic approach to statistical data security*, SIAM J. Comput., 17 (1988), pp. 552–571.
[16] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, New York, 1976.
[17] M. Y. Kao, *Minimal linear invariants*, in Algorithms, Concurrency and Knowledge, Proc. 1995 Asian Computing Science Conference, K. Kanchanasut and J. J. Levy, eds., Lecture Notes in Comput. Sci. 1023, Springer-Verlag, New York, 1995, pp. 23–33.
[18] M. Y. Kao, *Data security equals graph connectivity*, SIAM J. Discrete Math., 9 (1996), pp. 87–100.
[19] M. Y. Kao and D. Gusfield, *Efficient detection and protection of information in cross tabulated tables* I: *Linear invariant test*, SIAM J. Discrete Math., 6 (1993), pp. 460–476.
[20] M. Y. Kao and S. H. Teng, *Simple and efficient compression schemes for dense and complement graphs*, in Proc. 5th International Symposium on Algorithms and Computation, D. Z. Du and X. S. Zhang, eds., Lecture Notes in Comput. Sci. 834, Springer-Verlag, New York, 1994, pp. 201–210.
[21] S. Lang, *Complex Analysis*, Springer-Verlag, New York, 1985.

[22] L. H. Loomis and S. Sternberg, *Advanced Calculus*, Addison–Wesley, Reading, MA, 1968.
[23] F. M. Malvestuto, *A universal-scheme approach to statistical databases containing homogeneous summary tables*, ACM Trans. Database Systems, 18 (1993), pp. 679–708.
[24] F. M. Malvestuto and M. Moscarini, *Query evaluability in statistical databases*, IEEE Trans. Knowledge Data Engrg., 2 (1990), pp. 425–430.
[25] F. M. Malvestuto, M. Moscarini, and M. Rafanelli, *Suppressing marginal cells to protect sensitive information in a two-dimensional statistical table*, in Proc. ACM Symposium on Principles of Database Systems, ACM, New York, 1991, pp. 252–258.
[26] H. L. Royden, *Real Analysis*, Macmillan, New York, 1988.
[27] W. Rudin, *Principles of Mathematical Analysis*, McGraw–Hill, New York, 1975.
[28] G. Sande, *Towards automated disclosure analysis for establishment based statistics*, Technical report, Statistics Canada, Ottawa, ON, 1977.
[29] G. Sande, *A theorem concerning elementary aggregations in simple tables*, Technical report, Statistics Canada, Ottawa, ON, 1978.
[30] G. Sande, *Automated cell suppression to preserve confidentiality of business statistics*, Statist. J. United Nations, 2 (1984), pp. 33–41.
[31] G. Sande, *Confidentiality and polyhedra: An analysis of suppressed entries on cross tabulations*, Technical report, Statistics Canada, Ottawa, ON.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 243–254, February 1997

014

ON THE POWER OF REAL TURING MACHINES OVER BINARY INPUTS*

FELIPE CUCKER† AND DIMA GRIGORIEV‡

Abstract. In this paper, we study the computational power of real Turing machines over binary inputs. Our main result is that the class of binary sets that can be decided by real Turing machines in parallel polynomial time is exactly the class PSPACE/*poly*.

Key words. real-number machines and computations, complexity classes

AMS subject classifications. 68Q05, 68Q15

PII. S0097539794270340

Introduction. In recent years, the study of the complexity of computational problems involving real numbers has been an increasing research area. A foundational paper has been [5], where a computational model—the real Turing machine—for dealing with the above problems was developed.

One research direction that has been studied intensively during the last two years is the computational power of real Turing machines over binary inputs. The general problem can be roughly stated in the following way. Let us consider a class C of real Turing machines that work under some resource bound (for instance, polynomial time, branching only on equality, etc.). If we restrict these machines to work on binary inputs (i.e., finite words over $\{0,1\}$), they define a class of binary languages D. The question is, what can we say about D depending on C?

More formally, let us denote by $\mathbb{R}^\infty$ the direct sum of countably many copies of $\mathbb{R}$ and let $\mathcal{P}(\mathbb{R}^\infty)$ be the set of its subsets. Also, let us denote by Σ the subset $\{0,1\}$ of $\mathbb{R}$ and—as usual—by Σ^* the subset of $\mathbb{R}^\infty$ consisting of those vectors whose components are in Σ. Given any complexity class $\mathcal{C} \subseteq \mathcal{P}(\mathbb{R}^\infty)$, we define its Boolean part to be the class of binary languages

$$\mathrm{BP}(\mathcal{C}) = \{X \cap \Sigma^* : X \in \mathcal{C}\}.$$

Our problem now can be stated as follows: given a complexity class of real sets $\mathcal{C}$ characterize $\mathrm{BP}(\mathcal{C})$.

A possible origin of the problem is the recent interest in the computational power of neural networks. The first results characterized the power of nets with rational weights working within polynomial time by showing that they compute exactly the sets in P (cf. [28]). The same problem was then considered for neural networks with real weights, and it was shown that the power of these nets working within polynomial time is exactly P/*poly* (cf. [29] and [23]).

This latter problem considers in a natural way a setting in which an algebraic model having real constants operates over binary inputs. A next step was then taken

* Received by the editors June 27, 1994; accepted for publication (in revised form) April 28, 1995. http://www.siam.org/journals/sicomp/26-1/27034.html

† Universitat Pompeu Fabra, Balmes 132, Barcelona 08008, Spain (cucker@upf.es). The research of this author was partially supported by DGICyT PB 920498 and EC ESPRIT BRA Program contracts 7141 and 8556, projects ALCOM II and NeuroCOLT.

‡ Departments of Computer Science and Mathematics, Pennsylvania State University, University Park, PA 16802 (dima@cse.psu.edu). The research of this author was partially supported by the Volkswagen–Stiftung.

by Koiran, who passed from a structured model—the neural net—to a general one—the real Turing machine. However, he did not deal with the real Turing machine as it was introduced in [5] but with a restricted version of it that can do only a moderate use of multiplication, namely, all rational functions intermediately computed (in the input variables as well as in the machine's constants) must have degree and coefficient size bounded by the running time. For this weak model, he considered the class P_{W} of sets accepted in polynomial time and he proved that $\mathrm{BP}(\mathrm{P}_{\mathrm{W}}) = \mathrm{P}/\mathit{poly}$ (see [20]).

Subsequently, several papers exhibited new results on Boolean parts. In [13], it was shown that $\mathrm{BP}(\mathrm{PAR}_{\mathrm{W}}) = \mathrm{PSPACE}/\mathit{poly}$, where $\mathrm{PAR}_{\mathrm{W}}$ is the class of subsets of $\mathbb{R}^\infty$ decided in weak parallel polynomial time. Also, for additive machines (i.e., real Turing machines that do not perform multiplications at all), it was shown in [21] that $\mathrm{BP}(\mathrm{P}_{\mathrm{add}}) = \mathrm{P}/\mathit{poly}$ and $\mathrm{BP}(\mathrm{NP}_{\mathrm{add}}) = \mathrm{NP}/\mathit{poly}$. Here $\mathrm{P}_{\mathrm{add}}$ and $\mathrm{NP}_{\mathrm{add}}$ denote the obvious classes, but we recall that the nondeterministic guesses in this model are real numbers. Moreover, if the machines are order free, i.e., they are required to branch only on equality tests, we now have that $\mathrm{BP}(\mathrm{P}^{=}_{\mathrm{add}}) = \mathrm{P}$ and that $\mathrm{BP}(\mathrm{NP}^{=}_{\mathrm{add}}) = \mathrm{NP}$ [21]. These results were subsequently generalized in [11] to all the levels of the polynomial hierarchy constructed upon $\mathrm{NP}_{\mathrm{add}}$ (or $\mathrm{NP}^{=}_{\mathrm{add}}$) as well as to the class $\mathrm{PAR}_{\mathrm{add}}$ (or $\mathrm{PAR}^{=}_{\mathrm{add}}$) of sets computed in parallel polynomial time whose Boolean part is proven to be PSPACE/*poly* (respectively, PSPACE).

None of the results mentioned was done for the (unrestricted) real Turing machine. In fact, for this case, it was even asked whether there exists a subset of Σ^* not belonging to the $\mathrm{BP}(\mathrm{P}_{\mathbb{R}})$ (cf. [14]). First steps in this direction were done in [22], where it was shown that if we consider order-free machines, then we have the inclusion $\mathrm{BP}(\mathrm{P}^{=}_{\mathbb{R}}) \subseteq \mathrm{BPP}$ (the class of sets decided by randomized machines in polynomial time with bounded probability error; see [1, Chapter 6]) as well as a positive answer to the question above. In fact, if $\mathrm{PH}_{\mathbb{R}}$ is the polynomial hierarchy constructed upon $\mathrm{NP}_{\mathbb{R}}$, the existence of binary languages not belonging to $\mathrm{BP}(\mathrm{PH}_{\mathbb{R}})$ (and a fortiori not belonging to $\mathrm{BP}(\mathrm{P}_{\mathbb{R}})$) was also proved in [22].

The aim of this paper is to prove that $\mathrm{BP}(\mathrm{PAR}_{\mathbb{R}}) = \mathrm{PSPACE}/\mathit{poly}$, where $\mathrm{PAR}_{\mathbb{R}}$ is the class of sets computed in parallel polynomial time by (ordinary) real Turing machines. As a consequence, we obtain the existence of binary sets that do not belong to the Boolean part of $\mathrm{PAR}_{\mathbb{R}}$ (an extension of the result in [22] since $\mathrm{PH}_{\mathbb{R}} \subseteq \mathrm{PAR}_{\mathbb{R}}$). Also, a separation of complexity classes in the real setting follows. If we consider the class $\mathrm{EXP}_{\mathrm{W}}$ of subsets of $\mathbb{R}^\infty$ accepted by RTM in weak exponential time as defined in [20], the Boolean part above implies that the inclusion $\mathrm{PAR}_{\mathbb{R}} \subset \mathrm{EXP}_{\mathrm{W}}$ is strict.

Refining our main theorem a bit, we show that it is even possible to allow a polynomial advice (i.e., a polynomially long sequence of real numbers) to $\mathrm{PAR}_{\mathbb{R}}$ without modifying its Boolean part—in other words, that we have $\mathrm{BP}(\mathrm{PAR}_{\mathbb{R}})/\mathit{poly} = \mathrm{PSPACE}/\mathit{poly}$. Since it is known from [11] that the Boolean part of $\mathrm{PAR}_{\mathrm{add}}$ (where no multiplications are allowed) is PSPACE/*poly*, we deduce that multiplication or nonuniformity (under the form of a polynomial advice) are of no help to decide binary sets in the presence of parallelism.

1. Some geometrical background. In this paper, $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$, and $\mathbb{C}$ denote the sets of natural, integer, rational, real, and complex numbers, respectively. By $\mathbb{R}_{\mathrm{alg}}$ we denote the real closure of $\mathbb{Q}$, i.e., the field of all real algebraic numbers. Also, for any polynomial f with integer coefficients, we shall denote by $|\mathrm{coeff}(f)|$ the maximal absolute value of its coefficients.

The aim of this section is to show how to find real algebraic points in the connected components of nonempty open sets. We closelly follow [16]. Thus let $g_1, \ldots, g_N \in$

$\mathbb{Z}[X_1, \ldots, X_k]$ and let

$$V = \{x \in \mathbb{R}^k : g_1(x) > 0 \& \cdots \& g_N(x) > 0\}$$

be an open nonempty semialgebraic set. For the rest of this section, we consider d a bound on the degree of each g_i and consider L a bound for all $|\text{coeff}(g_i)|$.

LEMMA 1.1 (see [16, Lemma 10]). *Let $g = \prod_{i=1}^{N} g_i$ and d_g be the degree of g. Then there exists a positive integer γ_1 such that any connected component of V has a nonempty intersection with the ball $B(R)$, where*

$$R = L^{{d_g}^{k\gamma_1}}. \qquad \square$$

Let us now recall (see [6, section 9.5]) that a point $a \in \mathbb{R}^k$ is a *critical point* for a function $f : \mathbb{R}^k \to \mathbb{R}$ when it satisfies

$$\frac{\partial f}{\partial X_1}(a) = \cdots = \frac{\partial f}{\partial X_k}(a) = 0.$$

In this case, the value $b = f(a)$ is said to be a *critical value* of f. In the case when f is a polynomial function, Sard's lemma (see [6, Théorème 9.5.2] or [24]) implies that there are only a finite number of critical values of f.

This last fact was used in [16] (and in several subsequent papers) to reduce the dimension of nonempty semialgebraic sets to zero (thus avoiding cascading of projections) in the algorithm for deciding emptiness of semialgebraic sets.

Let us now consider the polynomials

$$g_0 = R^2 - \sum_{i=1}^{k} X_i^2$$

and

$$G = g_0 \prod_{i=1}^{N} g_i.$$

We have that $\deg G = d_G < Nd + 2$ and $L_G = |\text{coeff}(G)| \leq L^{{d_G}^{O(k)}} (Ld^k)^{O(N)}$ due to Lemma 1.1.

The following result gives a bound on the small critical values of G.

LEMMA 1.2. *There exists a positive integer γ_2 such that for every nonzero critical value b of G, we have $|b| > C^{-1}$, where*

$$C = L_G^{{d_G}^{k\gamma_2}}.$$

Proof. Let us consider the system of equations in the variables $X_1, \ldots, X_k, Z$,

$$G - Z = \frac{\partial G}{\partial X_1} = \cdots = \frac{\partial G}{\partial X_k} = 0,$$

as well as its set of solutions $S \subseteq \mathbb{R}^{k+1}$. On any connected component of S, the coordinate Z, being the critical value of G, is constant since G is continuous and due to Sard's lemma. Now, since the degrees and the coefficients of the polynomials appearing in this system are bounded by d_G and $O(L_G d_G)$, respectively, if we apply

the quantifier-elimination algorithm given in [18] or [25] along $X_1, \ldots, X_k$ onto Z, we get a finite set of points in $\mathbb{R}$ (just the critical values) such that each nonzero point has absolute value greater than

$$\left(L_G^{d_G{}^{k\gamma_2}}\right)^{-1}. \qquad \square$$

Remark 1. In the preceding proof, the use of quantifier elimination is not strictly necessary. One can instead use the bounds for the representative points from the connected components of S given in the main theorem of [16].

Because of the preceding lemma, we have that the algebraic set

$$W_0 = \{x \in \mathbb{R}^k : G(x) = C^{-1}\}$$

is a nonsingular, closed hypersurface with the property that each connected component of $V \cap B(R)$ contains at least one (bounded) connected component of W_0 (cf. [24]). Note that W_0 do not intersect the boundary of $B(R)$.

Now Lemma 5 of [16] asserts the existence of integers $0 \leq v_2, \ldots, v_n \leq (2d_G)^k$ such that the system

$$G - C^{-1} = \left(\frac{\partial G}{\partial X_2}\right)^2 - \frac{v_2}{(2d_G)^k k}\Delta = \cdots = \left(\frac{\partial G}{\partial X_k}\right)^2 - \frac{v_k}{(2d_G)^k k}\Delta = 0,$$

where $\Delta = \sum_{i=1}^k (\frac{\partial G}{\partial X_i})^2$, has a finite number of solutions in $\mathbb{R}^k$. Moreover, each of these solutions is an absolutely irreducible zero–dimensional component of the variety in C^k given by this system of equations. Due to Bezout's inequality, the number of real solutions is bounded by $(2d_G)^k$. Besides (cf. Lemma 4 in [16]), each bounded connected component of W_0 contains a point satisfying the system.

We can summarize the preceding results in the following theorem, which will be our main technical tool in the next section.

THEOREM 1.3. *Let $g_1, \ldots, g_N \in \mathbb{Z}[X_1, \ldots, X_k]$ satisfy for every $i \leq N$ the bounds $\deg(g_i) \leq d$ and $|\mathrm{coeff}(g_i)| \leq L$. Then with the notations introduced above, there are integers $0 \leq v_2, \ldots, v_k \leq (2d_G)^k$ such that the set $W \subseteq \mathbb{R}^k$ defined by*

$$G - C^{-1} = \left(\frac{\partial G}{\partial X_2}\right)^2 - \frac{v_2}{(2d_G)^k k}\Delta = \cdots = \left(\frac{\partial G}{\partial X_k}\right)^2 - \frac{v_k}{(2d_G)^k k}\Delta = 0$$

is finite. Moreover, the number of its points does not exceed $(2d_G)^k$ and every connected component of

$$V = \{x \in \mathbb{R}^k : g_1(x) > 0 \,\&\, \cdots \,\&\, g_N(x) > 0\}$$

contains at least one point of W. $\square$

2. Some background on machine models. In the following sections, we shall deal with real Turing machines (or RTMs for short) as introduced in [5].

Let us give a brief description of them for the finite-dimensional case. A finite-dimensional RTM is specified by an input space $\mathbb{R}^n$, an output space $\mathbb{R}^m$, and a state space $\mathbb{R}^s$ $(s \geq n, m)$ together with a finite directed graph whose nodes, labeled $\{1, \ldots, N\}$, are of four different types. The first one is the *input* node that initializes the computation by locating the input $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ in the first n coordinates of the ouput space and 0 in all of the others. It has only one next node. Then there

are *computation* nodes. They have associated a rational function in some of the s coordinates of the state space and they replace one such coordinate by the real number obtained by evaluating this rational function in the actual content of the state space. They also have only one next node. There are also *branch* nodes. They check whether $z_i \geq 0$ for a certain coordinate of the state space and select one of their two possible next nodes according to the answer. Finally, there is one *output* node. It has no next node, and therefore when it is reached, the computation halts. Moreover, the first m coordinates of the state space are mapped into the output space and constitute the output of the machine.

If the computation takes time at most t for all inputs, the RTM M computes a total function from $\mathbb{R}^n$ to $\mathbb{R}^m$. A key fact in what follows is that in this case, M can also be "unwound" into an algebraic computation tree T_M (see [3]) that has depth t and computes the same function, and at each branching node i of this tree, the value whose sign is tested can be written as $h_i(\alpha_1, \ldots, \alpha_k, x_1, \ldots, x_n)$, where $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is the input, $\alpha_1, \ldots, \alpha_k \in \mathbb{R}$ are the possible coefficients of the functions associated with the computation nodes of M, and

$$h_i \in \mathbb{Q}(Z_1, \ldots, Z_k, X_1, \ldots, X_n).$$

For the infinite-dimensional case, we want to consider as inputs and outputs real vectors of arbitrary dimension. Recall that we denote by $\mathbb{R}^\infty$ the direct sum $\bigoplus_{i\in\mathbb{N}} \mathbb{R}$. Then we want to replace the input, output, and state spaces in the description above for $\mathbb{R}^\infty$. Since the number of computation nodes is finite, the same thing happens for the number of coordinates that can be accessed or modified. Thus some mechanism for "moving" information inside the state space is needed. We refer the reader to [5] for the details of how this is done as well as for a comprehensive introduction to these machines. We recall from there that we denote by $\mathrm{P}_{\mathbb{R}}$ the class of subsets of $\mathbb{R}^\infty$ that can be decided by an RTM in polynomial time.

It is important to note that if we again have a bound t on the running time of an RTM M, the number of accessed coordinates of the input, output, and state space is bounded by t. Therefore, we can find a finite-dimensional machine M_f that computes the same function (by replacing the moving operations by rational functions on $\mathbb{R}^t$). Thus, following the discussion above, we can associate with the pair (M, t) an algebraic computation tree $T_{M,t}$ that computes the same function and branching as discussed above. This will be central in the proof of Theorem 3.2.

In the following sections, we will also be concerned with parallel computations. Therefore, let us recall from [10] the definition of a computational model for parallelism in the real Turing machine setting together with the complexity class it defines when restricted to polynomial time.

DEFINITION 1. *An* algebraic circuit $\mathcal{C}$ *over* $\mathbb{R}$ *is a directed acyclic graph where each node has in-degree* 0, 1, *or* 2. *Nodes with in-degree* 0 *are labeled either as inputs or with elements of* $\mathbb{R}$. *(We shall call the latter constant nodes.) Nodes with in-degree* 2 *are labeled with* "+","−","∗", *or* "/". *Finally, nodes with in-degree* 1 *are of a unique kind and are called sign nodes. There is one node with out-degree* 0 *called an output node. In what follows, the nodes of a circuit will be called* gates.

We inductively associate with each gate a function of the input variables in the usual way. In particular, we shall refer to the function associated with the output gate as the function computed by the circuit. Note that sign gates return 1 if their input is greater than or equal to 0 and return 0 otherwise.

DEFINITION 2. *For an algebraic circuit* $\mathcal{C}$, *we define its* size *to be the number*

of gates in $\mathcal{C}$ and its depth *to be the length of the longest path from some input or constant gate to the output gate.*

DEFINITION 3. *Given an algebraic circuit $\mathcal{C}$, the* canonical encoding *of $\mathcal{C}$ is a sequence of 4-tuples of the form $(g, op, g_l, g_r) \in \mathbb{R}^4$, where g represents the gate label, op is the operation performed by the gate, g_l is the gate which provides the left input to g, and g_r provides its right input. By convention, g_l and g_r are 0 if gate g is an input gate, and g_r is 0 if gate g is a sign gate (whose input is then given by g_l) or a constant gate (the associated constant then being stored in g_l). Also, we shall suppose that the first n gates are the input gates and the last gate is the output gate.*

DEFINITION 4. *Let $\{\mathcal{C}_n\}_{n\in\mathbb{N}}$ be a family of algebraic circuits. We shall say that the family is* P-uniform *if there exists a real Turing machine M that generates the encoding of the ith gate of $\mathcal{C}_n$ with input (i, n) in time polynomial in n.*

DEFINITION 5. *We shall say that a set S can be decided in* parallel polynomial time ($S \in \mathrm{PAR}_{\mathbb{R}}$ *for short*) *when there is a P-uniform family of circuits $\{\mathcal{C}_n\}$ with depth polynomial in n and such that $\mathcal{C}_n$ computes the characteristic function of S restricted to inputs of size n.*

Remark 2. It is possible to define [10] parallel polynomial time in a different way, namely, by putting an exponential number of RTMs to work together with the same program and in polynomial time. One can prove, however, that this model defines the same class $\mathrm{PAR}_{\mathbb{R}}$ that we just introduced.

The definition above closely follows the one given in the classical seting (i.e., over $\{0, 1\}$) of parallel machines. In this case, one uses Boolean circuits instead of algebraic ones and the uniformity condition is provided by a classical Turing machine. A description of this computational model and the classes it defines can be found in [2]. A particular feature of the classical setting is that the class defined by polynomial parallel time requirements coincides with PSPACE, the class defined by polynomial space bounds. This is an old result of Borodin (see [7] or [2, Chapter 4]) that we shall use repeatedly in this paper.

3. Computing with binary inputs. The goal of this section is to prove that the Boolean part of $\mathrm{P}_{\mathbb{R}}$ is included in PSPACE/*poly*, i.e., that any subset of Σ^* that can be decided in polynomial time by a real Turing machine can be decided by a (classical) Turing machine in polynomial space using a polynomial advice. In the next section, we will prove a more general result, namely, a similar inclusion for parallel real Turing machines. However, because of clarity of exposition, we will first show the inclusion for the Boolean part of $\mathrm{P}_{\mathbb{R}}$.

We begin by recalling the definition of nonuniform classes as given in [19], which we extend to complexity classes over the reals.

DEFINITION 6. *Let $\mathcal{C} \subseteq \Sigma^*$ (resp. $\mathcal{C} \subseteq \mathbb{R}^\infty$) be a class of sets and $\mathcal{F}$ be any class of functions from $\mathbb{N}$ to Σ^* (resp. from $\mathbb{N}$ to $\mathbb{R}^\infty$). The class $\mathcal{C}/\mathcal{F}$ is defined to be the class of all subsets $B \subseteq \Sigma^*$ (resp. $B \subseteq \mathbb{R}^\infty$) for which there exists a set $A \in \mathcal{C}$ and a function $f \in \mathcal{F}$ such that $B = \{x : \langle x, f(|x|)\rangle \in A\}$.*

We will be interested mainly in the case $\mathcal{F} = poly$, the class of functions f such that for some polynomial p, we have $|f(n)| \le p(n)$ for each $n \in \mathbb{N}$. For the Boolean case, one can find the main properties and characterizations of classes like $\mathrm{P}/poly$ or PSPACE/*poly* (as well as of some other nonuniform complexity classes) in Chapter 5 of [1].

A final result that we want to recall before proceeding to the main theorem of this section is the following.

PROPOSITION 3.1 (see [17, Proposition 12]). *Let W be a real algebraic set*

defined by a set of m polynomials with integer coefficients bounded in absolute value by L. Then we can determine the dimension of W within parallel time $(n \log(md \log L))^{O(1)}$. □

THEOREM 3.2. *The inclusion* $\mathrm{BP}(\mathrm{P}_{\mathbb{R}}) \subseteq \mathrm{PSPACE}/poly$ *holds.*

Proof. Let M be an RTM working in polynomial time, say n^q, and let $\alpha_1, \ldots, \alpha_k$ be its real constants.

For any $n \in \mathbb{N}$, machine M has an associated algebraic computation tree $T_{M,n}$ with depth n^q and size bounded by 2^{n^q}. To each branching node i of this tree, there corresponds a rational function $h_i \in \mathbb{Q}(Z_1, \ldots, Z_k, X_1, \ldots, X_n)$ such that the branching is done according to whether the actual input $x \in \mathbb{R}^n$ satisfies $h_i(\alpha, x) \geq 0$ or $h_i(\alpha, x) < 0$ for $\alpha = (\alpha_1, \ldots, \alpha_k)$.

The idea of the proof is to find $\beta_1, \ldots, \beta_p \in \mathbb{R}_{\mathrm{alg}}$ such that for every $x \in \Sigma^n$, the path followed by x in the tree $\widetilde{T}_{M,n}$ obtained by replacing the constants α_j by β_j is the same as the path followed in $T_{M,n}$. This ensures that the tree $\widetilde{T}_{M,n}$ accepts the same subset of Σ^n as $T_{M,n}$. On the other hand, we will require some codification of the β's that allows us to perform the operations in $\widetilde{T}_{M,n}$ in PSPACE together with a short way of writing this codification that will make possible to give it as a polynomial advice.

Before obtaining a description of the β's, let us do a final modification on $T_{M,n}$ that was first used in [20]. Let I be the set of branching nodes of $T_{M,n}$. For every $i \in I$ and every $x \in \Sigma^n$, we consider the rational functions

$$g_{i,x} \in \mathbb{Q}(Z_1, \ldots, Z_k)$$

defined by $g_{i,x} = h_i(Z_1, \ldots, Z_k, x_1, \ldots, x_n)$ and the real numbers $\tau_{i,x} = g_{i,x}(\alpha_1, \ldots, \alpha_k)$ The accepted subset of Σ^n can be characterized by the set of signs

$$\sigma_{i,x} = \mathrm{sign}(\tau_{i,x}),$$

where $\mathrm{sign}(z)$ is 1 if $z > 0$, -1 if $z < 0$, and 0 otherwise. Now for some i and x, the element $\tau_{i,x}$ can be zero. However, since the set of values

$$\{\tau_{i,x} : i \in I, x \in \Sigma^n\}$$

is finite, there exists an $\varepsilon > 0$ such that all of the negative values in the above set are strictly smaller than $-\varepsilon$, and for this ε the following equivalences hold:

$$\begin{aligned} g_{i,x}(\alpha_1, \ldots, \alpha_k) \geq 0 \quad &\text{iff} \quad g_{i,x}(\alpha_1, \ldots, \alpha_k) + \varepsilon > 0, \\ g_{i,x}(\alpha_1, \ldots, \alpha_k) < 0 \quad &\text{iff} \quad g_{i,x}(\alpha_1, \ldots, \alpha_k) + \varepsilon < 0. \end{aligned}$$

Replacing the tests $g_i(X) \geq 0$ by $g_i(X) + \varepsilon \geq 0$, we thus have that the new computation tree (which has real constants $\alpha_1, \ldots, \alpha_k, \varepsilon$) satisfies the following property: for every $x \in \Sigma^n$, all of the test values are different from zero.

Assuming that the rational functions $g_{i,x}(Z_1, \ldots, Z_k)$ are polynomials (something that we can do by simply replacing $g_{i,x}$ by the product of its numerator and denominator), we can rephrase the above remarks in the following way: the elements $\alpha_1, \ldots, \alpha_k, \varepsilon$ satisfy a system of polynomial inequations of the form

$$g_{i,x}\sigma_{i,x} > 0 : \quad i \in I, \quad x \in \Sigma^n \tag{1}$$

and any other real numbers $\beta_1, \ldots, \beta_k, \xi$ satisfying this system will, when used as constants in the tree $T_{M,n}$, produce the same outcome for every $x \in \Sigma^n$.

We can now describe how to obtain such numbers.

First, we construct the g_0 and G of the preceding section for the set of polynomials $\{g_{i,x}^2 \in \mathbb{Z}[Z_1, \ldots, Z_k, Y] : i \in I, x \in \Sigma^n\}$. Then, applying Theorem 1.3, we deduce the existence of integer vectors $\vec{v} = (v_2, \ldots, v_k, v_{k+1})$ such that the set W described in the statement of Theorem 1.3 is finite and nonempty. Let $\vec{v^*}$ be the first such vector for the lexicographical ordering in $\mathbb{N}^k$ and let W^* be its corresponding set of solutions. We then have that any connected component of the semialgebraic set S given by system (1) contains a point of W^*. (Note that by squaring all the $g_{i,x}$'s, we ensure that each connected component of S is one connected component of the set V defined by the conditions $g_{i,x}^2 > 0$.) Thus we take $\beta_1, \ldots, \beta_k, \xi$ to be any point of W^* belonging to S, and we distinguish it among the other points of W^* by its position p for the lexicographical ordering in $\mathbb{R}^{k+1}$. Note that from the equations defining W^* and this p, we can code (cf. [18] or [25]) each coordinate of this point (see the complexity analysis below).

The following nonuniform parallel algorithm then decides the same language as M when restricted to binary inputs.

input$(a_1, \ldots, a_n)$
get the advice p corresponding to n
for all $x \in \Sigma^n$ **in parallel do** (s1)
  for all path γ **in parallel do**
    for all i branch node in γ **do**
      compute the polynomial $g_{i,x}$
    od
  od
od
compute g_0 and G (s2)
compute C (s3)
compute $\vec{v^*}$ (s4)
code the coordinates of the pth point β of (s5)
the set W^* given by $\vec{v^*}, C$, and G
simulate the computation of M over $a_1, \ldots, a_n$ (s6)
replacing the $\alpha_1, \ldots \alpha_k, \varepsilon$ by the point β coded in (s5)

Let us estimate the complexity of the algorithm above. As we have seen, the number of nodes of the tree $T_{M,n}$ is bounded by 2^{n^q}. Therefore, the number of polynomials $g_{i,x}$ is bounded by $2^{n^q}2^n = 2^{n^q+n}$. Each of these polynomials is computed by a straight-line program of length n^q, and thus we get again a bound of 2^{n^q} for their degrees and $2^{2^{n^q}}$ for the absolute value of their coefficients. The degree d_G of G is then bounded by $2^{n^q} \cdot 2^{n^q} = 2^{O(1)n^q}$ and the absolute value of its coefficients L_G is bounded by

$$(2^{2^{n^q}})^{2^{n^q}} = 2^{2^{2n^q}}$$

(and thus by 2^{2n^q} in bit length). We can then—according to Theorem 1.3—bound by

$$(2d_G)^{k+1} = 2^{O(1)n^q}$$

the integers $v_2, \ldots, v_k, v_{k+1}$ and by

$$(2d_G)^{k+1} = 2^{O(1)n^q}$$

the number of points in W^*. A first consequence of these last two upper bounds is the fact that the advice above has polynomial size.

Concerning the running time, it is clear that step (s1) can be done in polynomial time using an exponential number of processors because, given an $x \in \Sigma^n$ and a path γ, the—at most—n^q polynomials that appear in that path have exponential degree in a constant number ($k+1$, in fact) of variables and therefore an exponential number of monomials. Any arithmetical operation between two such polynomials can be done within these resources, and we have a polynomial number of such operations.

The product G is computed with a binary tree of products having polynomial depth. Since each product can be done in parallel polynomial time, the same applies for the whole tree and then for step (s2). A similar remark holds for the constant C and thus for step (s3).

The determination of $\vec{v^*}$ can be done by checking in parallel for all possible vectors $\vec{v}$ whether the dimension of the resulting W is zero and then selecting the first one that gives a positive answer. However, the determination of the dimension of each W can be done in parallel time bounded by $(kn)^{O(q)}$ by a direct application of Proposition 3.1. Thus the overall parallel time needed to compute $\vec{v^*}$ is bounded by $(kn)^{O(q)}$.

For step (s5), one can apply the algorithms given in [18] or [26]. However, we remark here that a cylindrical algebraic decomposition together with the coding *à la Thom* (see [8] for the algorithms and [27] and [12] for complexity analysis) suffices because the double-exponential behavior of this algorithm is only in the number of variables—which is constant in our case—and it is NC in the rest of the parameters. This results on a procedure for (s5) working in parallel polynomial time.

Finally, note that each arithmetical operation of M is translated in step (s6) into an operation of elements in $\mathbb{Z}[Z_1, \ldots, Z_k]$, and this is also done in parallel polynomial time. On the other hand, at each test of the form $g(Z_1, \ldots, Z_k) \geq 0$, we use the same algorithm of step (s5) for determining the sign of $g(Z_1, \ldots, Z_k)+Y$ on the point coded in (s5).

The above considerations show that the algorithm runs in parallel polynomial time. Since this is equivalent to polynomial space, we have shown that the set decided by the algorithm above belongs to PSPACE/*poly*. □

4. Binary inputs for parallel real Turing machines. Our next goal is to extend our previous result to the class $\mathrm{PAR}_{\mathbb{R}}$ of sets decided in parallel polynomial time. Before going into the next theorem, we will recall a result concerning the number of satisfiable sign conditions of a polynomial system.

LEMMA 4.1 (see [15, Lemma 1]). *Let $f_1, \ldots, f_s \in \mathbb{R}[X_1, \ldots, X_k]$ be a finite family of polynomials and $D = \sum_{i=1}^{s} \mathrm{degree}(f_i)$. Then the number of satisfiable systems of the form*

$$f_1(X_1, \ldots, X_k)\sigma_1 \ \& \cdots \& \ f_s(X_1, \ldots, X_k)\sigma_s,$$

where σ_i belongs to $\{\geq 0, > 0\}$ for $i = 1, \ldots, s$, is bounded by $D^{O(k)}$. □

THEOREM 4.2. *The equality* $\mathrm{BP}(\mathrm{PAR}_{\mathbb{R}}) = \mathrm{PSPACE}/\mathit{poly}$ *holds.*

Proof. Let S be a set in $\mathrm{PAR}_{\mathbb{R}}$ and $\{\mathcal{C}_n\}$ be the family of circuits deciding S. Also, let M be the RTM that generates these circuits and $\alpha_1, \ldots, \alpha_k$ be its real constants.

Given any $n \in \mathbb{N}$, we consider for any sign gate i of $\mathcal{C}_n$ and any binary string $x \in \Sigma^n$ the rational function $g_{i,x,\alpha} \in \mathbb{Q}(\alpha_1, \ldots, \alpha_k)(X_1, \ldots, X_n)$ that the gate receives as input. Note that besides the trivial dependence of the coefficients of $g_{i,x,\alpha}$ on α, there is a more subtle dependence on x since these coefficients also depend on the output of previous sign gates. Since the number of possible answers to previous sign gates is doubly exponential, we obtain an a priori doubly exponential number of

rational functions, and therefore we cannot directly apply the construction of Theorem 3.2. However, we can use Lemma 4.1 to reduce this number.

Let us fix $x \in \Sigma^n$ and plug x into the input gates of $\mathcal{C}_n$. This will force us to consider rational functions in $\mathbb{Q}(Z_1, \ldots, Z_k)$. Also, let n^q be a bound on the depth of $\mathcal{C}_n$. At depth 1, there are at most 2^{n^q} sign gates whose input functions have degree bounded by 1. By Lemma 4.1, the number of possible outputs of these sign gates is bounded by

$$(2^{n^q})^{O(k)} = 2^{O(k)n^q}.$$

For each set of outputs ω at depth 1, we consider the sign gates at depth 2. There are at most 2^{n^q-1} of them, and their associated functions have degree bounded by 2. Thus, again by Lemma 4.1, we bound by

$$(2^{n^q})^{O(k)} = 2^{O(k)n^q}$$

the number of possible outputs for ω. Multiplying both expressions, we deduce that the total number of possible outputs at depths 1 and 2 is bounded by

$$2^{2O(k)n^q}.$$

Inductively, we prove that the number of possible outputs over all of the sign gates is bounded by

$$2^{O(k)n^q n^q} = 2^{O(k)n^{2q}},$$

a number which is singly exponential in n.

Let us then consider for any $x \in \Sigma^n$ the set of all rational functions $g_{i,x,\omega}(Z_1, \ldots, Z_k)$ obtained by varying i over all sign gates of $\mathcal{C}_n$ and ω over all possible outputs of the set of sign gates. If we now consider this set for any $x \in \Sigma^n$, we will have that the set decided by the circuit $\mathcal{C}_n$ is determined by the signs that the functions in this set take when evaluated at $\alpha_1, \ldots, \alpha_k$.

As in Theorem 3.2, we can assume the functions $g_{i,x,\omega}$ to be polynomials and we can also assume that they do not vanish on $\alpha_1, \ldots, \alpha_k$ by adding a new real number ε.

Since the number of polynomials $g_{i,x,\omega}$ is singly exponential in n, we can apply the method of Theorem 3.2. However, note that the corresponding step (s1) will now be required to select for any $x \in \Sigma^n$ and any depth l the possible sign conditions for the test gates at depth l. This is done sequentially in l in order to avoid dealing with a doubly exponential number of sign conditions. Once these possible sign conditions are known, the rest of the algorithm works like the one in Theorem 3.2, simulating the circuit $\mathcal{C}_n$ instead of the tree. This shows that the binary elements of S are a language in PSPACE/*poly*.

On the other hand, the inclusion of PSPACE/*poly* in $\mathrm{BP}(\mathrm{PAR}_{\mathbb{R}})$ is trivial. □

An immediate corollary of Theorem 4.2 is the following separation, left open in [13]. Recall that $\mathrm{EXP}_{\mathrm{W}}$ is the class of subsets of $\mathbb{R}^\infty$ accepted by RTMs in weak exponential time, i.e., in exponential time but such that for all intermediately computed rational functions g $\deg(g)$ and the bit length of $|\mathrm{coeff}(g)|$ are exponentialy bounded (see [20] or [13] for a formal definition of the weak model).

COROLLARY 4.3. *The inclusion* $\mathrm{PAR}_{\mathbb{R}} \subset \mathrm{EXP}_{\mathrm{W}}$ *is strict.*

Proof. The Boolean part of $\mathrm{EXP}_{\mathrm{W}}$ is the class of all subsets of Σ^*. Therefore, it strictly contains PSPACE/*poly*. □

Remark 3. Corollary 4.3 improves the separation $\mathrm{PAR}_{\mathbb{IR}} \neq \mathrm{EXP}_{\mathbb{IR}}$ shown in [9]. In this latter case, the fact that a real Turing machine working in exponential time can produce polynomials of doubly exponential degree (while a circuit of polynomial depth cannot) together with an irreducibility argument sufficed to show the separation. The arguments used now are much more delicate and, somewhat surprisingly, pass through the Boolean part of these classes.

We can still improve Theorem 4.2 a bit by allowing the real machine to take advice.

THEOREM 4.4. *The equality* $\mathrm{BP}(\mathrm{PAR}_{\mathbb{IR}}/poly) = \mathrm{PSPACE}/poly$ *holds.*

Proof. The polynomial advice in $\mathrm{PAR}_{\mathbb{IR}}/poly$ introduces a polynomial number of real constants, say n^h, for each input size n. One can now simply check that replacing the constant value k in the proof of Theorem 4.2 by n^h does not affect the exponential character of the bounds there, and thus the same arguments apply. The only limitation is that in steps (s5) and (s6), one cannot use cylindrical algebraic decomposition (because of the exponential dependence that it has in the number of variables for its parallel running time) and is restricted to using the "faster" algorithms given in [18] and [25]. □

5. Conclusions and open problems. Theorems 4.2 and 4.4 are rather surprising since they show that multiplication or nonuniformity (under the form of a polynomial advice function) do not help in the presence of parallelism to decide binary sets. Note that results weaker than Theorem 4.2, namely, that the Boolean part of $\mathrm{PAR}_{\mathrm{add}}$ (where no multiplications are allowed) or of $\mathrm{PAR}_{\mathrm{W}}$ (where few multiplications are allowed) coincide both with PSPACE/*poly*, were proved in [11] and [13].

On the other hand, a main question that remains open is whether $\mathrm{BP}(\mathrm{P}_{\mathbb{IR}}) = \mathrm{PSPACE}/poly$. We know that this Boolean part contains $\mathrm{P}/poly$, but its exact power is still to be determined. Note that for integer RAMs, it is known that the computational power of this model in polynomial time is exactly PSPACE for several sets of primitive operations. However, in all of these cases, there is a primitive operation that cannot be efficiently simulated by a real Turing machine. Thus, for instance, it is shown in [4] that integer RAMs with operations $(+, -, *, \div)$ have the power of PSPACE. However, the simulation of integer division by a real Turing machine over integers of exponential length takes exponential time, and therefore the arguments of [4] cannot be used to show the inclusion $\mathrm{PSPACE}/poly \subseteq \mathrm{BP}(\mathrm{P}_{\mathbb{IR}})$.

Acknowledgment. Thanks are due to Pascal Koiran for pointing out to us the possibility of allowing advice in the real complexity classes that lead from Theorem 4.2 to Theorem 4.4.

REFERENCES

[1] J. L. BALCÁZAR, J. DÍAZ, AND J. GABARRÓ, *Structural Complexity* I, EATCS Monographs on Theoretical Computer Science 11, Springer-Verlag, Berlin, 1988.

[2] J. L. BALCÁZAR, J. DÍAZ, AND J. GABARRÓ, *Structural Complexity* II, EATCS Monographs on Theoretical Computer Science 22, Springer-Verlag, Berlin, 1990.

[3] M. BEN-OR, *Lower bounds for algebraic computation trees*, in Proc. 15th ACM Symposium on the Theory of Computing, ACM, New York, 1983, pp. 80–86.

[4] A. BERTONI, G. MAURI, AND N. SABADINI, *Simulations among classes of random access machines and equivalence among numbers succinctly represented*, Ann. Discrete Math., 25 (1985), pp. 65–90.

[5] L. Blum, M. Shub, and S. Smale, *On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines*, Bull. Amer. Math. Soc., 21 (1989), pp. 1–46.
[6] J. Bochnak, M. Coste, and M.-F. Roy, *Géométrie algébrique réelle*, Springer-Verlag, Berlin, 1987.
[7] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.
[8] M. Coste and M.-F. Roy, *Thom's lemma, the coding of real algebraic numbers and the topology of semi-algebraic sets*, J. Symbolic Comput., 5 (1988), pp. 121–129.
[9] F. Cucker, $\mathrm{P}_{\mathbb{R}} \neq \mathrm{NC}_{\mathbb{R}}$, J. Complexity, 8 (1992), pp. 230–238.
[10] F. Cucker, *On the complexity of quantifier elimination: The structural approach*, Comput. J., 36 (1993), pp. 400–408.
[11] F. Cucker and P. Koiran, *Computing over the reals with addition and order: Higher complexity classes*, J. Complexity, 11 (1995), pp. 358–376.
[12] F. Cucker, H. Lanneau, B. Mishra, P. Pedersen, and M.-F. Roy, *NC algorithms for real algebraic numbers*, Appl. Algebra Engrg. Comm. Comput., 3 (1992), pp. 79–98.
[13] F. Cucker, M. Shub, and S. Smale, *Complexity separations in Koiran's weak model*, Theoret. Comput. Sci., 133 (1994), pp. 3–14.
[14] J. B. Goode, *Accessible telephone directories*, J. Symbolic Logic, 59 (1994), pp. 92–105.
[15] D. Y. Grigoriev, *Complexity of deciding Tarski algebra*, J. Symbolic Comput., 5 (1988), pp. 65–108.
[16] D. Y. Grigoriev and N. N. Vorobjov, *Solving systems of polynomial inequalities in subexponential time*, J. Symbolic Comput., 5 (1988), pp. 37–64.
[17] J. Heintz, T. Krick, M.-F. Roy, and P. Solernó, *Geometric problems solvable in single exponential time*, in Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes: Proc. 8th International Conference, S. Sakata, ed., Lecture Notes in Comput. Sci. 508, Springer-Verlag, Berlin, 1991, pp. 11–23.
[18] J. Heintz, M.-F. Roy, and P. Solernó, *Sur la complexité du principe de Tarski–Seidenberg*, Bull. Soc. Math. France, 118 (1990), pp. 101–126.
[19] R. Karp and R. Lipton, *Turing machines that take advice*, Enseign. Math., 28 (1982), pp. 191–209.
[20] P. Koiran, *A weak version of the Blum, Shub and Smale model*, in Proc. 34th Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 486–495.
[21] P. Koiran, *Computing over the reals with addition and order*, Theoret. Comput. Sci., 133 (1994), pp. 35–47.
[22] P. Koiran, *A weak version of the Blum, Shub and Smale model*, Technical report 94-10, DIMACS, Rutgers University, Piscataway, NJ, 1994.
[23] W. Maass, *Bounds for the computational power and learning complexity of analog neural nets*, in Proc. 25th Symposium on the Theory of Computing, ACM, New York, 1993, pp. 335–344.
[24] J. Milnor, *Topology from the Differentiable Viewpoint*, University Press of Virginia, Charlottesville, VA, 1965.
[25] J. Renegar, *On the computational complexity and geometry of the first-order theory of the reals, part* III, J. Symbolic Comput., 13 (1992), pp. 329–352.
[26] J. Renegar, *On the computational complexity and geometry of the first-order theory of the reals, part* I, J. Symbolic Comput., 13 (1992), pp. 255–299.
[27] M.-F. Roy and A. Szpirglas, *Complexity of computation on real algebraic numbers*, J. Symbolic Comput., 7 (1990), pp. 39–51.
[28] H. T. Siegelmann and E. D. Sontag, *On the computational power of neural nets*, in Proc. 5th ACM Workshop on Computational Learning Theory, ACM, New York, 1992, pp. 440–449.
[29] H. T. Siegelmann and E. D. Sontag, *Analog computation via neural networks*, Theoret. Comput. Sci., 131 (1994), pp. 331–360.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 255–272, February 1997

015

AN $\mathcal{NC}$ ALGORITHM FOR MINIMUM CUTS*

DAVID R. KARGER† AND RAJEEV MOTWANI‡

Abstract. We show that the minimum-cut problem for weighted undirected graphs can be solved in $\mathcal{NC}$ using three separate and independently interesting results. The first is an (m^2/n)-processor $\mathcal{NC}$ algorithm for finding a $(2+\epsilon)$-approximation to the minimum cut. The second is a randomized reduction from the minimum-cut problem to the problem of obtaining a $(2+\epsilon)$-approximation to the minimum cut. This reduction involves a natural combinatorial *set-isolation problem* that can be solved easily in $\mathcal{RNC}$. The third result is a derandomization of this $\mathcal{RNC}$ solution that requires a combination of two widely used tools: pairwise independence and random walks on expanders. We believe that the set-isolation approach will prove useful in other derandomization problems.

The techniques extend to two related problems: we describe $\mathcal{NC}$ algorithms finding minimum k-way cuts for any constant k and finding all cuts of value within any constant factor of the minimum. Another application of these techniques yields an $\mathcal{NC}$ algorithm for finding a *sparse k-connectivity certificate* for all polynomially bounded values of k. Previously, an $\mathcal{NC}$ construction was only known for polylogarithmic values of k.

Key words. randomized algorithms, derandomization, parallel algorithms, minimum cut, multiway cut, edge connectivity, connectivity certificate

AMS subject classifications. 68Q22, 68Q25

PII. S0097539794273083

1. Introduction.

Some of the central open problems in the area of parallel algorithms are those of devising $\mathcal{NC}$ algorithms for s-t minimum cuts and maximum flows, maximum matchings, and depth-first search trees. There are $\mathcal{RNC}$ algorithms for all of these problems [1, 24, 29]. The problem of finding *global* minimum cuts belongs to this category of unsolved derandomization problems, and it is representative in that obtaining an $\mathcal{NC}$ algorithm for the case of *directed* graphs would resolve the other derandomization questions [18]. We take a (possibly small) step towards resolving these open problems by presenting the first $\mathcal{NC}$ algorithm for the min-cut problem in *weighted undirected* graphs. Our results extend to minimum multiway cuts and to the problem of enumerating all approximately minimal cuts.

The *min-cut problem* is defined as follows: given a multigraph with n vertices and m (possibly weighted) edges, we wish to partition the vertices into two nonempty sets S and T so as to minimize the number of edges crossing from S to T (if the graph is weighted, we wish to minimize the total weight of crossing edges). We distinguish the minimum-cut problem from the s-t minimum-cut problem, where we require that two specified vertices s and t be on opposite sides of the cut; in the minimum-cut

* Received by the editors August 22, 1994; accepted for publication (in revised form) April 28, 1995. A preliminary version of this paper appeared as an extended abstract in *Proc. 25th ACM Symposium on Theory of Computing*, ACM, New York, 1993, pp. 487–506 [22].

http://www.siam.org/journals/sicomp/26-1/27308.html

† Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02138 (karger@lcs.mit.edu, http://theory.lcs.mit.edu/∼karger). Part of this work was done while this author was at Stanford University and supported by a National Science Foundation Graduate Fellowship, NSF grants CCR-9010517 and CCR-9357849, and Mitsubishi.

‡ Department of Computer Science, Stanford University, Stanford, CA 94305 (motwani@cs.stanford.edu, http://theory.stanford.edu/∼motwani). The research of this author was supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Development Award, grants from Mitsubishi and the OTL, NSF grant CCR-9010517, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, the Schlumberger Foundation, the Shell Foundation, and Xerox Corporation.

problem there is no such restriction. Our work deals only with minimum cuts. We assume that the graph is connected since otherwise the problem is trivial. The value of a minimum cut in an unweighted graph is also called the graph's *edge connectivity*.

The min-cut problem has numerous applications in many fields. The problem of determining the connectivity of a network arises frequently in the study of network design and network reliability [9]. (Recently, Karger [21] has shown that enumerating all nearly minimum cuts is the key to a fully polynomial-time approximation scheme for the all-terminal network-reliability problem.) Picard and Queyranne [32] survey many other applications of weighted minimum cuts. In information retrieval, minimum cuts have been used to identify clusters of topically related documents in hypertext systems [5]. Padberg and Rinaldi [31] discovered that the solution of minimum-cut problems was the computational bottleneck in cutting-plane-based algorithms for the traveling-salesman problem and many other combinatorial problems whose solutions induce connected graphs. Applegate [3] also observed that a faster algorithm for finding all minimum cuts might accelerate the solution of traveling-salesman problems.

The approach we take is typical of derandomization techniques that treat random bits as a resource. We develop a randomized algorithm and then show that it can be made to work using few random bits. If we can reduce the number of bits the algorithm needs to examine for a size-n problem to $O(\log n)$ without affecting its probability of correctness, then we know that it runs correctly on at least some of these small random inputs. Therefore, by trying all $n^{O(1)}$ possible $O(\log n)$-bit random inputs, we ensure that we will run correctly at least once and thus find the correct answer (this requires that we can check which of our many answers is correct, but here that just involves comparing the values of the cuts that are found).

The $\mathcal{NC}$ algorithm we devise is clearly impractical; it serves to demonstrate the existence of a deterministic parallel algorithm rather than to indicate what the "right" such algorithm is.

A preliminary version of this paper has appeared earlier as an extended abstract [22]. A more extensive version of this article and its context can be found in the first author's dissertation [19].

1.1. Previous work. The first minimum-cut algorithms used the duality between s-t minimum cuts and maximum flows [11, 12]. An s-t max-flow algorithm can be used to find an s-t minimum cut, and by taking the minimum over all $\binom{n}{2}$ possible choices of s and t, a minimum cut may be found. Until recently, the best sequential algorithms for finding minimum cuts used this approach [15]. Parallel solutions to the min-cut problem have also been studied. Goldschlager, Shaw, and Staples [14] showed that the s-t min-cut problem on weighted directed graphs is $\mathcal{P}$-complete. A simple reduction [18, 19] shows that the (unrestricted) min-cut problem is also $\mathcal{P}$-complete in such graphs.

For unweighted graphs, any $\mathcal{RNC}$ matching algorithm can be combined with a well-known reduction of s-t maximum flows to matching [24] to yield $\mathcal{RNC}$ algorithms for s-t minimum cuts. By performing n of these computations in parallel, we can solve the min-cut problem in $\mathcal{RNC}$. For an input graph with n vertices and m edges, the $\mathcal{RNC}$ matching algorithm of Karp, Upfal, and Wigderson [24] runs in $O(\log^3 n)$ time using $O(n^{6.5})$ processors, while the one due to Mulmuley, Vazirani, and Vazirani [29] runs in $O(\log^2 n)$ time using $O(n^{3.5}m)$ processors. The processor bounds are quite large, and the technique does not extend to graphs with large edge weights. No deterministic parallel algorithm is known. Indeed, derandomizing max-flow on

unweighted, undirected graphs is equivalent to derandomizing maximum bipartite matching—a problem that has long been open. A reduction in [18, 19] shows that the global min-cut problem for *directed* graphs is also equivalent.

1.2. Contraction-based algorithms. Recently, a new paradigm has emerged for finding minimum cuts in undirected graphs [18, 23, 30]. This approach is based on *contracting* graph edges. Given a graph G and an edge (u, v), contracting (u, v) means replacing u and v with a new vertex w and transforming each edge (x, u) or (x, v) into a new edge (x, w). Any (u, v) edge turns into a self loop on w and can be discarded.

A key fact is that contracting edges cannot decrease the minimum cut. The reason is that any cut in the contracted graph corresponds to a cut of exactly the same value in the original graph—if u and v were contracted to w, then a vertex partition (A, B) in the contracted graph with $w \in A$ corresponds to a partition $(A \cup \{u, v\} - \{w\}, B)$ in the original graph that cuts the same edges. Let us fix a particular minimum cut, which from now on we will refer to as *the* minimum cut (there may be as many as $\binom{n}{2}$ [10, 18]). The power of contractions comes from their interaction with cuts. If we contract an edge that is not in the minimum cut, then the minimum cut in the contracted graph is equal to the minimum cut in the original graph.

Several contraction-based minimum-cut algorithms have recently been developed. They all work by contracting non-min-cut edges until the graph has been reduced to two vertices. These two vertices define a cut in the original graph. If no min-cut edge is contracted, then the corresponding cut must be a minimum cut. The edges connecting the two vertices correspond to the cut edges.

Nagamochi and Ibaraki [30] used graph contraction to develop an $O(mn+n^2 \log^2 n)$-time algorithm for the min-cut problem. In $O(m + n \log n)$ time, they find a *sparse connectivity certificate* (i.e., a subgraph that contains all the min-cut edges) that excludes some edge of the graph. This edge can be contracted without affecting the minimum cut. Constructing a sparse certificate to identify an edge to contract requires $O(m + n \log n)$ time and must be done n times; thus the running time. Matula [27] used the Nagamochi–Ibaraki certificate algorithm in a linear-time algorithm for finding a $(2 + \epsilon)$-approximation to the minimum cuts—the change is to use the sparse certificate to identify a large number of edges that can be contracted simultaneously.

Karger [18] observed that a randomly selected graph edge is unlikely to be in the minimum cut; it followed that repeated random selection and contraction of graph edges could be used to find a minimum cut. This led to the contraction algorithm, the first $\mathcal{RNC}$ algorithm for the weighted min-cut problem, which used mn^2 processors. Karger and Stein [23] improved the processor cost of the contraction algorithm, as well as its sequential running time, to $\tilde{O}(n^2)$; this is presently the most efficient known min-cut algorithm for weighted graphs.[1] A side effect of the analysis of [18] was a bound on the number of *approximately minimal* cuts in a graph; this plays an important role in our analysis.

Luby, Naor, and Naor [26] observed that in the contraction algorithm, it is not necessary to choose edges randomly one at a time. Instead, given that the min-cut size is c, they randomly mark each edge with probability $1/c$ and contract all of the marked edges. With constant probability, no min-cut edge is marked while the number of graph vertices is reduced by a constant factor. Thus after $O(\log n)$ phases of contraction, the graph is reduced to two vertices that define a cut. Since

[1] The notation $\tilde{O}(f(n))$ denotes $O(f(n) \operatorname{polylog} n)$.

the number of phases is $O(\log n)$ and there is a constant probability of missing the minimum cut in each phase, there is an $n^{-O(1)}$ probability that no min-cut edge is ever contracted; if this happens, then the cut determined at the end is the minimum cut. Observing that pairwise-independent marking of edges can be used to achieve the desired behavior, they show that $O(\log n)$ random bits suffice to run a phase. Thus $O(\log^2 n)$ bits suffice to run this modified contraction algorithm through its $O(\log n)$ phases.

Unfortunately, this algorithm cannot be fully derandomized. It is indeed possible to try all (polynomially many) random seeds for a phase and be sure that one of the outcomes is good (i.e., contracts nonmin-cut edges incident on a constant fraction of the vertices); however, there is no way to determine *which* outcome is good. In the next phase, it is thus necessary to try all possible random seeds on each of the polynomially many outcomes of the first phase, squaring the number of outcomes after two phases. In all, $\Omega(n^{\log n})$ combinations of seeds must be tried to ensure that we find the desired sequence of good outcomes leading to a minimum cut.

1.3. Overview of results. Our main result is an $\mathcal{NC}$ algorithm for the min-cut problem. Our algorithm is not a derandomization of the contraction algorithm but is instead a new contraction-based algorithm. Throughout, we take G to be a multigraph with n vertices, m edges, and min-cut value c. Most of the paper discusses unweighted graphs; in section 6.4, we reduce the weighted graph problem to the unweighted graph problem. Our algorithm extends to finding *minimum multiway cuts* that partition the graph into $r \geq 2$ disconnected pieces.

Our algorithm depends upon three major building blocks. The first building block (sections 2 and 3) is an $\mathcal{NC}$ algorithm that uses m^2/n processors to find a $(2+\epsilon)$-approximation to the minimum cut. Recall that Matula's sequential algorithm [27] was based on the sequential sparse certificate algorithm of Nagamochi and Ibaraki [30] (discussed in the previous section). It repeatedly finds a sparse certificate containing all min-cut edges and then contracts the edges not in the certificate, terminating after a small number of iterations. Our $\mathcal{NC}$ algorithm uses a new parallel sparse certificate algorithm to parallelize Matula's algorithm. A parallel sparse k-connectivity certificate algorithm with running time $\tilde{O}(k)$ was given by Cheriyan, Kao, and Thurimella [6]; we improve this in a necessary way by presenting an algorithm that runs in $O(\log m)$ time using km processors and is thus in $\mathcal{NC}$ for all $k = n^{O(1)}$.

Our next building block (section 4) uses a result obtained from the analysis of the contraction algorithm. Karger [18] proved that there are only polynomially many cuts whose size is within a constant factor of the minimum cut. If we find a collection of edges that contains one edge from every such cut except for the minimum cut, then contracting this set of edges yields a graph with no small cut except for the minimum cut. We can then apply the $\mathcal{NC}$ approximation algorithm mentioned in the previous paragraph. Since the minimum cut will be the unique contracted-graph cut within the approximation bounds, it will be found by the approximation algorithm. One can view this approach as a variant on the isolating lemma approach used to solve the perfect matching problem [29]. As was the case there, the problem is relatively easy to solve if the solution is unique, so the goal is to destroy all but one solution to the problem and then to easily find the unique solution.

Randomization yields a simple solution to this problem: contract each edge independently with probability $\Theta(\log n/c)$. Because the number of small cuts is polynomially bounded, there is a sufficient (larger than one over a polynomial) probability

that no edge from the minimum cut is contracted but one edge from every other small cut is contracted. Of course, our goal is to do away with randomization.

A step towards this approach is a modification of the Luby, Naor, and Naor technique. If we contract each edge with probability $\Theta(1/c)$, then with constant probability we contract no min-cut edge while contracting edges in a constant fraction of the other small cuts. Pairwise independence in the contracting of edges is sufficient to make such an outcome likely. However, this approach seems to contain the same flaw as before: $\Omega(\log n)$ phases of selection are needed to contract edges in all the small cuts, and thus $\Omega(\log^2 n)$ random bits are needed.

We work around this problem with our third building block (section 5). The problem of finding a good set of edges to contract can be formulated abstractly as the *set-isolation problem*: given an unknown collection of sets (the cuts) over a known universe, with one of the unknown sets declared "safe," find a collection of elements that intersects every set except for the safe one. After giving a simple randomized solution, we show that this problem can be solved in $\mathcal{NC}$ by combining the techniques of pairwise independence [7, 25] with the technique of random walks on expanders [2]. We feel that this combination should have further application in derandomizing algorithms; similar ideas were used previously to save random bits, e.g., in the work of Bellare, Goldreich, and Goldwasser [4].

Finally, in section 6, we apply the above results to finding minimum cuts, minimum multiway cuts, and weighted minimum cuts and to enumerating approximately minimum cuts.

2. An approximation algorithm. In the next two sections, we describe an $\mathcal{NC}$ algorithm that, for any constant $\epsilon > 0$, finds a cut whose value is less than $(2+\epsilon)$ times that of the minimum cut. We use the fact that contracting a non-min-cut edge does not change the value of the minimum cut. We first formalize this notion of contraction. To contract an edge (v_1, v_2), we replace both endpoints by a vertex v and let the set of edges incident on v be the union of the sets of edges incident on v_1 and v_2. We do not merge edges from v_1 and v_2 that have the same other endpoint; instead, we create multiple instances of those edges. However, we remove self loops formed by edges originally connecting v_1 to v_2. Formally, we delete all edges (v_1, v_2) and replace each edge (v_1, w) or (v_2, w) with an edge (v, w). The rest of the graph remains unchanged.

Since contracting a non-min-cut edge does not affect the minimum cut, we can find the minimum cut by repeatedly finding and contracting non-min-cut edges. Each time we do this, the number of graph vertices decreases by one; thus after $n-2$ iterations, we will have a two-vertex graph with an obvious minimum cut. The need to find non-min-cut edges motivates the following definition and lemma.

DEFINITION 2.1. *A* k-jungle *is a set of* k *disjoint forests in* G. *A* maximal k-jungle *is a* k*-jungle such that no other edge in* G *can be added to any one of the jungle's forests without creating a cycle in that forest.*

LEMMA 2.2 (see [30]). *A maximal* k*-jungle contains all the edges in any cut of* k *or fewer edges.*

Proof. Consider a maximal k-jungle J, and suppose it contains fewer than k edges of some cut C. Then some forest F in J must have no edge crossing C. Now suppose some edge e from C is not in the forest (if all cut-edges are in the forest, we are done). There is no path in F connecting the endpoints of e since such a path would have to cross C. Thus e can be added to F, contradicting the maximality of J. Thus all edges in C must already be in J. □

Nagamochi and Ibaraki [30] gave an algorithm for constructing a k-jungle that excluded one non-min-cut edge which could then be contracted. This led to an algorithm with running time $O(mn)$. Subsequently, Matula [27] observed that if we were willing to settle for a $(2+\epsilon)$-approximation to the minimum cut, we could construct a k-jungle, $k > c$, that excluded many edges, all of which could then be contracted in one step. This allows much faster progress towards a two-vertex graph, leading to a linear-time min-cut algorithm. We show how this algorithm can be parallelized.

The approximation algorithm is described in Figure 1. We give it as an algorithm to approximate the cut value; it is easily modified to find a cut with the returned value. The basic idea is to consider the minimum graph degree δ as an approximation to the minimum cut c. Clearly, $c < \delta$. If $(2+\epsilon)c > \delta$, then our approximation is good enough. Otherwise, we will see that a k-jungle that excludes many edges can be constructed with $k > c$.

PROCEDURE APPROX-MIN-CUT(multigraph G)
1. Let δ be the minimum degree of G.
2. Let $k = \delta/(2+\epsilon)$.
3. Find a maximal k-jungle.
4. Construct G' from G by contracting all nonjungle edges.
5. Return $\min(\delta, \text{Approx-Min-Cut}(G'))$.

FIG. 1. *The approximation algorithm*

LEMMA 2.3. *Given a graph with minimum cut c, the approximation algorithm returns a value between c and $(2+\epsilon)c$.*

Proof. Clearly, the value is at least c because it corresponds to some cut the algorithm encounters. That is, the minimum-degree vertex in a contracted intermediate graph corresponds to a cut of the same value in the original graph. For the upper bound, we use induction on the size of G. We consider two cases. If $\delta < (2+\epsilon)c$, then since we return a value of at most δ, the algorithm is correct. On the other hand, if $\delta \geq (2+\epsilon)c$, then $k \geq c$. It follows from Lemma 2.2 that the jungle we construct contains all the min-cut edges. Thus no edge in the minimum cut is contracted while forming G', so G' has minimum cut c. By the inductive hypothesis, the recursive call returns a value between c and $(2+\epsilon)c$. □

LEMMA 2.4. *There are $O(\log m)$ levels of recursion in the approximation algorithm.*

Proof. If G has minimum degree δ, then summing over vertices, G must have at least δn edge-endpoints and thus at least $\delta n/2$ edges. On the other hand, the graph G' that we construct contains only jungle edges; since each forest of the jungle contains only $n-1$ edges, G' can have at most $k(n-1) = \delta(n-1)/(2+\epsilon)$ edges. It follows that each recursive step reduces the number of edges in the graph by a constant factor; thus at a recursion depth of $O(\log m)$, the problem can be solved trivially. □

Note that the extra ϵ factor above 2 is needed to ensure a significant reduction in the number of edges at each stage and thus keep the recursion depth small. The depth of recursion is in fact $\Theta(\epsilon^{-1} \log m)$.

Each step of this algorithm, except for step 3, can be implemented in $\mathcal{NC}$ using m processors. Since the number of iterations is $O(\log m)$, the running time of this algorithm is $O(T(m,n) \operatorname{polylog} m)$, where $T(m,n)$ is the time needed to construct a maximal jungle. It only remains to show how to construct a maximal k-jungle in $\mathcal{NC}$.

3. Finding maximal jungles. The notation needed to describe this construction is somewhat complex, so first we give some intuition. To construct a maximal jungle, we begin with an empty jungle and repeatedly *augment* it by adding additional edges from the graph until no further augmentation is possible. Consider one of the forests in the jungle. The nonjungle edges that may be added to that forest without creating a cycle are just the edges that cross between two different trees of that forest. We let each tree claim some such edge incident upon it. Hopefully, each forest will claim and receive a large number of edges, thus significantly increasing the number of edges in the jungle.

Two problems arise. The first is that several trees may claim a particular edge. However, the arbitration of these claims can be transformed into a matching problem and solved in $\mathcal{NC}$. Another problem is that since each tree is claiming an edge, a cycle might be formed when the claimed edges are added to the forest—for example, two trees may each claim an edge connecting those two trees. We will remedy this problem as well.

3.1. Augmentations.

DEFINITION 3.1. *An* augmentation *of a k-jungle $J = \{F_1, \ldots, F_k\}$ is a collection $A = \{E_1, \ldots, E_k\}$ of k disjoint sets of nonjungle edges from G. At least one of the sets E_i must be nonempty. The edges of E_i are added to forest F_i.*

DEFINITION 3.2. *A* valid augmentation *of J is one that does not create any cycles in any of the forests of J.*

FACT 3.3. *A jungle is maximal if and only if it has no valid augmentation.*

Given a jungle, it is convenient to view it in the following fashion. We construct a *reduced (multi)graph* G_F for each forest F. For each tree T in F, the reduced graph contains a *reduced vertex* v_T. For each edge e in G that connects trees T and U, we add an edge e_F connecting v_T and v_U. Thus the reduced graph is what we get if we start with G and contract all the forest edges. Since many edges can connect two forests, the reduced graph may have parallel edges. An edge e of G may induce many different edges, one in each forest's reduced graph.

Given any augmentation, the edges added to forest F can be mapped to their corresponding edges in G_F, inducing an *augmentation subgraph* of the reduced graph G_F.

FACT 3.4. *An augmentation is valid if and only if the augmentation subgraph it induces in each forest's reduced graph is a forest.*

Care should be taken not to confuse the forest F with the forest that is the augmentation subgraph of G_F.

3.2. The augmentation algorithm. Our construction proceeds in a series of $O(\log m)$ phases in which we add edges to the jungle J. In each phase, we find a valid augmentation of J whose size is a constant fraction of the largest possible valid augmentation. Since we reduce the maximum possible number of edges that can be added to J by a constant fraction each time, and since at the beginning the maximum number of edges we can add is at most m, J will have to be maximal after $O(\log m)$ phases.

To find a large valid augmentation, we solve a maximal matching problem on a bipartite graph H. Let one vertex set of H consist of the vertices v_T in the various reduced multigraphs, i.e., the trees in the jungle. Let the other vertex set consist of one vertex v_e for each nonjungle edge e in G. Connect each reduced vertex v_T of G_F to v_e if e_F is incident on v_T in G_F. Equivalently, we are connecting each tree in the jungle to the edges incident upon it in G. Note that this means each edge in G_F is a

valid augmenting edge for F. To bound the size of H, note that each vertex v_e will have at most $2k$ incident reduced-graph edges because it will be incident on at most two trees of each forest. Thus the total number of edges in H is $O(km)$.

LEMMA 3.5. *A valid augmentation of J induces a matching in H of the same size.*

Proof. Consider a valid augmentation of the jungle. We set up a corresponding matching in H between the edges of the augmentation and the reduced vertices as follows. For each forest F in J, consider its reduced multigraph G_F. Since the augmentation is valid, the augmenting edges in G_F form a forest (Fact 3.4). Root each tree in this forest arbitrarily. Each nonroot reduced vertex v_T has a unique augmentation edge e_F leading to its parent. Since edge e is added to F, no other forest F' will use edge $e_{F'}$, so we can match v_T to v_e. It follows that every augmentation edge is matched to a unique reduced vertex. □

LEMMA 3.6. *Given a matching in H, a valid augmentation of J of size at least half the size of the matching can be constructed in $\mathcal{NC}$.*

Proof. If edge $e \in G$ is matched to reduced vertex $v_T \in G_F$, tentatively assign e to forest F. Consider the set A of edges in G_F that correspond to the G-edges assigned to F. The edges of A may induce cycles in G_F, which would mean (Fact 3.4) that A does not correspond to a valid augmentation of F. However, if we find an acyclic subset of A, then the G-edges corresponding to this subset will form a valid augmentation of F.

To find this subset, arbitrarily number the vertices in the reduced graph G_F. Direct each edge in A away from the reduced vertex to which it was matched (so each vertex has out-degree one) and split the edges into two groups: $A_0 \subseteq A$ are the edges directed from a smaller-numbered to a larger-numbered vertex, and $A_1 \subseteq A$ are the edges directed from a larger-numbered to a smaller-numbered vertex. One of these sets, say A_0, contains at least half the edges of A. However, A_0 creates no cycles in the reduced multigraph. Its (directed) edges can form no cycle obeying the edge directions since such a cycle must contain an edge directed from a larger-numbered to a smaller-numbered vertex. On the other hand, any cycle disobeying the edge directions must contain a vertex with out-degree two, an impossibility. It follows that the edges of A_0 form a valid augmentation of F of at least half the size of the matching.

If we apply this construction to each forest F in parallel, we get a valid augmentation of the jungle. Furthermore, each forest will gain at least half the edges assigned to it in the matching, so the augmentation has the desired size. □

THEOREM 3.7. *Given G and k, a maximal k-jungle of G can be found in $\mathcal{NC}$ using $O(km)$ processors.*

Proof. We begin with an empty jungle and repeatedly augment it. Given the current jungle J, construct the bipartite graph H as previously described and use it to find an augmentation. Let a be the size of a maximum augmentation of J. Lemma 3.5 shows that H must have a matching of size a. It follows that any maximal matching in H must have size at least $a/2$, since at least one endpoint of each edge in any maximum matching must be matched in any maximal matching. Several $\mathcal{NC}$ algorithms for maximal matching exist—for example, that of Israeli and Shiloach [17]. Lemma 3.6 shows that after we find a maximal matching, we can (in $\mathcal{NC}$) transform this matching into an augmentation of size at least $a/4$. If we add these augmentation edges, the resulting graph has a maximum augmentation of at most $3a/4$ (since it can be combined with the previous size-$a/4$ augmentation to get an augmentation of the

starting graph). Since we reduce the maximum augmentation by 3/4 each time, and since the maximum jungle size is m, the number of augmentations needed to make a J maximal is $O(\log m)$. Since each augmentation is found in $\mathcal{NC}$, the maximal jungle can be found in $\mathcal{NC}$.

The processor cost of this algorithm is dominated by that of finding the matching in the graph H. The algorithm of Israeli and Shiloach requires a linear number of processors and is run on a graph of size $O(km)$. □

COROLLARY 3.8. *A $(2+\epsilon)$-approximate minimum cut can be found in $\mathcal{NC}$ using m^2/n processors.*

Proof. A graph with m edges has a vertex with degree $O(m/n)$; the minimum cut can therefore be no larger. It follows that our approximation algorithm will construct k-jungles with $k = O(m/n)$. □

4. Reducing to approximation. In this section, we show how the problem of finding a minimum cut in a graph can be reduced to that of finding a 3-approximation.[2] Our technique is to "kill" all cuts of size less than $3c$ other than the minimum cut itself. The minimum cut is then the only cut of size less than $3c$, and thus it must be the output of the $(2+\epsilon)$-approximation algorithm of section 2 if we run it with $\epsilon = 1$. To implement this idea, we focus on a particular minimum cut that partitions the vertices of G into two sets A and B. Consider the graphs induced by A and B.

LEMMA 4.1. *The minimum cuts in A and in B have value at least $c/2$.*

Proof. Suppose A has a cut into X and Y of value less than $c/2$. Only c edges go from $A = X \cup Y$ to B, so one of X or Y (say X) must have at most $c/2$ edges leading to B. Since X also has less than $c/2$ edges leading to Y, the cut $(X, \overline{X})$ has value less than c, a contradiction. □

THEOREM 4.2 (see [18]). *There are $O(n^{2\alpha})$ cuts of value at most α times the minimum.*

Combining Lemma 4.1 and Theorem 4.2, it follows that in each of A and B, every cut has value at least $c/2$ and there are $O(n^6)$ cuts of value less than $3c$. Note that these are *not* the small cuts in G but rather those in the graphs induced by A and B. Call these cuts the *target cuts*.

LEMMA 4.3. *Let Y be a set containing edges from every target cut but not the minimum cut. If every edge in Y is contracted, then the contracted graph has a unique cut of weight less than $3c$—the one corresponding to the original minimum cut.*

Proof. Clearly, contracting the edges of Y does not affect the minimum cut. Now suppose this contracted graph had some other cut C of value less than $3c$. It corresponds to some cut of the same value in the original graph. Since it is not the minimum cut, it must induce a cut in either A or B, and this induced cut must also have value less than $3c$. This induced cut is then a target cut, so one of its edges will have been contracted. However, this prevents C from being a cut in the contracted graph, a contradiction. □

It follows that after contracting Y, running the approximation algorithm of section 2 on the contracted graph will reveal the minimum cut since the actual minimum cut is the only one that is small enough to meet the approximation criterion. Our goal is thus to find a collection of edges that intersects every target cut but not the minimum cut. This problem can be phrased more abstractly as follows: Over some universe U, an adversary selects a polynomially sized collection of "target" sets of

[2] We reduce to 3-approximation for simplicity. Should this approach ever become practical, it will most likely be more efficient to reduce to $(2+\epsilon)$-approximation for some smaller ϵ.

roughly equal size (the small cuts' edge sets), together with a disjoint "safe" set of about the same size (the min-cut edges). We want to find a collection of elements that intersects every target set but not the safe set. Note that we do not know what the target or safe sets are, but we do have an upper bound on the number of target sets. We proceed to formalize this problem as the *set-isolation problem.*

5. The set-isolation problem. We describe a general form of the set-isolation-problem. Fix a universe $U = \{1, \ldots, u\}$ of size u.

DEFINITION 5.1. *A* (u, k, α) set-isolation instance *consists of a* safe set $S \subseteq U$ *and a collection of* k target sets $T_1, \ldots, T_k \subseteq U$ *such that*

- $\alpha > 0$ *is a constant,*
- *for* $1 \leq i \leq k$, $|T_i| \geq \alpha |S|$, *and*
- *for* $1 \leq i \leq k$, $T_i \cap S = \emptyset$.

We will use the notation that $s = |S|$, $t_i = |T_i|$, and $t = \alpha s \leq t_i$. It is important to keep in mind that the value of s is not specified in a set-isolation instance but, as will become clear shortly, it is reasonable to assume that it is known explicitly. Finally, while the safe set S is disjoint from all of the target sets, the target sets may intersect each other.

DEFINITION 5.2. *An* isolator *for a set-isolation instance is a set that intersects all of the target sets but not the safe set.*

An isolator is easy to compute (even in parallel) for any given set-isolation instance provided the sets $S, T_1, \ldots, T_k$ are explicitly specified. However, our goal is to find an isolator in the setting where only u, k, and α are known but the actual sets S, T_1, $\ldots, T_k$ are not specified. We can formulate this as the problem of finding a universal isolating family.

DEFINITION 5.3. *A* (u, k, α)-universal isolating family *is a collection of subsets of* U *that contains an isolator for any* (u, k, α) *set-isolation instance.*

To see that this general formulation captures our cut-isolation problem, note that the minimum cut is the safe set in an (m, k, α) set-isolation instance. The universe is the set of edges and is of size m; the target sets are the small cuts of the two sides of the minimum cut; k is the number of such small cuts and (by Lemmas 4.1 and 4.2) can be bounded by a polynomial in $n < m$; and $\alpha = 1/2$ since each target cut has size at least $c/2$ (by Lemma 4.1). The safe-set size s is the min-cut size c.

If we had an (m, k, α)-universal isolating family, then one of the sets in it would be an isolator for the set-isolation instance corresponding to the minimum cut. By Lemma 4.3, contracting all of the edges in this set would isolate the minimum cut as the only small cut. If the size of the universal family were polynomial in m and k, we could try each set in the universal family in parallel in $\mathcal{NC}$ and be sure that one such set isolates the minimum cut so that the approximation algorithm can find it.

In section 7, we give an $\mathcal{NC}$ algorithm for constructing a polynomial-size (in u and k) (u, k, α)-universal isolating family. Before doing so, we give the details of how it can be used to solve the minimum-cut problem in $\mathcal{NC}$.

6. Minimum cuts and extensions. We start by solving the min-cut problem for unweighted graphs. To extend this result to weighted graphs, we must first digress to finding *minimum multiway cuts* (minimum sets of edges that partition the graph into more than two parts) and *approximately minimum cuts* (cuts with value nearly equal to the minimum-cut). The weighted minimum-cut problem is then solved by reduction to these problems.

6.1. Unweighted minimum cuts. We first consider the unweighted min-cut problem. We have already shown (in section 4) that all we need to do is solve the set-isolation problem for the $n^{O(1)}$ small cuts on both sides of the minimum cut. In our case, the universe size u is just the number of graph edges m, the safe-set size is $c = O(m/n)$ (which we can estimate to within a factor of 3 using the approximation algorithm), and there are $n^{O(1)}$ target sets. Thus in $\mathcal{NC}$ we can generate and try all members of a universal isolating family of $m^{O(1)}$ sets. One of the sets we try will be an isolator for our problem, intersecting all small cuts except for the minimum cut. When we contract the edges in this set, running the approximation algorithm on the contracted graph will find the minimum cut. The number of processors used is $m^{O(1)}$, and the running time is polylogarithmic in m. In other words, the minimum cut can be found in $\mathcal{NC}$.

6.2. Extension to multiway cuts. The r-way min-cut problem is to partition a graph's vertices into r nonempty groups so as to minimize the number of edges crossing between groups. An $\mathcal{RNC}$ algorithm for constant r appears in [18], and a more efficient one appears in [23]. For constant r, we can use the set-isolation technique to solve the r-way cut problem in $\mathcal{NC}$. The next lemma reduces to Lemma 4.3 when $r = 2$.

LEMMA 6.1. *In an r-way min-cut $(X_1, \ldots, X_r)$ of value c, each X_i has minimum cut at least $2c/(r-1)(r+2)$.*

Proof. Assume that set X_1 has a cut (A, B) of value w. We prove the lemma by lower-bounding w.

Suppose that two sets X_i and X_j are connected by more than w edges (where $1 \neq i \neq j \neq 1$). Then merging X_i and X_j and splitting X_1 into A and B would yield an r-way cut of smaller value, a contradiction. Summing over $\binom{r-1}{2}$ pairs X_i and X_j, it follows that the total number of cut-edges not incident on X_1 can be at most $\binom{r-1}{2}w$.

Now suppose that more than $2w$ edges connect X_1 and some X_j for $j \neq 1$. Then more than w edges lead from X_j to either A or B, say A. Thus splitting X_1 into A and B and merging A with X_j would produce a smaller r-way cut, a contradiction. It follows that the number of edges incident on X_1 can be at most $2(r-1)w$.

Combining the previous two arguments, we see that the r-way cut value c must satisfy

$$c \leq \binom{r-1}{2} w + 2w(r-1),$$

implying the desired result. □

Combining the two previous lemmas shows that there is a polynomial-sized set of target cuts that we can eliminate with the set-isolation technique to isolate the minimum r-way cut.

THEOREM 6.2. *On unweighted graphs, the r-way min-cut problem can be solved in $\mathcal{NC}$ for any constant r.*

Proof. We proceed exactly as in the two-way min-cut case. Consider the minimum r-way cut $(X_1, \ldots, X_r)$ of value c. By the previous lemma, the minimum cut in each component is at least $2c/(r-1)(r+2)$. Thus by Lemma 4.3, the number of cuts in each X_i whose size is less than $2c$ is $O(n^{2(r-1)(r+2)})$, a polynomial for constant r. It follows that we can find a universal isolating family containing an isolator for the minimum r-way cut. Contracting the edges in this isolator yields a graph in which each component of the r-way minimum cut has no small cut. Then the (two-way) minimum cut in this contracted graph must be a "part of" the r-way minimum cut.

More precisely, it cannot cut any one of the X_i's, so each X_i is entirely on one or the other side of the cut. We can now find minimum cuts in each of the sides of the minimum cut; again, they must be part of the r-way minimum cut. If we repeat this process r times, we will find the r-way minimum cut. □

6.3. Extension to approximate cuts. We can similarly extend our algorithm to enumerate all cuts with value within any constant-factor multiple of the minimum cut. This plays an important role in our extension to weighted graphs.

LEMMA 6.3 (see [18]). *The number of r-way cuts with value within a multiplicative factor of α of the r-way min-cut is $O(n^{2\alpha(r-1)})$.*

LEMMA 6.4. *Let c be the min-cut value in a graph. If (A, B) is a cut with value αc, then the minimum r-way cut in A has value at least $(r - \alpha)c/2$.*

Proof. Let $\{X_i\}_{i=1}^r$ be the optimum r-way cut of A, with value β. Let us contract each X_i to a single vertex (removing resulting self loops) and sum the degrees of these r vertices two different ways. There are β edges (the r-way cut edges) with one endpoint in each of two different X_i's. This contributes 2β to the sum of contracted-vertex degrees. There are also αc edges with exactly one endpoint in A (and thus in sum X_i), namely, the edges crossing cut (A, B). These contribute an additional αc to the sum of degrees. Thus the sum of the degrees of the X_i's is $2\beta + \alpha c$. Counting a different way, we know that each X_i has degree no less than the minimum cut, so the sum of degrees is at least rc. Thus $2\beta + \alpha c \geq rc$, and the result follows. □

THEOREM 6.5. *For any constant α, all cuts with value at most α times the minimum cut's can be found in $\mathcal{NC}$.*

Proof. For simplicity, assume without loss of generality that α is an integer. Fix a particular cut (A, B) of value αc. Let $r = \alpha + 2$. By Lemma 6.4, the minimum r-way cut in A (and in B) has value at least c. Lemma 6.3 says that as a consequence there are $n^{O(1)}$ r-way cuts in A (or B) with value less than $3r\alpha c$. Define a set-isolation instance whose target sets are all such multiway cuts and whose safe set is the cut (A, B). By finding an isolator for the instance and contracting the edges in it, we ensure that the minimum r-way cut in each of A and B exceeds $3r\alpha c$.

Suppose that after isolating the cut we want, we run our parallelization of Matula's algorithm, constructing k-jungles with $k = \alpha c$. Since the r-way cut is at least $3r\alpha c$ in each of A and B, at most $(r - 1)$ vertices in each set have degree less than $6\alpha c$. It follows that so long as the number of vertices exceeds $4r$, the number of edges will reduce by a constant factor in each iteration of the algorithm. In other words, in $O(\log m)$ steps, the number of vertices will be reduced to $4r$ in such a way that the cut of value αc is preserved. We can find it by examining all possible partitions of the $4r$ remaining vertices since there are only a constant number. □

There is an obvious extension to approximate multiway cuts; however, we omit the notationally complicated exposition.

6.4. Extension to weighted graphs. If the weights in a graph are polynomially bounded integers, we can transform the graph into a multigraph with a polynomial number of edges by replacing an edge of weight w with w parallel unweighted edges. Then we can use the unweighted multigraph algorithm to find the minimum cut.

If the edge weights are reals, we use the following reduction from [18, 19] to the case of integral polynomial edge weights. We first estimate the minimum cut to within a multiplicative factor of $O(n^2)$. To do so, we simply compute a maximum spanning tree of the weighted graph and then let w be the weight of the minimum-weight edge of this maximum spanning tree. Removing this edge partitions the maximum spanning tree into two sets of vertices such that no edge of G connecting them has

weight greater than w (or else it would be in the maximum spanning tree). Therefore, the minimum cut is at most n^2w. On the other hand, the maximum spanning tree has only edges of weight at least w, so one such edge crosses every cut. Thus the minimum cut is at least w. This estimate can clearly be done in $\mathcal{NC}$.

Given this estimate, we can immediately contract all edges of weight exceeding n^2w since they cannot cross the min-cut. Afterwards, the total amount of weight remaining in the graph is at most n^4w. Now multiply each edge weight by n^3/w so that that the minimum cut is scaled to be between n^3 and n^5. If we now round each edge weight to the nearest integer, we will be changing the value of each cut by at most n^2 in absolute terms, implying a relative change by at most a $(1 + 1/n)$ factor. Thus the cut of minimal weight in the original graph has weight within a $(1 + 1/n)$ factor of the minimum cut in the new graph. By Theorem 6.5, all such nearly minimum cuts can be found in $\mathcal{NC}$ with the previously described algorithms. All we need to do to find the actual minimum cut is inspect every one of the small cuts we find in the scaled graph and compute its value according to the original edge weights.

7. Solving the set-isolation problem. It remains to show how to construct a universal isolating family in $\mathcal{NC}$. Our goal is, given U and k, to generate a (u, k, α)-universal isolating family of size polynomial in u and k in $\mathcal{NC}$. We first give an existence proof for universal families of the desired size. The proof uses the probabilistic method; for this and other ideas in this section involving randomization, refer to the book by Motwani and Raghavan [28].

THEOREM 7.1. *There exists a (u, k, α)-universal isolating family of size $uk^{O(1)}$ for any constant α.*

Proof. First, assume that the safe-set size s is known explicitly. We use a standard probabilistic existence argument. Fix attention on a particular set-isolation instance with safe-set size s. Suppose we mark each element of the universe with probability $(\log 2k)/\alpha s$ and let the marked elements form one member of the universal family. With probability $k^{-O(1)}$, the safe-set is not marked but all the target sets are. If so, then the marked elements form an isolator for the given instance. Thus if we perform $k^{O(1)}$ trials, we can reduce the probability of not producing an isolator for this instance to $1/2$. If we do this $uk^{O(1)}$ times, then the probability of failure on the instance is $2^{-uk^{O(1)}}$. If we now consider all $2^{uk^{O(1)}}$ set-isolation instances, the probability that we fail to to generate an isolator for all of them during all the trials is less than 1.

Now consider the assumption that s is known. It can be removed by performing the above randomized marking trial for each value of s in the range $1, \ldots, u$; their union would be a universal isolating family. This would increase the size of the family by a factor of u. More efficiently, since a constant-factor estimate of s suffices, we could apply the construction for $s = 1, 2, 4, 8, \ldots, u$ and take the union of the results. This would increase the number of sets in the universal isolating family by a factor of $\log u$ but would increase the total size (in number of elements) of all sets in the family by only a constant factor. □

It is not very hard to see that this existence proof can be converted into a randomized ($\mathcal{RNC}$) construction of a polynomial-size (u, k, α)-universal isolating family. In the application to the minimum-cut problem, we only know of an upper bound on the value of k, but it is clear that this suffices for the existence proof and the randomized construction. Furthermore, since we can get a constant-factor approximation to the minimum cut, we do not in fact need to construct isolators for all possible values of s but only for the estimate.

7.1. Deterministically constructing universal families. We proceed to derandomize the construction of a universal isolating family. As in the randomized construction, we can assume that the safe-set size s is known. While performing the derandomization, we fix our attention on a particular set-isolation instance and show that our construction will contain an isolator for that instance. It will follow that our construction contains an isolator for every instance.

Our derandomization happens in two steps. We first replace the randomized construction's fully independent marking by pairwise-independent marking and show that despite this change we have a good chance of marking any *one* target set while avoiding the safe set. We then use random walks on expanders to let us mark *all* the target sets simultaneously while avoiding the safe set.

7.1.1. Pairwise independence. We first show how pairwise-independent marking isolates any one target set from the safe set. The analysis of the use of pairwise instead of complete independence is fairly standard [7, 25, 28], and the particular proof given below is similar to that of Luby, Naor, and Naor [26].

Choose $p = 1/(2+\alpha)s$. Suppose each element of U is marked with probability p *pairwise independently* to obtain a *mark set* $M \subseteq U$. For any element $x \in U$, we define the following two events under the pairwise-independent marking:

- $\mathcal{M}_x$: the event that element x is marked;
- $\mathcal{S}_x$: the event that element x is marked but no element of the safe set S is marked.

We have that $\Pr[\mathcal{M}_x] = p$ and the events $\{\mathcal{M}_x\}$ are pairwise independent. We say that a mark set is *good for* T_i if some element of T_i is marked but no element of S is marked, and in that case the set of marked elements M is called a *good set for* T_i. The mark set is an isolator if it is good for every T_i.

Observe that the mark set M is good for a target set T_i if and only if the event $\mathcal{S}_x$ occurs for some element $x \in T_i$. The following lemmas help to establish a constant lower bound on the probability that M is good for T_i.

LEMMA 7.2. *For any element* $x \in U \setminus S$,

$$\Pr[\mathcal{S}_x] \geq p(1 - sp).$$

Proof. The probability that x is marked but no element of S is marked can be written as the product of the following two probabilities:

- the probability that x is marked, and
- the probability that no element of S is marked conditional upon x being marked.

We obtain that

$$\begin{aligned}
\Pr[\mathcal{S}_x] &= \Pr[\cap_{j\in S}\overline{\mathcal{M}}_j \cap \mathcal{M}_x] \\
&= \Pr[\cap_{j\in S}\overline{\mathcal{M}}_j \mid \mathcal{M}_x] \times \Pr[\mathcal{M}_x] \\
&= (1 - \Pr[\cup_{j\in S}\mathcal{M}_j \mid \mathcal{M}_x]) \times \Pr[\mathcal{M}_x] \\
&\geq \left(1 - \sum_{j\in S} \Pr[\mathcal{M}_j \mid \mathcal{M}_x]\right) \times \Pr[\mathcal{M}_x].
\end{aligned}$$

Since $x \notin S$, we have that $j \neq x$. The pairwise independence of the marking now

implies that $\Pr[\mathcal{M}_j \mid \mathcal{M}_x] = \Pr[\mathcal{M}_j]$, and so we obtain that

$$\begin{aligned}\Pr[\mathcal{S}_x] &\geq \left(1 - \sum_{j \in S} \Pr[\mathcal{M}_j]\right) \times \Pr[\mathcal{M}_x] \\ &= (1 - sp)p. \qquad \square\end{aligned}$$

LEMMA 7.3. *For any pair of elements x, $y \in U \setminus S$,*

$$\Pr[\mathcal{S}_x \cap \mathcal{S}_y] \leq p^2.$$

Proof. Using conditional probabilities as in the proof of Lemma 7.2, we have that

$$\begin{aligned}\Pr[\mathcal{S}_x \cap \mathcal{S}_y] &= \Pr[(\mathcal{M}_x \cap \mathcal{M}_y) \cap (\cap_{j \in S} \overline{\mathcal{M}}_j)] \\ &= \Pr[\cap_{j \in S} \overline{\mathcal{M}}_j \mid \mathcal{M}_x \cap \mathcal{M}_y] \times \Pr[\mathcal{M}_x \cap \mathcal{M}_y] \\ &\leq \Pr[\mathcal{M}_x \cap \mathcal{M}_y] \\ &= p^2,\end{aligned}$$

where the last step follows from the pairwise independence of the marking. $\square$

THEOREM 7.4. *The probability that the pairwise-independent marking is good for any specific target set T_i is bounded from below by a positive constant.*

Proof. Recall that $|T_i| \geq t = \alpha s$ and arbitrarily choose a subset $T \subseteq T_i$ such that $|T| = t = \alpha s$, assuming without loss of generality that t is a positive integer. The probability the mark set M is good for T_i is given by $\Pr[\cup_{x \in T_i} \mathcal{S}_x]$. We can lower bound this probability as follows:

$$\begin{aligned}\Pr[\cup_{x \in T_i} \mathcal{S}_x] &\geq \Pr[\cup_{x \in T} \mathcal{S}_x] \\ &\geq \sum_{x \in T} \Pr[\mathcal{S}_x] - \sum_{x,y \in T} \Pr[\mathcal{S}_x \cap \mathcal{S}_y],\end{aligned}$$

using the principle of inclusion-exclusion. Invoking Lemmas 7.2 and 7.3, we obtain that

$$\begin{aligned}\Pr[\cup_{x \in T_i} \mathcal{S}_x] &\geq tp(1 - sp) - \binom{t}{2} p^2 \\ &\geq tp(1 - sp) - t^2 p^2 \\ &= \alpha sp(1 - sp) - (\alpha sp)^2 \\ &= \frac{\alpha}{2(2 + \alpha)},\end{aligned}$$

where the last expression follows from our choice of $sp = 1/(2 + \alpha)$. Clearly, for any positive constant α, the last expression is a positive constant. $\square$

A pairwise-independent marking can be achieved using $O(\log u + \log s)$ random bits as a seed to generate pairwise-independent variables for the marking trial [7]. The $O(\log u)$ term comes from the need to generate u random variables; the $O(\log s)$ term comes from the fact that the denominator in the marking probability is proportional to s. Since $s \leq u$, the number of random bits needed to generate the pairwise-independent marking is $O(\log u)$.

We can boost the probability of success to any desired constant β by using $O(1)$ independent iterations of the random marking process, each yielding a different mark

set. This increases the size of the seed needed by only a constant factor. We can think of this pairwise-independent marking algorithm as a function f that takes a truly random seed R of $O(\log u)$ bits and returns $O(1)$ subsets of U. Randomizing over seeds R, the probability that $f(R)$ contains at least one good set for target T_i is at least β.

7.1.2. Expander walks. The above construction lets us isolate any *one* target set from the safe set with reasonable probability. The next step is to isolate *all* target sets simultaneously. We do so by reducing the probability of failure from a constant $1-\beta$ to $k^{-O(1)}$, making it unlikely that we fail to mark any one target set. This relies on the behavior of random walks on expanders.

We need an explicit construction of a family of bounded-degree expanders, and a convenient construction is that of Gabber and Galil [13]. They show that for sufficiently large n, there exists a graph G_n on n vertices with the following properties: the graph is 7-regular; it has a constant expansion factor; and, for some constant ϵ, the second eigenvalue of the graph is at most $1-\epsilon$. The following is a minor adaptation of a result due to Ajtai, Komlós, and Szemerédi [2] which presents a crucial property of random walks on the expander G_n. (Refer to Cohen and Wigderson [8], Impagliazzo and Zuckerman [16], and Motwani and Raghavan [28] for a formal definition of expanders and further details about random walks on expanders.)

THEOREM 7.5 (see [2]). *There exist constants $\beta, \gamma > 0$ such that for any subset $B \subseteq V(G_n)$ of size at most $(1-\beta)n$ and for a random walk of length $\gamma \log k$ on G_n, the probability that the vertices visited by the random walk are all from B is $O(k^{-2})$.*

Notice that performing a random walk of length $\gamma \log k$ on G_n requires $O(\log n + \log k)$ random bits—choosing a random starting vertex requires $\log n$ random bits and, since the degree is constant, each step of the walk requires $O(1)$ random bits. We use this random walk result as follows. Each vertex of the expander corresponds to a seed for the mark-set generator f described above; thus $\log n = O(\log u)$, implying that we need a total of $O(\log u + \log k)$ random bits for the random walk. Choosing B to be the set of bad seeds for T_i, i.e., those that generate set families containing no good sets for T_i, and noting that by construction B has size $(1-\beta)n$ allows us to prove the following theorem.

THEOREM 7.6. *A (u, k, α) universal family for U of size $(uk)^{O(1)}$ can be generated in $\mathcal{NC}$ for any constant α.*

Proof. Use $\Theta(\log u + \log k)$ random bits in the expander walk to generate $\Theta(\log k)$ pseudorandom seeds. Then use each seed as an input to the mark-set generator f. Let H denote the $\Theta(\log k)$ sets generated throughout these trials (we give $\Theta(\log k)$ inputs to f, each of which generates $O(1)$ sets). Since the probability that f generates a good-for-i set on a random input is β, we can choose constants and apply Theorem 7.5 to ensure that with probability $1 - 1/k^2$, one of our pseudorandom seeds is such that H contains a good set for T_i. It follows that with probability $1 - 1/k$, H contains good sets for every one of the T_i's. Note that the good sets for different targets might be different. However, consider the collection C of all possible unions of sets in H. Since H has $O(\log k)$ sets, C has size $2^{|H|} = k^{O(1)}$. One set in C consists of the union of all the good-for-some-i sets in H; this set intersects every T_i but does not intersect the safe set, and it is thus an isolator for our instance.

We have shown that with $O(\log u + \log k)$ random bits, we generate a family of $k^{O(1)}$ sets such that there is a nonzero probability that one of the sets isolates the safe set. It follows that if we try all possible $O(\log u + \log k)$-bit seeds, one of them must yield a collection that contains an isolator. All these seeds together will generate

$(uk)^{O(1)}$ sets, one of which must be the desired one.

For a given input seed, the pairwise-independent generation of sets by f is easily parallelizable. Given a particular $O(\log u + \log k)$-bit seed for the expander walk, Theorem 7.5 says that performing the walk to generate the seeds for f takes $O(\log u + \log k)$ time. We can clearly do this in parallel for all possible seeds. The various sets that are output as a result must contain a solution for any particular set-isolation instance; it follows that the output collection is a (u, k, c)-universal isolating family. □

It should be noted that by itself, this set-isolation construction is not sufficient for derandomization. Combined directly with the technique of Luby, Naor, and Naor [26], it can find a set of edges that contains an edge incident on each vertex but not any of the minimum-cut edges. Unfortunately, contracting such an edge set need only halve the number of vertices (e.g., if the edge set is a perfect matching), so $\Omega(\log n)$ phases would still be necessary. This approach would therefore use $\Omega(\log^2 n)$ random bits, just as [26] did. The power of the technique comes through its combination with the approximation algorithm, which allows us to solve the entire problem in a single phase with $O(\log n)$ random bits. This, of course, lets us fully derandomize the algorithm.

8. Conclusion. We have shown that, in principle, the minimum-cut problem can be solved in $\mathcal{NC}$. This should be viewed as a primarily theoretical result since the algorithm in its present form is extremely impractical. A natural open problem is to find an efficient $\mathcal{NC}$ algorithm for minimum cuts. An easier goal might be to improve the efficiency of the approximation algorithm. Our algorithm uses m^2/n processors. Matula's sequential approximation algorithm uses only linear time, and the $\mathcal{RNC}$ min-cut algorithm of [23] uses only n^2 processors. Also, an $\mathcal{RNC}$ $(2+\epsilon)$-approximation algorithm using only a linear number of processors is given in [20]. These facts suggest that a more efficient $\mathcal{NC}$ algorithm might be possible.

We also introduced a new combinatorial problem, the set-isolation problem. This problem seems very natural, and it would be nice to find further applications for it. Other applications of the combination of pairwise independence and random walks would also be interesting.

REFERENCES

[1] A. Aggarwal and R. J. Anderson, *A random $\mathcal{NC}$ algorithm for depth first search*, Combinatorica, 8 (1988), pp. 1–12.

[2] M. Ajtai, J. Komlós, and E. Szemerédi, *Deterministic simulation in logspace*, in Proc. 19th ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 132–140.

[3] D. Applegate, AT&T Bell Labs, Murray Hill, NJ, personal communication, 1992.

[4] M. Bellare, O. Goldreich, and S. Goldwasser, *Randomness in interactive proofs*, Comput. Complexity, 3 (1993), pp. 319–354; abstract in Proc. 31st Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 563–572.

[5] R. A. Botafogo, *Cluster analysis for hypertext systems*, in Proc. 16th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR), ACM, New York, 1993, pp. 116–125.

[6] J. Cheriyan, M. Y. Kao, and R. Thurimella, *Scan-first search and sparse certificates: An improved parallel algorithm for k-vertex connectivity*, SIAM J. Comput., 22 (1993), pp. 157–174.

[7] B. Chor and O. Goldreich, *On the power of two-point sampling*, J. Complexity, 5 (1989), pp. 96–106.

[8] A. Cohen and A. Wigderson, *Dispersers, deterministic amplification, and weak random sources*, in Proc. 30th Annual Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp 14–19.

[9] C. J. Colbourn, *The Combinatorics of Network Reliability*, International Series of Monographs on Computer Science, Vol. 4. Oxford University Press, Oxford, UK, 1987.

[10] E. A. Dinitz, A. V. Karzanov, and M. V. Lomonosov, *On the structure of a family of minimum weighted cuts in a graph*, in Studies in Discrete Optimization, A. A. Fridman, ed., Nauka, Moscow, 1976, pp. 290–306.

[11] L. R. Ford, Jr. and D. R. Fulkerson, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

[12] L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[13] O. Gabber and Z. Galil, *Explicit construction of linear-sized superconcentrators*, J. Comput. System Sci., 22 (1981), pp. 407–420.

[14] L. M. Goldschlager, R. A. Shaw, and J. Staples, *The maximum flow problem is logspace complete for P*, Theoret. Comput. Sci., 21 (1982), pp. 105–111.

[15] J. Hao and J. B. Orlin, *A faster algorithm for finding the minimum cut in a directed graph*, J. Algorithms, 17 (1994), pp. 424–446; preliminary version in Proc. 3rd Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Phialdelphia, 1992, pp. 165–174.

[16] R. Impagliazzo and D. Zuckerman, *How to recycle random bits*, in Proc. 30th Annual Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 222–227.

[17] A. Israeli and Y. Shiloach, *An improved parallel algorithm for maximal matching*, Inform. Process. Lett., 22 (1986), pp. 57–60.

[18] D. R. Karger, *Global min-cuts in $\mathcal{RNC}$ and other ramifications of a simple mincut algorithm*, in Proc. 4th Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1993, pp. 21–30.

[19] D. R. Karger, *Random sampling in graph optimization problems*, Ph.D. thesis, Stanford University, Stanford, CA, 1994; contact author at (karger@lcs.mit.edu); available by ftp from (theory.lcs.mit.edu), directory (pub/karger).

[20] D. R. Karger, *Using randomized sparsification to approximate minimum cuts*, in Proc. 5th Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1994, pp. 424–432.

[21] D. R. Karger, *A randomized fully polynomial approximation scheme for the all terminal network reliability problem*, in Proc. 27th ACM Symposium on Theory of Computing, ACM, New York, 1995, pp. 11–17.

[22] D. R. Karger and R. Motwani, *Derandomization through approximation: An $\mathcal{NC}$ algorithm for minimum cuts*, in Proc. 25th ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 497–506.

[23] D. R. Karger and C. Stein, *An $\tilde{O}(n^2)$ algorithm for minimum cuts*, in Proc. 25th ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 757–765.

[24] R. M. Karp, E. Upfal, and A. Wigderson, *Constructing a perfect matching is in random $\mathcal{NC}$*, Combinatorica, 6 (1986), pp. 35–48.

[25] M. G. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1053.

[26] M. G. Luby, J. Naor, and M. Naor, *On removing randomness from a parallel algorithm for minimum cuts*, Technical report TR-093-007, International Computer Science Institute, Berkeley, CA, 1993.

[27] D. W. Matula, *A linear time $2 + \epsilon$ approximation algorithm for edge connectivity*, in Proc. 4th Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1993, pp. 500–504.

[28] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, 1995.

[29] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[30] H. Nagamochi and T. Ibaraki, *Computing edge connectivity in multigraphs and capacitated graphs*, SIAM J. Discrete Math., 5 (1992), pp. 54–66.

[31] M. Padberg and G. Rinaldi, *An efficient algorithm for the minimum capacity cut problem*, Math. Programming, 47 (1990), pp. 19–39.

[32] J. C. Picard and M. Queyranne, *Selected applications of minimum cuts in networks*, INFOR: Canad. J. Oper. Res. Inform. Process., 20 (1982), pp. 394–422.

SIAM J. COMPUT.
Vol. 26, No. 1, pp. 273-290, February 1997

016

RESOURCE BOUNDS FOR SELF-STABILIZING MESSAGE-DRIVEN PROTOCOLS*

SHLOMI DOLEV†, AMOS ISRAELI‡, AND SHLOMO MORAN§

Abstract. Self-stabilizing message-driven protocols are defined and discussed. The class *weak exclusion* that contains many natural tasks such as *ℓ-exclusion* and *token passing* is defined, and it is shown that in any execution of any self-stabilizing protocol for a task in this class, the configuration size must grow at least in a logarithmic rate. This last lower bound is valid even if the system is supported by a time-out mechanism that prevents communication deadlocks. Then we present three self-stabilizing message-driven protocols for token passing. The rate of growth of configuration size for all three protocols matches the aforementioned lower bound. Our protocols are presented for two-processor systems but can be easily adapted to rings of arbitrary size. Our results have an interesting interpretation in terms of automata theory.

Key words. self-stabilization, message passing, token passing, shared memory

AMS subject classifications. 68M10, 68M15, 68Q10, 68Q20

PII. S0097539792235074

1. Introduction.

A distributed system is a set of state machines, called *processors*, which communicate either by *shared variables* or by *message-passing*. In the first case, the system is a *shared-memory* system; in the second case, the system is a *message-passing* system. A distributed system is *self-stabilizing* if it can be started in any *possible* global state. Once started, the system regains its consistency by itself, without any kind of an outside intervention. The self-stabilization property is very useful for systems in which processors may crash and then recover spontaneously in an arbitrary state. When the intermediate period in between one recovery and the next crash is long enough, the system stabilizes. Self-stabilizing systems were defined and discussed first in the fundamental paper of Dijkstra [7]. The work of [7] as well as most of the following work on self-stabilizing systems assume the communication model of shared variables. Among these papers are [2], [4], [6], [8], [9], [11], [14], [15], [17], [19], [20], and [22].

In the study of fault-tolerant message-passing systems, it is customarily assumed that messages might be corrupted over links; hence processors may enter arbitrary states and link contents may be arbitrary. Self-stabilizing protocols treat these problems naturally since they are designed to recover from inconsistent global states. Surprisingly, there are very few papers which address self-stabilizing message-passing systems. The earliest research in this model was done by Gouda and Multari in [13], [21]. In that work, they have developed a self-stabilizing sliding-window protocol and two-way handshake that use unbounded counters. They proved that any self-stabilizing message-passing protocol must use time-outs and have infinite number of safe states. Following [13], two additional works dealt with self-stabilizing protocols

* Received by the editors July 29, 1992; accepted for publication (in revised form) May 3, 1995. An extended abstract of this work was presented at the 10th Annual ACM Symposium on Principles of Distributed Computing, 1991 [10].
http://www.siam.org/journals/sicomp/26-1/23507.html
† Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel (dolev@cs.bgu.ac.il).
‡ Intel, P.O. Box 1659, Haifa 31015, Israel (aisraeli@iil.intel.com). The research of this author was partially supported by the NWO through NFI Project ALADDIN under contract NF 62-376.
§ Department of Computer Science, Technion, Haifa 32000, Israel (moran@cs.technion.ac.il).

in this model. The work of Katz and Perry [16] presents a general tool for extending an arbitrary message-passing protocol to a self-stabilizing protocol. The work of Afek and Brown [1] presents a self-stabilizing version of the well-known alternating-bit protocol (see, e.g., [5]).

In this work, we research complexity issues related to self-stabilizing message-passing systems; to do this, we define a *configuration* of any message-passing system as a list of the states of the processors and of the messages which are in transit on each link. The *size* of a configuration of a message-passing system is the number of bits required to encode the configuration entirely. A protocol for a message-passing system is *message driven* if any action of the processors is initiated by receiving a message. In the work of Gouda and Multari [13], it is proven that any message-driven protocol has a possible configuration in which all processors are waiting for messages but there are no messages on any link. This unwanted situation is called *communication deadlock*. A self-stabilizing system should stabilize when started from any possible initial configuration, including a configuration with communication deadlock. This implies that a nontrivial, completely asynchronous, self-stabilizing system cannot be message-driven. This problem can be dealt with in at least two methods. Gouda and Multari [13] proposed the use of a time-out mechanism which preserves the message-driven structure of the protocol at the expense of compromising the complete asynchronicity. On the other hand, Katz and Perry [16] chose to give up the message-driven structure and present protocols for which at any configuration there is at least one processor whose next operation is sending a message. Thus there is an execution in which in every atomic step a message is sent, and no message is ever received. In this execution, the size of the configurations grows *linearly*.

In this work, we define and study the class of *self-stabilizing message-driven* protocols. By the argument of [13], there exists no self-stabilizing message-driven protocol which is completely asynchronous. Since we look for protocols whose configuration size does not grow at a linear rate, we resort to slightly limited assumptions of asynchronous behavior. For lower bounds, we assume an abstract time-out device which detects communication deadlocks and initiates the system upon their occurrence. Consequently, the lower bounds we present take into account only executions in which no communication deadlock occurs. Our upper bounds assume that in every initial configuration, there is at least one message on some link. This assumption is much weaker than the assumption on a general time-out mechanism.

A specific task which we study in details is *token passing*. Informally, the *token-passing* task is to pass a single token fairly among the system's processors. Usually, it is assumed that in the system's predefined initial configuration, there exists a single token. In self-stabilizing system in which there is no predefined initial configuration, each execution should reach a configuration in which exactly one token is present in the entire system. Token passing is a very basic task in fault-tolerant systems; among other works, it was studied in [12] for some fault-tolerant message-passing systems and in [14] for self-stabilizing shared-memory systems. The token-passing task can be looked at as a special case of mutual exclusion since possession of the single token can be interpreted as a permission to enter the critical section.

In the first part of the presentation we prove a lower bound on the configuration size for protocols for a large class of tasks called *weak exclusion*. The weak-exclusion class contains all nontrivial tasks which require continuous changes in the system's configuration; in particular, this class includes both ℓ*-exclusion* and *token passing*. We show that the configuration size of any self-stabilizing protocol which realizes any

weak-exclusion task is at least logarithmic in the number of steps executed by the protocol. The lower bound holds for message-driven protocols for any weak-exclusion task, including protocols for systems equipped with time-out mechanism. This result should be compared with a result of [13] where it is shown that any message-driven self-stabilizing protocol (not necessarily for weak-exclusion tasks) must have infinitely many safe system configurations, but it is not shown that each specific execution must contain infinitely many distinct configurations, as implied by our results. Our lower bound does not specify which *part* of the system grows; is it the size of the memory used by the state machines, the size of messages stored on the links, the number of messages stored on the links, or all of these together?

We then present three self-stabilizing message-driven protocols for token passing. The communication-deadlock problem is avoided by the assumption that at least a single message is present on some communication link. Using this assumption, we present three token-passing protocols for two processors each. The rate of growth of configuration size for all three protocols matches the aforementioned lower bound. All protocols are presented for systems with two processors but can be easily adapted to work on rings of arbitrary size *without increasing their asymptotic complexity.* This is done by considering the ring as a single virtual link.

In the first protocol, the sizes of both processors' memory and messages grow unboundedly with time; this protocol uses ideas similar to the ideas of the sliding-window protocol of [13]. The second protocol is an improvement on the first protocol in which the size of the memory of the processors grows (in logarithmic rate) while the size of the link content is bounded. The second protocol is an improvement of the deterministic alternating-bit protocol of [1]. The third protocol is a self-stabilizing token-passing protocol in which processors are *deterministic* finite-state machines and messages are of fixed size. The only growing part of the system is the number of messages on the links; the rate of growth matches the lower bound mentioned above.

Our results can also be described in terms of automata theory, as follows. Let Σ be an alphabet. Define a *queue machine* Q to be a finite-state machine which is equipped with a queue, which initially contains an arbitrary nonempty word from Σ^+. Initially, Q is in an arbitrary state, and in each step it performs the following: (a) reads and deletes a letter from the head of the queue, (b) adds one or more letters from Σ to the tail of the queue, and (c) moves to a new state. The computational power of a queue machine is severely limited by the fact that its input alphabet and its work alphabet are *identical.* In particular, a queue machine cannot perform simple tasks like computing the length of the input word or even deciding whether the input word contains a specific letter.

Assume that the alphabet contains a specified subset τ of *token letters.* A queue machine is a *token controller* if, starting with a nonempty queue of arbitrary content, the queue eventually contains exactly one occurrence of a letter from τ forever. Our lower-bound result implies that if a token controller exists, then in every computation, the size of the queue must grow at least logarithmically in the number of moves of the machine. Our third protocol implies that a token controller whose configuration size growth matches the lower bound exists. In view of the fact that a queue machine *cannot* compute any estimation of the number of occurrences of letters from τ in the input word, this latter result appears to be somewhat counterintuitive.

2. Self-stabilizing message-driven systems.

2.1. Asynchronous message-driven systems. An *asynchronous, distributed, message-passing system* contains n processors, where each processor is a state machine.

Processors communicate using message passing along *links*. An edge $e = (i, j)$ of G stands for two directed links, one from P_i to P_j and the other from P_j to P_i. A message sent from P_i to P_j can be delayed for an unbounded amount of time on the connecting link. Messages which did not yet reach their destination are stored on the link and transferred in first-in first-out (FIFO) order.

A processor is uniquely defined by the set of its *atomic steps*. Whenever a processor is active, it executes one of its atomic steps. In a *message-driven* protocol, an atomic step of any processor P begins with a **receive** operation in which P receives a message from one of its incoming links. The atomic step ends with zero or more **send** operations in which P sends messages along some of its outgoing links. An atomic step a of P_i is defined by $a = (i,\ s_{i_1},\ (e, \mathrm{msg}),\ (e_1, \mathrm{msg}_1),\ (e_2, \mathrm{msg}_2), \ldots, (e_\ell, \mathrm{msg}_\ell), s_{i_2})$, meaning that P_i is in state s_{i_1}, e is the link through which P_i receives the message msg, $e_1, e_2, \ldots, e_\ell$ are the outgoing links along which P_i sends $\mathrm{msg}_1, \mathrm{msg}_2, \ldots, \mathrm{msg}_\ell$, respectively, and s_{i_2} is the state of P_i following the execution of this atomic step.

Let n and m be the number of processors and links, respectively, in the system. For $1 \le i \le n$, denote the set of states of P_i by S_i. A *configuration* of the system is a vector of states of all processors together with m lists—a list for every link—of messages stored on that link. A configuration is denoted by $c = (s_1, s_2, \ldots, s_n, M_{e_1}, M_{e_2}, \ldots, M_{e_j}, \ldots, M_{e_m})$, where $s_i \in S_i$, $1 \le i \le n$, and M_{e_j} is a list of the messages stored on e_j for $1 \le j \le m$. Let c be a configuration as above, and let $a = (i, s_{i_1}, (e, \mathrm{msg}), (e_1, \mathrm{msg}_1), (e_2, \mathrm{msg}_2), \ldots, (e_\ell, \mathrm{msg}_\ell), s_{i_2})$ be an atomic step. a is *applicable* to (P_i in) c if P_i is in state s_{i_1} in c and msg is the first message stored on e in c.

Application of a to c yields the *result* configuration c'. We denote this fact by $c \xrightarrow{a} c'$. A sequence of atomic steps, $A = (a_1, a_2, \ldots)$, is applicable to configuration c_0 if the first atomic step in the sequence, a_1, is applicable to c_0, the second atomic step is applicable to c_1, where $c_0 \xrightarrow{a_1} c_1$, and so on. An *execution* $E = (c_0, a_1, c_1, a_2, \ldots)$ is a (finite or infinite) sequence which starts with some arbitrary configuration c_0 and for every $i > 0$, $c_{i-1} \xrightarrow{a_i} c_i$; that is, the sequence of atomic steps $A = (a_1, a_2, \ldots)$ is applicable to c_0. *Note:* Since we deal with self-stabilizing systems, we do not assume any particular initial configuration; every configuration is a valid initial configuration. Execution E is *fair* if every atomic step that is applicable infinitely often is executed infinitely often.

Each execution E defines a partial order on the atomic steps of E by the relation *happened before* of Lamport in [18]:

1. If a_i and a_j are atomic steps executed by the same processor in E and a_i appears before a_j in E, then a_i *happened before* a_j.
2. If during a_i the message msg is sent and during a_j the same message msg is received, then a_i *happened before* a_j.
3. If a_i *happened before* a_j and a_j *happened before* a_k, then a_i *happened before* a_k.

We also adopt the definition of *concurrent* atomic steps from [18]: atomic steps $a_1, \ldots, a_k$ are said to be *concurrent* in an execution E if for $1 \le i < j \le k$, a_i does not *happen before* a_j and a_j does not *happen before* a_i in E. The following proposition gives a sufficient condition for a set of steps to be concurrent in some execution.

PROPOSITION 2.1. *Let $P_{i_1}, \ldots, P_{i_k}$ be k distinct processors and let $\{a_1, \ldots, a_k\}$ be a set of atomic steps where a_j is applicable to P_{i_j}, $1 \le j \le k$, in some configuration c. Then there exists an execution in which the atomic steps $a_1, \ldots, a_k$ are concurrent.*

Proof. Observe that once step a is applicable to processor P in configuration

c, step a remains applicable to P in all subsequent configurations. The execution E is defined as the execution that starts from c in which processors $P_{i_1}, \ldots, P_{i_k}$ are activated one after the other and each processor P_{i_j} executes a_j. The proof follows since the processors are distinct and since in E, no message that was sent during a_j, $1 \leq j \leq k$, is received before a_k is executed. Note that the proposition holds for any system in which once some step is applicable, it remains applicable as long as it is not executed. □

An asynchronous protocol PR is defined by a set of n processors. By the above definitions, an asynchronous protocol defines a set of executions that satisfy the following:

1. Let $E = (c_0, a_1, c_1, a_2, \ldots)$ be an arbitrary execution of PR. Then every prefix of E is also an execution of PR.

2. Let $E = (c_0, a_1, c_1, a_2, \ldots, a_r, c_r)$ be arbitrary finite execution of PR. Then for every atomic step a and configuration c satisfying $c_r \xrightarrow{a} c$ PR has an execution $E \circ (a, c)$.[1]

2.2. Self-stabilizing message-driven protocols. A self-stabilizing system demonstrates a *legitimate behavior* some time after it is started from an arbitrary configuration. A natural way to specify a behavior in an abstract way is by a set of sequences of configurations. We define *tasks* as sets of *legitimate sequences*. The semantics of any specific task is expressed by requirements on its sequences. Intuitively, each legitimate sequence can be thought of as an execution of a protocol, but we do not require this formally. For instance, the mutual-exclusion task is defined as the set of sequences of configurations which satisfy the following: Each processor has a subset of its states called *critical section*; in each configuration, at most one processor is in its critical section, and every processor is in its critical section in infinitely many configurations. To formally define a task T, one should specify for each possible system ST a set of *legitimate sequences* for ST. The *task* T is defined as the union of the legitimate sequence set over all possible systems. A configuration c of a system is *safe* with respect to a task T and a protocol PR if any fair execution of PR starting from c belongs to T.

In proving lower-bound results on self-stabilizing message-driven protocols, we assume that the system can recover from a *communication deadlock* (called deadlock from now on). In other words, when we prove our lower bounds, we assume only that the protocol stabilizes in executions in which no deadlock occurs. For this purpose, we distinguish between two types of deadlocks: *global* and *local*. A configuration c is a global deadlock configuration if no atomic step is applicable to c. Our first lower bound holds for asynchronous systems that can recover from global deadlocks by applying a *global time-out* mechanism. This abstract mechanism initiates a system in a global deadlock configuration to a default initial configuration, after which no deadlock occurs. Below we present the requirement for self-stabilizing systems equipped with a global time-out mechanism. In this definition, the system is required to reach a safe configuration in every infinite fair execution. Note that by our definition, an infinite fair execution does not have a deadlock configuration.

Self-stabilization (*assuming global time-out mechanism*). Let PR and LE be a message-driven protocol and set of legitimate sequences, respectively. Protocol PR is self-stabilizing relative to LE if for every c, there is an execution of PR that starts with c and every such infinite fair execution reaches a safe configuration with respect

[1] For sequences S_1 and S_2, $S_1 \circ S_2$ denotes the concatenation of S_1 and S_2.

to LE and PR.

Later on, we prove a lower bound that holds for systems immune to a stronger type of communication deadlock called local deadlock. Processor P is in a *local deadlock* during execution E if P is activated (i.e., executes an atomic step) only finitely many times during E. The second lower bound holds for systems equipped with an abstract *local* time-out mechanism which prevents such executions (e.g., by enabling each processor which is idle for a sufficiently long time to initiate the system to some default configuration after which no deadlock is possible). Note that a local time-out mechanism is strictly stronger than a global time-out mechanism.

Self-stabilization (*assuming local time-out mechanism*). Let PR and LE be a message-driven protocol and set of legitimate sequences, respectively. Protocol PR is sellf-stabilizing relative to LE if for every c, there is an execution of PR that starts with c, and every such infinite fair execution, in which each processor is activated infinitely often, reaches a safe configuration with respect to LE and PR.

3. Lower bound. In this section, we prove a lower bound on the rate in which the configuration size grows along every execution of any protocol for a large class of tasks called *weak exclusion*. This class contains all nontrivial tasks which require continuous changes in the system's configuration; in particular, this class includes both *ℓ-exclusion* and *token passing*. For an execution E, denote by $\mathcal{A}_i(E)$ the set of distinct atomic steps executed by P_i during E. A task belongs to the class *weak exclusion* if its set of legitimate sequences, LE, satisfies the following:

WE. For any $E \in LE$, there exists a set of two or more atomic steps $B = \{a_{i_1}, \ldots, a_{i_k}\}$, $k \leq n$, where $a_j \in \mathcal{A}_{i_j}(E)$, such that the atomic steps in B are never concurrent during E.

We first consider self-stabilizing protocols for systems equipped with a global time-out mechanism. For these protocols, we prove that in every execution (in which no communication deadlock occurs), all configurations are distinct. From this we conclude that the configuration size of every self-stabilizing protocol which realizes any weak-exclusion task is at least logarithmic in the number of steps executed by the protocol. Throughout the proof, we assume that PR is a self-stabilizing message-driven protocol for an arbitrary weak-exclusion task in a system with a global time-out mechanism. At the end of this section, we present a slightly weaker lower bound for systems with a local time-out mechanism.

For any configuration c and any link e, denote by M_e^c the sequence of messages present on e in c. For any execution E, denote by $M_{e,s}^E$ ($M_{e,r}^E$) the sequence of messages sent (received) along e during E.

PROPOSITION 3.1. *For every execution $E = (c_0, a_1, \ldots, a_r, c_r)$ and for every link e, $M_e^{c_0} \circ M_{e,s}^E = M_{e,r}^E \circ M_e^{c_r}$.*

Proof. The left-hand side of the equation contains the messages present on e in c_0 concatenated with the messages sent during E through e. The right-hand side of the equation contains the messages received during E through e concatenated with the messages left on e in c_r. It is not hard to verify that both sides of the equation represent the same sequence of messages. □

An execution $E = (c_0, a_1, \ldots, c_{\ell-1}, a_\ell, c_\ell)$ whose result configuration c_ℓ is equal to its initial configuration c_0 is called a *circular* execution. A link e is *active* in a circular execution E if some messages are received (and hence, by the circularity of E, some messages are sent) along e in E. Repeating a circular execution E forever yields an infinite execution E^∞ which is not necessarily fair—the original execution may have an applicable step a which is never executed during E. The step a is

applicable throughout E^∞, but it is never executed. To avoid this problem, the original circular execution is changed by removing all messages from links that are not active throughout E. The resulting execution, which is still called E, is still circular, and its infinite repetition E^∞ is a fair infinite execution. Observe that an execution in which a certain configuration appears more than once has a circular subexecution, $\overline{E} = (c_i, a_{i+1}, \ldots, a_{i+\ell}, c_{i+\ell}) \equiv (\overline{c_0}, \overline{a_1}, \ldots, \overline{a_\ell}, \overline{c_\ell})$, where $c_i = c_{i+\ell} = \overline{c_0} = \overline{c_\ell}$. Thus to show that in every execution of PR, all the configurations are distinct, we assume that PR has a circular subexecution $\overline{E}$ and reach a contradiction by showing that PR is not self-stabilizing.

Using $\overline{E}$, we now construct an initial configuration c_{init} by changing the list of messages in transit on the system's links. For each link e, the list of messages in transit on e at c_{init} is obtained by concatenating the list of messages in transit on e at $\overline{c_0}$ with the list of all messages sent on e during $\overline{E}$. Roughly speaking, the effect of this change is creating an additional "layer" of messages that helps to decouple each **send** from its counterpart **receive** and achieve an additional flexibility in the system which enables the proof of the lower bound. Formally, c_{init} is obtained from $\overline{c_0}$ as follows:

- The state of each processor in c_{init} is equal to its state in $\overline{c_0}$.
- For any active link in $\overline{E}$, $M_e^{c_{\text{init}}} = M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}}$, and for any nonactive link in $\overline{E}$, $M_e^{c_{\text{init}}}$ is empty.

Let $\overline{A}(i)$ be the sequence of atomic steps executed by P_i during $\overline{E}$. Define $merge(\overline{A})$ to be the set of sequences obtained by all possible mergings of all sequences $\overline{A}(i)$, $1 \le i \le n$, while keeping the internal order in each $\overline{A}(i)$. Note that all of the sequences in $merge(\overline{A})$ have the same finite length and contain the same atomic steps in different orders.

LEMMA 3.2. *Every $A \in merge(\overline{A})$ is applicable to c_{init}, and the resulting execution, $E_A = (c_{\text{init}}) \circ A$, is a circular execution of PR.*

Proof. Let A be an arbitrary sequence in $merge(\overline{A})$ and let P_i be an arbitrary processor of the system. Then we have the following: (i) The initial state of P_i in c_{init} is equal to its initial state in $\overline{c_0}$. (ii) In c_{init}, all messages which P_i receives during $\overline{E}$ are stored on P_i's appropriate incoming links in the right order. (iii) The atomic steps of P_i appear in A in the same order in which they appear in $\overline{A}(i)$. (i)–(iii) imply that the sequence A is applicable to c_{init}, and the application of A to c_{init} yields an execution E_A with resulting configuration c_{res}, whose state vector is equal to the state vector of c_{init} and in which for every active link, $M_{e,s}^{E_A} = M_{e,s}^{\overline{E}}$ and $M_{e,r}^{E_A} = M_{e,r}^{\overline{E}}$.

To prove that the execution obtained is circular, it remains to be shown that the content of every link in the resulting configuration c_{res} is equal to its content in c_{init}, i.e., $M_e^{c_{\text{init}}} = M_e^{c_{\text{res}}}$. For any arbitrary link e it holds that

1. $M_e^{c_{\text{init}}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{c_{\text{res}}}$ (by Proposition 3.1 and by the fact that $M_{e,s}^{E_A} = M_{e,s}^{\overline{E}}$ and $M_{e,r}^{E_A} = M_{e,r}^{\overline{E}}$) and
2. $M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{\overline{c_0}}$ (by Proposition 3.1 and the circularity of $\overline{E}$).

Replacing $M_e^{c_{\text{init}}}$ in equation 1 with its explicit contents yields

3. $M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{c_{\text{res}}}$.

Using equation 2 to replace $M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}}$ by $M_{e,r}^{\overline{E}} \circ M_e^{\overline{c_0}}$ in equation 3 gives

4. $M_{e,r}^{\overline{E}} \circ M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} = M_{e,r}^{\overline{E}} \circ M_e^{c_{\text{res}}}$.

Dropping $M_{e,r}^{\overline{E}}$ from the two sides of equation 4 yields the desired result: $M_e^{c_{\text{init}}} = M_e^{\overline{c_0}} \circ M_{e,s}^{\overline{E}} = M_e^{c_{\text{res}}}$, which proves the lemma. □

Define *blowup*($\overline{E}$) to be the set of executions whose initial sate is c_{init} and whose sequence of atomic steps belongs to *merge*($\overline{A}$). Notice that for every circular execution $\overline{E}$ and for every execution $E \in$ *blowup*($\overline{E}$), it holds that $\mathcal{A}_i(\overline{E}) = \mathcal{A}_i(E)$.

LEMMA 3.3. *For any set of atomic steps $B = \{a_1, \ldots, a_k\}$, $k \leq n$, where $a_j \in \mathcal{A}_{i_j}(\overline{E})$, there is an execution $E \in$ blowup($\overline{E}$) that contains a configuration for which all the atomic steps in B are concurrent.*

Proof. For notational simplicity, assume that $k = n$ and that $B = \{a_1, a_2, \ldots, a_n\}$. Let $A \in$ *merge*($\overline{A}$) be the sequence constructed as follows: first take all the steps in $\overline{A}(1)$ that precede a_1, then take all the steps in $\overline{A}(2)$ that precede $a_2, \ldots$, then take all the steps in $\overline{A}(n)$ that precede a_n. Applying the sequence constructed so far to c_{init} results in a configuration in which all of the a_i's are applicable. This sequence is completed to a sequence A in *merge*($\overline{A}$) by taking the remaining atomic steps in an arbitrary order, which keeps the internal order of each $\overline{A}_i$. □

LEMMA 3.4. *Let PR be a self-stabilizing message-driven protocol for an arbitrary weak-exclusion task T in a system with a global time-out mechanism. If PR has a circular execution $\overline{E}$, then PR has an infinite fair execution E^∞ none of whose configurations are safe for T.*

Proof. Let E be an arbitrary execution in *blowup*($\overline{E}$). Define E^∞ to be the infinite execution obtained by repeating E forever. By the definition of *blowup*($\overline{E}$), E^∞ is fair. Thus it remains to show that no configuration in E^∞ is safe.

Assume by way of contradiction that some configuration c_0 in E^∞ is safe. Now we construct a finite circular execution E' whose sequence of atomic steps A' is obtained by concatenating sequences from *merge*($\overline{A}$), that is, $\mathcal{A}_i(E') = \mathcal{A}_i(\overline{E})$. Since PR is a protocol for some weak-exclusion task, E' should have some set of atomic steps $B = \{a_1, \ldots, a_k\}$, where $a_j \in \mathcal{A}_{i_j}$, that are never applicable for a single configuration c during E'. We reach a contradiction by refuting this statement for E'. For this we choose some arbitrary enumeration $\mathcal{B} = B_1, \ldots, B_s$, of all the sets containing n atomic steps of n distinct processors. Execution E' is constructed by first continuing the computation from c_0 as in E until configuration c_{init} is reached. Then apply Lemma 3.3 to extend E' by s consecutive executions $E_1, \ldots, E_s$, where E_k, $1 \leq k \leq s$, contains a configuration in which all the steps in B_k are applicable and that ends with c_{init}. The proof follows. *Note:* Execution E' can be repeated forever to obtain an infinite execution which does not have any suffix in LE; thus the protocol PR is not even pseudo-self-stabilizing (see [3]). □

The proof for the lower bound is completed by the following theorem.

THEOREM 3.5. *Let PR be a self-stabilizing message-driven protocol for an arbitrary weak-exclusion task in a system with a global time-out mechanism. For every execution E of PR, all the configurations of E are distinct. Hence for every $t > 0$, the size of at least one of the first t configurations in E is at least $\lceil \log_2(t) \rceil$.*

Proof. Assume by way of contradiction that there exists an execution E of PR in which not all the configurations are distinct; then E contains a circular subexecution $\overline{E}$. By Lemma 3.4, there exists an infinite execution E' of PR which is obtained by an infinite repetition of some execution from *blowup*($\overline{E}$) and which never reaches a safe configuration—a contradiction. □

To prove a similar lower bound for systems with a *local* time-out mechanism, the definition of a circular execution must be modified. Removing messages from nonactive links to construct an infinite execution from $\overline{E}$ as in the proof of Theorem 3.5 may yield an infinite execution in which some processor is enabled only finitely many times. In order to allow repetitions of finite executions to form an infinite fair

execution, in which every processor is active infinitely often, we require that each such finite execution contains an atomic step of each processor in the system. For this we need the concept of a *round* of an execution. Let E' be a minimal prefix of an execution E in which every processor receives a message; E' is the first *round* of E. Let E'' be the suffix of E which satisfies $E = E' \circ E''$. The second round of E is the first round of E'', and so on. Let E_i be the prefix that contains the first i atomic steps of E. Let $t_i = R(E_i)$ be the number of rounds in E_i. The next theorem presents a lower bound for systems equipped with a local time-out mechanism. The proof is similar to the proof of Theorem 3.5.

THEOREM 3.6. *Let PR be a self-stabilizing message-driven protocol for an arbitrary weak-exclusion task in a system with a local time-out mechanism. For every execution E of PR, E does not contain a circular subexecution which contains a complete round. From this we conclude that in each execution of PR, E, the first t rounds contain at least t distinct configurations. Hence for every $t > 0$, the size of at least one configuration in E_i at least $\lceil \log_2(t_i) \rceil$. In particular, in any fair execution, the configuration size is unbounded.*

4. Upper bound. The *token-passing* task is defined informally as a set of executions in which a single token is present in the entire system and is passed fairly among the system's processors. Token passing is a special case of mutual exclusion since possession of the single token can be interpreted as permission to enter the critical section. For this reason, token passing also satisfies the weak-exclusion property, and hence the lower bound of section 3 holds for it. In particular, it means that any self-stabilizing message-driven protocol PR for token passing must use some unbounded resource since in any infinite execution, the system size grows beyond any bound. In this section, we present three self-stabilizing token-passing protocols for systems of two processors. In each protocol, the configuration size grows during every execution at a rate that matches the lower bound. Each of these protocols can be easily adapted to work on rings of arbitrary size *without increasing its asymptotic complexity* by considering the ring as a single virtual link. Similar ideas can be used for adapting the protocols to arbitrary rooted tree systems.

By a standard symmetry argument, there exists no self-stabilizing, deterministic, token-passing protocol if the processors are identical. Hence in this section, we assume that the system consists of two distinct processors, called *sender* and *receiver*, connected by two links. The first link carries messages from the sender to the receiver while the second link carries messages from the receiver back to the sender. The receiver processor is identical in all three protocols and it is probably the simplest possible finite-state machine. Its program is to copy each message it receives from its incoming link to its outgoing link without any alteration. To the outside world, the combined behavior of the receiver and the two links looks like the behavior of a single queue whose head and tail are used by the receiver. In our analysis, we ignore the receiver and consider systems with a single processor, the sender, communicating with itself using a single link on which messages are kept in FIFO order. In each step, the sender consumes a message from the head of the link and puts one (or more) messages back at the tail of the link. Tokens are represented by a special symbol, T, which is appended to some of the messages. Our protocols specify the messages that carry a token, but they do not explicitly use the token symbol T. The protocol should guarantee that there is eventually a unique message in the system to which T is appended. All of our protocols assume that there is initially at least one message on the link (this assumption is weaker than both the global and the local versions of the

time-out mechanism). With this last assumption, the requirement that the link never becomes empty is equivalent to the requirement that whenever a message is received, at least one message is sent. Hence in every step of the protocol, the sender receives the message on the head of the (single) link and then puts one or more messages at the link's end. The three protocols we present are as follows:

Protocol 1. In this protocol, the sender is an infinite-state machine, and in every execution, the link capacity is unbounded.

Protocol 2. In this protocol, the sender is an infinite-state machine, but in each infinite execution, the link capacity is bounded (the bound for each specific execution depends on its initial configuration).

Protocol 3. In this protocol, both processors are finite-state machines.

```
1 do forever
2     receive(msg_counter)
3     if msg_counter ≥ counter then (* token arrives *)
4     begin (* send new token *)
5        counter:= msg_counter +1
6        send(counter, T)
7     end
8     else send(counter)
9 end
```

FIG. 1. *Protocol* 1.

Protocol 1 (of the sender) appears in Figure 1. The sender uses a variable called *counter*. Each message consists of the present value of *counter*, possibly with the token symbol T. Whenever the sender receives a message whose counter value, msg_*counter*, is not smaller than *counter*, it sets *counter* := msg_*counter* + 1 and sends this new value of *counter* together with the token T; otherwise, the sender just sends the current value of *counter* (without the token T). The token letter T is not used by the protocol itself. The correctness of the protocol is based on the fact that eventually the value of *counter* will be larger than all the values that appear in the messages present on the link in the initial configuration. The asymptotic size of *counter* in each execution is $\Omega(\log t)$, where t is the number of messages sent. The details of the proof are omitted.

4.1. Aperiodic sequences. Protocols 2 and 3 use the following method: each message is associated with some ternary number which is called *color*. The protocol considers any message whose color is different from the color of the previous message as carrying a token. The sender has a local variable called *token_color*. At any given configuration the sender is sending a sequence of messages whose color is equal to (the value of) *token_color*, at the same time, the sender waits for a message whose color is equal to *token_color*. As long as the sender receives messages of different colors, it sends messages whose color is equal to *token_color*. Once the sender receives a message whose color is equal to *token_color*, it chooses a new *token_color* and initiates a new sequence of messages whose color is the new *token_color* by sending the first message in this new sequence. This first message is carrying a (virtual) token. Then the sender continues sending messages of the new *token_color* (without tokens), until it receives a message of the new *token_color*, and so on. Our goal is to reach a configuration after which the link always holds at most two consecutive sequences of messages where the colors of all messages in each sequence are equal.

In every step, the sender consumes a single message from the first sequence whose color is the previous *token_color* and produces one or more messages whose color is equal to the present *token_color*. After the last message whose color is the previous *token_color* is consumed, the link contains a single sequence of messages whose color is *token_color*. In the next step, the sender receives the (single) token carried by this sequence and sends it once again by initiating a new sequence of messages whose color is the new *token_color*. In each of the configurations described, there exists a single token which is carried by the first message of the sequence whose color is *token_color*. The correctness of the protocols follows from the fact that the sequences of token colors sent by the receiver is *aperiodic*, as defined below.

DEFINITION. *A sequence* $A = (a_1, a_2, \ldots)$ *is* periodic *if for some positive integer* k *and for all* $i \geq 1$, $a_i = a_{i+k}$. *The sequence* A *is* eventually periodic *if it has a suffix which is periodic.* A *is* aperiodic *if it is not eventually periodic.*

Aperiodic sequences over the integers $\{0, 1, 2\}$ were used in [1] in order to obtain self-stabilizing data-link protocols. Such sequences are created there by either a random-number generator or an infinite-state machine (in the first case, the algorithm is randomized). The elements of this sequence are used by the protocol of [1] whenever it has to decide on the ternary number to be sent with a new message. In this paper, aperiodic sequences are generated by using a counter and the sequence *xor* defined below.

DEFINITION. *For an integer* i, $xor(i)$ *is the sum of the bits* (mod 2) *in the binary representation of* i (*e.g.,* $xor(1) = xor(2) = 1, xor(3) = 0$). *The sequence* $(xor(1), xor(2), \ldots)$ *is denoted by xor.*

As we show later, the sequence *xor* is aperiodic.

```
1 do forever
2     receive(color)
3     if color = token_color then (* token arrives *)
4     begin (* send new token *)
5        token_color:= (color +xor(counter) + 1) (mod 3)
6        counter := counter +1
7     end
8     send(token_color)
9 end
```

FIG. 2. *Protocol* 2.

Protocol 2 (of the sender), which appears in Figure 2, is an improvement of the protocol that appears in [1] in the sense that it achieves the lower bound of the previous section. (The amount of memory used for producing the aperiodic sequence is neither addressed nor specified in [1].) In Protocol 2, the sender keeps a counter in its local memory; whenever a message with a new color is sent, the counter is incremented. The new color $\in \{0, 1, 2\}$ is determined by the previous color and by applying *xor* to the counter. Roughly speaking, the correctness of the protocol is implied by the fact that since *xor* is aperiodic, the sequence of *colors* generated by the sender is aperiodic as well. The nature of the variables and the correctness proof of Protocol 2 are easily derived from the description of Protocol 3 and from its correctness proof; hence they are omitted.

4.2. Informal description of Protocol 3. We now present Protocol 3, in which both processors are finite-state machines. It is easily observed that when an

aperiodic sequence is supplied by some external device, a finite-state machine can use this sequence to perform the protocol in [1]. Our construction uses the fact that the finite-state machine augmented with the previously described FIFO link can generate an aperiodic sequence. The finite-state machine uses the link both for message passing and for generating the aperiodic sequence, while its size is kept within the optimal bound. Protocol 3 can be easily transformed to a self-stabilizing data-link protocol in which both processors are finite-state machines.

Protocol 3 appears in Figure 3. In this protocol, each message is a pair $(color, bit)$, where $color \in \{0, 1, 2\}$ and $bit \in \{0, 1\}$. The local variables *color* and *token_color* are ternary variables while the local variables *counter_bit, counter_xor, carry,* and *new_counter_bit* are binary. The binary *xor* operation is denoted by $\oplus$. For a sequence $s = ((color_1, bit_1), \ldots, (color_k, bit_k))$ of such messages, $N(s)$ denotes the integer whose binary representation is bit_k, $bit_{k-1}, \ldots, bit_1$ (bit_1 is the least significant bit). A maximal sequence of consecutive messages of the same color sent by the sender is called a *block*. For each block b, $N(b)$ denotes the integer described above and $|b|$ denotes the number of messages in b. The first message in each block is viewed as a token. To show that the protocol is self-stabilizing, we have to prove that eventually the link contains exactly one message which is the first message in a block. This goal is achieved by making the sequence of the colors of the blocks aperiodic.

The sender uses a local variable called *token_color*, which denotes the color of the block it is now sending. It continues to send messages of this color as long as the colors of the messages it receives are different from *token_color*. Once the sender receives a message whose color is equal to *token_color* (which eventually means that all messages on the link belong to the same block), it (a) possibly sends one last message of the current block, (b) changes the value of *token_color*, and (c) sends the first message of a new block with this new color.

```
1  do forever
2     receive(color,counter_bit)
3     if color = token_color then (* token arrives *)
4       begin
5           if carry = 1 then send (color, 1)
            (* new token *)
6           token_color := (color + counter_xor + 1) (mod 3)
7           counter_xor := 0
8           carry := 1
9       end
10    counter_xor := counter_xor ⊕ counter_bit
11    new_counter_bit := carry ⊕ counter_bit
12    carry := carry ∧ counter_bit
13    send (token_color,new_counter_bit)
14 end
```

FIG. 3. *Protocol* 3.

In Lemma 4.1, we show that in every execution, the sender initiates infinitely many blocks. Let $b_1, b_2, \ldots$ be the sequence of blocks initiated by the sender, where the color of b_i is $color(b_i)$ and the integer it represents is $N(b_i)$, as defined above. The protocol is designed so that the following properties are kept:

(p1) The sequence $(color(b_1), color(b_2), \ldots)$ is aperiodic.

(p2) For every large enough i, $N(b_{i+1}) = N(b_i) + 1$, and the *bit* field in the last message of b_i is 1; that is, $N(b_i) = i + const$ for some constant *const*, and the representation of $N(b_i)$ by b_i has no leading zeroes, implying that $|b_i| = \lceil \log_2 N(b_i) \rceil$.

We will prove that (p1) implies that there is eventually only one token in the system, while (p2) guarantees that the size of the system is logarithmic in the number of steps. We now show that the protocol indeed satisfies (p1) and (p2). For this we describe the two rules by which the sender computes the bits and the colors it sends. We need the following definition.

DEFINITION. *Let $k \geq 1$. Denote by s_k the sequence of messages whose colors are different from color(b_k) which are received by the sender while it sends the block b_k, and denote by $N(s_k)$ the integer represented by s_k. Note that s_k consists of one or more complete blocks.*

Rule 1 (rule for computing *counter_bit*s). The *counter_bit* sent with each message is sent so that for each k, $N(b_k) = N(s_k)+1$, and $|b_k| = \max\{|s_k|, \lceil \log_2(N(b_k)) \rceil\}$. In other words, the *counter_bit*s sent in block b_k are obtained by adding 1 to the binary number represented by the messages received while this block is sent.

Rule 2 (rule for computing *token_color*). When receiving a message whose color is equal to the value of *token_color*, the new value of *token_color*, which is the color of the next block, b_{k+1}, is determined as follows: $color(b_{k+1}) = color(b_k) + xor(N(s_k)) + 1 \pmod 3$.

Note that Rule 1 can be implemented by a binary adder which is set to zero at the initiation of each new block, and Rule 2 can be implemented by a counter (mod 2). Thus both rules are easily implemented by a finite-state machine.

4.3. Correctness and complexity proofs of Protocol 3.

LEMMA 4.1. *In every fair execution E, the sender initiates an infinite number of blocks.*

Proof. The sender initiates a new block whenever it receives a message whose *color* is equal to the current value of *token_color*. In every atomic step in which the sender receives a message whose *color* is not equal to *token_color*, it sends a message—say M'—whose color is *token_color*. Since the link carries messages in FIFO order, the message M is eventually received by the sender and it initiates a new block not later than upon receipt of M. The lemma follows. □

A configuration in an execution is called a *limit configuration* if in the next step of the sender, a new *token_color* is computed; that is, the color of the next arriving message is equal to the present value of *token_color*. Observe that at a limit configuration c, the link contains a finite (possibly zero) number of complete blocks and one possibly incomplete block at the tail of the link. (This block may be incomplete since upon receipt of the next message, the sender may send one more message in this block by executing line 5 of the code.) The first block has the same color as the last (possibly incomplete) block. For an execution E, we denote by i_k the index of the kth limit configuration in E. In other words, c_{i_k} is the limit configuration just before b_k is initiated.

Next, we prove that the number of blocks in consecutive limit configurations does not increase.

LEMMA 4.2. *Let ℓ_k be the number of blocks in the limit configuration c_{i_k} (including the possibly incomplete block). Then $\ell_k \geq \ell_{k+1}$ with equality only if s_k is a single block.*

Proof. Let $m_k \geq 1$ be the number of blocks in s_k. In the subexecution starting with c_{i_k} and ending with $c_{i_{k+1}}$, one block is added to the link (namely b_k), and m_k

blocks of s_k are removed from it. Therefore, $\ell_{k+1} = \ell_k + 1 - m_k \leq \ell_k$. □

Next, we show that the number of blocks in the limit configurations must eventually get down to one. First, we need a technical lemma.

LEMMA 4.3.

(a) *The sequence xor is aperiodic.*

(b) *Let $(a_1, a_2, \ldots)$ be an eventually periodic sequence, and let $b_i = a_{i+1} - a_i$. Then the sequence $B = (b_1, b_2, \ldots)$ is also eventually periodic.*

(c) *Let $(a_1, a_2, \ldots)$ be an eventually periodic sequence. Then for each $i, p > 0$, the sequence $A(i,p) = (a_i, a_{i+p}, a_{i+2p}, \ldots)$ is also eventually periodic.*

Proof. (a) Assume in contradiction that the sequence $xor = (xor(1), xor(2), \ldots)$ is eventually periodic. Then there exist i and ℓ such that $xor(j) = xor(j + \ell)$ for every $j \geq i$. Let q be a nonnegative integer such that $2^q \leq \ell < 2^{q+1}$ and let d be an integer satisfying $d \geq q + 2$ and $2^d \geq i$. Consider the following cases:

- $xor(\ell) = 1$: By the definition of d, it holds that $xor(2^d + \ell) = 0$. Thus $1 = xor(2^d) \neq xor(2^d + \ell) = 0$.
- $xor(\ell) = 0$: Then $xor(\ell) = xor(2^q + \ell) = 0$, and $2^q + \ell < 2^d$. Hence $xor(2^d + 2^q + \ell) = 1$. Thus $0 = xor(2^d + 2^q) \neq xor(2^d + 2^q + \ell) = 1$.

Thus there exist a and b such that (1) $a > i$ and $b > i$, (2) $a - b = \ell$, and (3) $xor(a) \neq xor(b)$—a contradiction.

(b) This claim is trivial.

(c) Let j and ℓ be such that $xor(k) = xor(k + \ell)$ for every $k \geq j$. Then for every $p > 1$ and $k \geq j$, it holds that $a_k = a_{k+\ell p}$. Thus the sequence $A(i,p)$ is eventually periodic with period length $\leq \ell$. □

LEMMA 4.4. *In every fair execution E, there exists a suffix in which the number of blocks in the limit configurations is always one.*

Proof. By Lemma 4.2, this number never increases, and hence it eventually remains L for some constant $L > 0$ forever. We shall assume that $L > 1$ and derive a contradiction.

Call a limit configuration c_{i_k} *ultimate* if ℓ_k, the number of blocks in c_{i_k} is L. If c_{i_k} is ultimate, then $\ell_{k+1} = \ell_k$ and hence by Lemma 4.2, s_k is a single block, which must be b_{k-L}. Thus the first block that follows s_k is b_{k-L+1}. By the protocol, b_k is terminated when the sender receives a message whose color is equal to the color of b_k. Therefore, we have that the color of (the messages in) the block b_{k-L+1} is equal to the color of the messages in b_k, i.e., $color(b_{k-L+1}) = color(b_k)$. Hence the sequence $COLORS = (color(b_1), color(b_2), \ldots)$ is eventually periodic with period length $L - 1 > 0$. Let $BXOR = (xor(N(b_1)), xor(N(b_2)), \ldots)$. By the way in which $color(b_{k+1})$ is computed, we have that for an ultimate configuration c_{i_k}, $xor(N(b_k - L)) - [color(b_{k+1}) - color(b_k)] \pmod 3 - 1$. Hence by Lemma 4.3(b), if $COLORS$ is eventually periodic, so is $BXOR$. We shall derive a contradiction by showing that the sequence $BXOR$ is aperiodic.

Lemma 4.3(c) implies that in order to show that $BXOR$ is aperiodic, it is sufficient to show that for some positive i and p, the sequence $BXOR(i,p) = (xor(N(b_i)), xor(N(b_{i+p})), xor((N(b_{i+2p})), \ldots)$ is aperiodic. For this observe that for an ultimate configuration c_{i_k}, it must hold that $N(b_k) = N(s_k) + 1 = N(b_{k-L}) + 1$. Hence for any integer i, we have that $BXOR(i,L) = (xor(N(b_i)), xor(N(b_{i+L})), xor((N(b_{i+2L})), \ldots) = (xor(N), xor(N+1), xor(N+2), \ldots)$, where $N = N(b_i)$. Thus $BXOR(i,L)$ is a suffix of the sequence xor, which is aperiodic by Lemma 4.3(a). Hence $BXOR(i,L)$ is also aperiodic. This yields the desired contradiction. □

Lemma 4.4 and its proof imply that properties (p1) and (p2) hold. Property (p1)

holds since the proof of Lemma 4.4 shows that the sequence $COLORS$ is aperiodic. Property (p2) is proved as follows: Let E' be a suffix of E satisfying Lemma 4.4, and let c_{i_k} be any limit configuration in E'. Then by Rule 1, $N(b_{k+1}) = N(s_{k+1}) + 1 = N(b_k) + 1$, which easily implies (p2).

We now show that the space complexity of Protocol 3 indeed matches the lower bound of the previous section. Since both the number of states of a processor and the number of distinct messages in our protocol are constants, the size of a configuration is proportional to the number of messages in it. Therefore, to bound the size of a configuration from above, it is enough to bound the number of messages in it. In the next lemma we show that for each execution $E = (c_0, a_1, c_1, \ldots)$ of the protocol, the size of the ith configuration of E, c_i, is $O(\log_2(i))$. Let c_{i_k} denote the kth limit configuration of E, and let b_k be the corresponding block. We shall prove that $|b_k| = O(\log k)$.

LEMMA 4.5. *For every large enough k, the number of messages in the limit configuration c_{i_k} is $\lceil \log_2 N(b_{k-1}) \rceil$.*

Proof. By Lemma 4.4, there exists a suffix E' of E such that every limit configuration in E' contains one block. Clearly, it is suffices to prove the lemma for E'. As observed above, property (p2) eventually holds for every limit configuration in E'. The lemma follows. □

COROLLARY 4.6. *The number of messages in c_ℓ, the ℓth configuration of E, is $O(\log_2(\ell))$.*

Proof. Let E' be a suffix of E as in Lemma 4.5, and assume that ℓ is large enough so that c_ℓ belongs to E'. Then the number of messages in c_ℓ is equal to the number of messages in the next limit configuration, c_{i_k}, which is $O(\log_2 k)$ (for some k). The proof is completed by the observation that since $i_j \geq j$ for all j and since configuration $c_{i_{k-1}}$ precedes c_ℓ in E, we have that $\ell \geq i_{k-1} + 1 \geq (k-1) + 1 = k$. □

4.4. Larger systems. We now describe how to use our protocols in directed rings with more than two processors. The processors of the ring are denoted by $P_1, \ldots, P_n$, where P_1 is a sender while $P_2, \ldots, P_n$ are receivers. Whenever a processor P_i, $1 < i < n$, receives a message M from P_{i-1}, P_i sends M to P_{i+1}. Similarly, whenever P_n receives a message M from P_{n-1}, it sends M to P_1. Thus the ring behaves like a virtual link from the sender P_1 to itself. It is not hard to see that the existence of a single message on the entire ring prevents communication deadlocks; thus we assume that there is a time-out mechanism that guarantees this condition (this time-out mechanism is invoked only once to recover from the initial deadlock configuration). It can be proved in a way similar to our previous proofs that our protocols guarantee that there is eventually exactly one token that encircles the ring from the sender to itself. Actually, our protocols can be used in any connected system by hardwiring a directed ring that spans the entire system.

4.5. Construction of a token controller. In this subsection, we define queue machines and token controllers and interpret our results in these terms.

A *queue machine* Q is a finite-state machine which is equipped with a queue, which initially contains a nonempty word from Σ^+ for some (finite) alphabet Σ. In each step of its computation Q performs the following: (a) reads and deletes a letter from the head of the queue, (b) adds zero or more letters from Σ to the tail of the queue, and (c) moves to a new state. The computation terminates when Q halts or when its queue becomes empty, which prevents Q from performing any further steps.

The main difference between queue machines and various types of Turing machines is that the input alphabet and the work alphabet of a queue machine are identical.

For this reason, a queue machine cannot perform simple tasks like deciding the length of the input word or even deciding whether the input word contains a specific letter.[2]

We now define a *token controller*, which is a special type of queue machine. Assume that the alphabet Σ contains a specified subset τ of *token letters*. A queue machine is a *token controller* if, starting with a nonempty queue of arbitrary content, the queue eventually contains exactly one occurrence of a letter from τ forever.

A priori, it is not clear that a token controller exists. Observe that if a token controller exists, then its queue never becomes empty (since once the queue is empty, it remains so forever). More importantly, a token controller (if it exists) can never halt since it cannot guarantee that upon halting, the queue contains exactly one occurrence of a token letter. The last two observations imply that a token controller can be viewed as a special case of a token-passing system in which Σ is the set of messages sent by the protocol and τ is the set of messages that carry the token. We show below how to transform the sender from Protocol 3 to a token controller.

Define the alphabet Σ to be a set of triplets $(color, bit, t)$, where *color* and *bit* are as in Protocol 3 and t is either T in case the message carries a token (i.e., it is the first message of some block) or *nil* in case it does not. The set τ is defined as the set of all possible triplets whose third component is T. The two antiparallel FIFO links between the sender and the receiver are considered as a single queue. Receiving a message is regarded as deleting a letter from the head of the queue, while sending a message is regarded as appending a message to the end of the queue.

Since Protocol 3 guarantees that eventually exactly one message in every configuration is carrying a token, the queue machine described above is a token controller. Moreover, our lower bound results imply that this token controller is optimal with respect to the rate in which the size of the queue grows.

5. Self-stabilizing simulation of shared memory. In this section, we present a method for simulating self-stabilizing shared-memory protocols by self-stabilizing message-driven protocols. The simulated protocols are assumed to be in the shared-memory model defined in [9]. In this model, communication between neighbors P_i and P_j is carried out using a two-way link. The link is implemented by two shared registers which support **read** and **write** atomic operations. Processor P_i reads from one register and writes in the other while these functions are reversed for P_j. In the implementing system, every link is simulated by two directed links: one from P_i to P_j and the other from P_j to P_i. The heart of the simulation is a self-stabilizing implementation of the **read** and **write** operations.

The proposed simulation implements these operations by using a self-stabilizing token-passing protocol. For any pair of neighbors, we run the protocol on the two links connecting them. In order to implement our self-stabilizing token-passing protocol, we need to define for each link which of the processors acts as the sender and which of the processors acts as the receiver. We assume that the processors have distinct identifiers. Every message sent by each of the processors carries the identifier of that processor. Eventually, each processor knows the identifier of all its neighbors. In each link, the processor with the larger identifier acts as the sender while the other processor acts as the receiver. Since each pair of neighbors uses a different instance of the protocol, a separate time-out mechanism is needed for every such pair. In other words, a correct operation of the simulation requires that for any pair of neighbors,

[2] A variant of queue machine which can use arbitrary work alphabet is, in fact, an oblivious Turing machine, which is as powerful as a standard Turing machine.

there exists at least a single message on one of the two links connecting the neighbors.

We now describe the simulation of some arbitrary link e connecting P_i and P_j: In the shared-memory model, e is implemented by a register $R_{i,j}$ in which P_i writes and from which P_j reads and by a register $R_{j,i}$ for which the roles are reversed. In the simulating protocol, processor P_i (P_j) keeps a local variable called $r_{i,j}$ ($r_{j,i}$), which keeps the values of $R_{i,j}$ (respectively, $R_{j,i}$). Every token has an additional field called *VALUE*. Every time P_i receives a token from P_j, P_i writes the current value of $r_{i,j}$ in the *VALUE* field of that token. A **write** operation of P_i into $R_{i,j}$ is implemented simply by locally writing into $r_{i,j}$. A **read** operation of P_i from $R_{j,i}$ is implemented by the following steps:

1. P_i receives a token from P_j and then
2. P_i receives another token from P_j. The value read is the *VALUE* attached to the second token.

The correctness of the simulation is proved by showing that for every execution E whose initial configuration contains at least one message on each link, it is possible to linearize all the simulated **read** and **write** operations executed in E so that eventually every simulated **read** operation from $R_{i,j}$ returns the last value that was written to it (i.e., that the protocol simulates executions in the shared-memory model in which the registers are eventually atomic; see [20]). Define the time of a simulated **write** operation to $R_{i,j}$ to be the time in which the local write operation to $r_{i,j}$ is executed. Define the time of a simulated **read** operation of P_j from $R_{i,j}$ to be the time in which P_i sends the value of its local variable $r_{i,j}$ attached to the token that later reaches P_j in step 2 of the simulated **read**. Once each link holds a single token, all of the operations to register $r_{i,j}$ are linearized, and every read operation from $r_{i,j}$ returns the last value written to $r_{i,j}$.

Acknowledgment. We thank Alan Fekete for helpful remarks.

REFERENCES

[1] Y. Afek and G. M. Brown, *Self-stabilization of the alternating-bit protocol*, Distrib. Comput., 7 (1993), pp. 27–34.

[2] G. M. Brown, M. G. Gouda, and C. L. Wu, *A self-stabilizing token system*, IEEE Trans. Comput., 38 (1989), pp. 845–852.

[3] J. Burns, M. G. Gouda, and R. E. Miller, *Stabilization and pseudo stabilization*, Distrib. Comput., 7 (1993), pp. 35–42.

[4] J. E. Burns and J. Pachl, *Uniform self-stabilizing rings*, ACM Trans. Programing Lang. Systems, 11 (1989), pp. 330–344.

[5] K. A. Bartlet, R. A. Scantlebury, and P. T. Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Comm. Assoc. Comput. Mach., 12 (1969), pp. 260–261.

[6] J. E. Burns, *Self-stabilizing rings without demons*, Technical report GIT-ICS-87/36, Georgia Institute of Technology, Atlanta, 1987.

[7] E. W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Comm. Assoc. Comput. Mach., 17 (1974), pp. 643–644.

[8] E. W. Dijkstra, *Self-stabilizing systems in spite of distributed control* (EWD391), reprinted in Selected Writing on Computing: A Personal Perspective, Springer-Verlag, Berlin, 1982, pp. 41–46.

[9] S. Dolev, A. Israeli, and S. Moran, *Self stabilization of dynamic systems assuming only read/write atomicity*, Distrib. Comput., 7 (1993), pp. 3–16.

[10] S. Dolev, A. Israeli, and S. Moran, *Resource bounds for self stabilization message driven protocols*, in Proc. 10th Annual ACM Symposium on Principles of Distributed Computing, ACM, New York, 1991, pp. 281–294.

[11] S. Dolev, A. Israeli, and S. Moran, *Uniform dynamic self-stabilizing leader election*, in Proc. 5th International Workshop on Distributed Algorithms, Springer-Verlag, Berlin, 1991, pp. 167–179; IEEE Trans. Parallel Distrib. Systems, to appear.

[12] D. Dolev and D. Koller, *Token survival*, preprint, 1986.
[13] M. G. Gouda and N. J. Multari, *Stabilizing communication protocols*, IEEE Trans. Comput., 40 (1991), pp. 448–458.
[14] A. Israeli and M. Jalfon, *Token management schemes and random walks yield self stabilizing mutual exclusion*, in Proc. 9th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1990, pp. 119–131.
[15] A. Israeli and M. Jalfon, *Self-stabilizing ring orientation*, in Proc. 4th International Workshop on Distributed Algorithms, Springer-Verlag, Berlin, 1990, pp. 1–14; Inform. and Comput., 104 (1993), pp. 175–196.
[16] S. Katz and K. J. Perry, *Self-stabilizing extensions for message-passing systems*, Distrib. Comput., 7 (1993), pp. 17–26.
[17] H. S. M. Kruijer, *Self-stabilization (in spite of distributed control) in tree-structured systems*, Inform. Process. Lett., 8 (1979), pp. 91–95.
[18] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Comm. Assoc. Comput. Mach., 21 (1978), pp. 558–565.
[19] L. Lamport, *Solved problems, unsolved problems, and nonproblems in concurrency*, in Proc. 3rd ACM Symposium on Principles of Distributed Computing, ACM, New York, 1984, pp. 1–11.
[20] L. Lamport, *On interprocess communication, part* I: *Basic formalism*, Distrib. Comput. 1 (1986), pp. 77–85.
[21] M. Multari, *Toward a theory for self-stabilizing protocols*, Ph.D. dissertation, Department of Computer Science, University of Texas at Austin, Austin, TX, 1989.
[22] M. Tchuente, *Sur l'auto-stabilisation dans un réseau d'ordinateurs*, RAIRO Inform. Théor., 15 (1981), pp. 47–66.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 291–330, April 1997

FAIL-STOP SIGNATURES*

TORBEN PRYDS PEDERSEN† AND BIRGIT PFITZMANN‡

Abstract. Fail-stop signatures can briefly be characterized as digital signatures that allow the signer to prove that a given forged signature is indeed a forgery. After such a proof has been published, the system can be stopped. This type of security is strictly stronger than that achievable with ordinary digital signatures as introduced by Diffie and Hellman in 1976 and formally defined by Goldwasser, Micali, and Rivest in 1988, which was widely regarded as the strongest possible definition.

This paper formally defines fail-stop signatures and shows their relation to ordinary digital signatures. A general construction and actual schemes derived from it follow. They are efficient enough to be used in practice. Next, we prove lower bounds on the efficiency of any fail-stop signature scheme. In particular, we show that the number of secret random bits needed by the signer, the only parameter where the complexity of all our constructions deviates from ordinary digital signatures by more than a small constant factor, cannot be reduced significantly.

Key words. cryptography, authentication, digital signatures, fail-stop, discrete logarithm, factorization, randomization, computational security, information-theoretic security

AMS subject classification. 94A60

PII. S009753979324557X

1. Introduction and overview of results.

Traditional digital signatures, as introduced in [12] and formally defined in [20], allow a person, A (for Alice), to make signatures that everyone who knows A's public key can test. Such signatures are only computationally secure for the signer because they can be forged by persons with sufficiently large computing power. A person able to factor large integers can, for example, very easily forge RSA (Rivest–Shamir–Adleman) signatures (see [38]). Hence the security of these schemes relies on a computational assumption. Moreover, if a signature should be forged, it will be difficult for A to convince the bearer of the signed document or a third party that she did not make that signature.

Fail-stop signatures solve this problem by offering the signer a method for proving that a forgery has taken place. More precisely, even if a forger with unlimited com-

*Received by the editors March 11, 1993; accepted for publication (in revised form) April 3, 1995. This paper is the full version of the extended abstracts [E. van Heyst and T. P. Pedersen, "How to make efficient fail-stop signatures," in *Proc.* 1992 *Eurocrypt*, Lecture Notes in Comput. Sci. 658, Springer-Verlag, Berlin, 1993, pp. 366–377] and [E. van Heijst, T. P. Pedersen, and B. Pfitzmann, "New constructions of fail-stop signatures and lower bounds," in *Proc.* 1992 *Crypto*, Lecture Notes in Comput. Sci. 740, Springer-Verlag, Berlin, 1993, pp. 15–30] together with the definitions of fail-stop signatures, including their relations to ordinary digital signatures. A preliminary version of the definitions was only available in "grey" literature as parts of [B. Pfitzmann, "Für den Unterzeichner unbedingt sichere digitale Signaturen und ihre Anwendung," Diploma thesis, Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe, Karlsruhe, Germany, 1989] and [B. Pfitzmann and M. Waidner, "Formal aspects of fail-stop signatures," Technical Report 22/90, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, 1990]. Several intermediate papers with previous, less efficient constructions and discussions of applications are only referred to.

http://www.siam.org/journals/

†Cryptomathic, Århus Science Park, DK-8000 Århus, Denmark (tpp@cryptomathic.aau.dk). The research of this author was performed while at the Department of Computer Science of Århus University and supported by the Carlsberg Fondet.

‡Institut für Informatik, Universität Hildesheim, Samelsonplatz 1, D-31141 Hildesheim, Germany, (pfitzb@informatik.uni-hildesheim.de). Future address: Informatik VI, Universität Dortmund, D-44221 Dortmund, Germany. The Isaac Newton Institute in Cambridge, UK hosted this author during the final updates to this paper.

puting power makes an "optimal" forgery, a polynomially bounded signer can prove that the underlying computational assumption has been broken when she sees the forgery (except with negligible probability). Thus the signer can be protected from an arbitrarily powerful forger. Moreover, after the first forgery, all participants, or the system operator, know that the signature scheme has been broken, so that it can be stopped. Hence the name "fail-stop."

1.1. More about ordinary digital signatures. Digital signatures were introduced in [12] and became popular with the RSA scheme [38]. However, it was subsequently discovered that the security requirements made in [12] were too weak, and the structure of signature schemes described there did not allow the desired stronger security. A satisfactory definition was finally published in [20], together with a construction that is secure in this sense (see [20] for the intermediate history). We call such signature schemes "ordinary."

However, the notion of security was always only computational. For signature schemes of the original structure, [12] already showed that this is unavoidable: The signer has a secret key, which she uses to make signatures, and the signatures can be tested by everyone who knows her corresponding public key. Since signing and testing are polynomial-time, one can forge signatures, i.e., find values that pass the test, in nondeterministic polynomial time simply by guessing among all values up to a certain length. Additionally, nonpolynomial lower bounds for problems within NP are not known; hence to prove the security of any ordinary digital signature scheme, one has to make a computational assumption. The same argument also applies to the more general construction in [20]. Hence the effort in the theoretical treatment of signature schemes after [20] concentrated on weakening the necessary assumptions [1, 30, 39].

Note, additionally, that with ordinary digital signatures, it is always the signer whose security relies on the computational assumption. If the signer were allowed to disavow forged signatures, she could also disavow her real signatures (because there is no difference between them), even if the assumption was not broken at all. The recipient's security is unconditional, i.e., all signatures that he has accepted will definitely also be accepted by any third party asked to settle a dispute.

1.2. More about the fail-stop property. The fail-stop property can best be described by considering a judge in a dispute between the signer and a recipient of a digital signature. Usually, the judge will test if the signature is correct and give his verdict—"ok" or "not ok"—accordingly. Fail-stop signatures supply the judge with a third possibility: If the signer can prove that the signature is forged, the judge may say "forgery proved," which can be interpreted as saying that the basic assumption of the system has been broken. Naturally, this possibility of distinguishing forged signatures from authentic signatures only exists as long as the forger has not stolen the signer's key.

The definition of fail-stop signatures does not specify for how much of a system a particular proof of forgery is valid. As long as forging a single signature is provably as hard as breaking a particular computational assumption, it is wise to stop the whole system after any forgery because if one signature could be forged, one must expect that the same forger can make more forgeries. Therefore, the constructions usually assume that there is only one type of proof of forgery. However, it is no problem to make proofs of forgery specific to the keys of individual signers or even (although currently with some loss in efficiency) to each particular signature.

Furthermore, it is not a matter of the definition how one acts after the output "forgery proved." In particular, one obtains exactly the properties of ordinary digital signatures again if the technical verdict "forgery proved" is interpreted just like "signature correct" by the judge and everybody else. On the other hand, if it is agreed that all signatures for which forgery can be proved are rejected, one obtains a signature scheme in which the signer is unconditionally protected against forgeries, whereas the recipient is only computationally protected, i.e., an unrestricted adversary may achieve that a signature accepted by the recipient is later rejected by an honest judge. We call such a signature scheme a "dual signature scheme" because it is dual to ordinary digital signatures with respect to the security for the signer and recipients. Hence fail-stop signatures constitute the first (published) examples of dual signature schemes. Since fail-stop signatures furthermore allow the system to be stopped as soon as the basic assumption has been broken, they are a strictly stronger notion than each of these types of signatures.

As an example where fail-stop signatures, in their special role as dual signatures, may be advantageous, consider an electronic payment system where a customer signs her requests to the bank digitally. Since the bank most likely has much more computing power than the customer and since it can select the system and choose the security parameters, it is reasonable to protect the customer unconditionally, whereas the bank can rely on the customer not having sufficient computing power to repudiate her signatures. Thus the customer should sign with dual signatures and the bank with ordinary digital signatures. For more details about possible benefits of fail-stop signatures and possible advantages for the acceptability of digital signatures in law, see [36, 32].

1.3. Construction idea. Fail-stop signatures work very much like ordinary digital signatures. The signer has a secret key, which she uses to make signatures, and the signatures can be tested by everyone who knows her corresponding public key. A signature that passes this test is called *acceptable*. Now, the basic idea of the existing constructions of fail-stop signatures is that every message has many different acceptable signatures, of which the signer can only construct one (unless she breaks the underlying assumption); this one is called the *correct* signature. However, even an arbitrarily powerful forger does not have sufficient (Shannon) information to determine which of the many acceptable signatures is the correct signature on a new message. Consequently, with very high probability, a forged signature is different from the correct signature. Given a forged signature, the signer can exploit the knowledge of two different signatures on the same message (the forged signature and the correct one) to compute a proof of forgery.

Note that this construction allows an unrestricted signer to disavow her real signatures. However, since this can only occur if the computational assumption has in fact been broken, this is no problem. A proof of forgery does not indicate by *whom* the assumption has been broken, and hence it may have been the signer; but in any case, if this has happened, the system should be stopped.

1.4. Previous results. Fail-stop signatures were first mentioned in [43], and in the reports [31, 35], it was proved that such signatures exist if claw-free pairs of permutations exist; this is a computational assumption known from [20] (see also [4, 36] for descriptions). In particular, this shows that fail-stop signatures exist if factoring large integers or computing discrete logarithms is hard. The construction uses one-time signatures, similar to [25], i.e., messages are basically signed bit by bit.

Therefore, although messages can be hashed before signing and tree authentication is used (similar to [26]), this general construction is not very efficient.

In [32], an efficient variant especially suited for making customers unconditionally secure in on-line payment systems was presented. However, in this scheme, all signatures by one customer with the same key must have the same recipient (e.g., the bank in a payment system), and it is only a dual signature scheme, not a fail-stop signature scheme. Furthermore, signing is a three-round protocol between the signer and the recipient.

1.5. Related types of schemes. In [8], a dual undeniable signature scheme was presented. Undeniable signatures, introduced in [7], are a type of signature in which there is no public predicate that everyone can use to test signatures. Instead, signatures are verified and disavowed using interactive protocols that must be carried out with the signer. This construction was the first signature scheme to achieve unconditional security for signers without bit-by-bit signing. However, it is not as efficient as the following schemes. Although the signatures themselves are efficient, the verification protocol requires quite a lot of computation because it needs σ challenges (similar to signatures) to achieve an error probability of $2^{-\sigma}$. A similar scheme, but with so-called convertible undeniable signatures (cf. [5]), is contained in [22].

In [9], unconditionally secure signatures were introduced, i.e., signature-like schemes where both the signer and the recipients are unconditionally secure. In [37], a transferable version was presented, i.e., signatures can be passed on from one recipient to another. With this extension, unconditionally secure signatures could in principle replace other signatures in many applications. However, these constructions are too inefficient to be used in practice because they require a complicated interactive key-generation protocol and the signatures are very long. We showed in [23] (proofs in [34]) that the latter is unavoidable: to achieve an error probability of $2^{-\sigma}$, the length of unconditionally secure signatures that can be tested by M participants, including those that only have to settle disputes, is at least $M \cdot \sigma$.

1.6. Results in this paper. This paper presents fail-stop signatures in a unified way by giving definitions, efficient constructions, and lower bounds. Preliminary versions of the definitions appeared in [31, 35], and preliminary versions of the constructions and lower bounds appeared in [22, 23].

Definition (see section 3). The formal definition of fail-stop signatures is somewhat more general than that in the report [35] because it allows a more general method for key generation and more memory in the signing algorithm. This increases the scope of validity of the lower bounds.

Constructions (see sections 4 and 5). The first step in our constructions of fail-stop signatures is a general construction based on the concept of bundling homomorphisms, which allows only one message to be signed. This construction has as special cases the two schemes from the extended abstracts [22, 23], which we describe. Signing in the first of these two schemes requires two modular multiplications, whereas signing in the second requires approximately one exponentiation. In both schemes, testing a signature requires approximately two exponentiations. Thus these signatures compare very well with the currently proposed ordinary digital signatures. We then show extensions for signing an arbitrary number N of messages. In special cases, no efficiency is lost; in the general case, the length of signatures and the time needed for testing signatures on short messages grow by a factor of $\log(N)$.

Lower bounds (see section 6). Based on the definition of fail-stop signatures, we prove lower bounds on the size of the keys and the size of fail-stop signatures. For

these bounds, we assume that the probability that a forgery cannot be repudiated is smaller than $2^{-\sigma}$ for some security parameter σ and that the recipients have a similar level of security against very simple brute-force algorithms. Then the most important result is as follows:

- If N messages are to be signed, the signer needs at least $(N+1)(\sigma-1)$ secretly chosen random bits.

Thus the secret key in fail-stop signature schemes is basically a one-time key. However, this does not prevent efficient fail-stop signatures where many messages can be signed: We present an efficient construction where the size of the secret storage space is logarithmic in the number of messages to be signed, and an otherwise less efficient variant where this size is constant. This does not contradict the lower bounds because many secret bits can be deleted soon after they have been generated and used, i.e., the stored secret key varies with time. These constructions with small secret storage are important because secret storage is quite hard to realize since one needs a tamper-resistant device.

Additionally, we show the following:

- The entropy, and hence the length, of a signature is at least $2\sigma - 1$, and the entropy of the public key is at least σ.

These bounds are not much larger than similar bounds for ordinary digital signatures because they concern parameters where the difference between fail-stop signatures and ordinary digital signatures is quite small.

A comparison of the efficient constructions of fail-stop signatures with the lower bounds on unconditionally secure signatures mentioned in section 1.5 shows that fail-stop signatures provide the most viable way of protecting the signer unconditionally.

2. Notation. For a given probability space S (which will always be clear from the context), $\mathrm{P}(E)$ denotes the probability of the event E, and $[S]$ denotes the set of elements with positive probability. Choosing a value from S and assigning it to a variable, x, is denoted by $x \leftarrow S$.

If a is a probabilistic algorithm, $a(i)$ denotes the probability space defined by running a on input i. If $a_1, \ldots, a_n$ are n probabilistic algorithms and p is an n-ary predicate ($n \in \mathbb{N}$), then $\mathrm{P}(p(x_1, \ldots, x_n) :: x_1 \leftarrow a_1(i_1); \ldots; x_n \leftarrow a_n(i_n))$ denotes the probability that the predicate $p(x_1, \ldots, x_n)$ is true after the result of running a_j on input i_j has been assigned to x_j, for $j := 1, 2, \ldots, n$ (in this order). We also allow assignments from other probability spaces, like $x_j \leftarrow S_j$, in such a formula.

The notion "for k sufficiently large" means "$\exists k_0 \forall k \geq k_0$." The characteristic function of a predicate *pred* is denoted by 1_{pred} and the length of a string by $|\cdot|$. The length of a number is the length of the binary representation of that number. This is denoted by $|\cdot|_2$.

The ring of integers modulo a number n is denoted by $\mathbb{Z}_n$, and its multiplicative group, which contains only the integers relatively prime to n, by $\mathbb{Z}_n^*$.

3. Definition of fail-stop signatures. Briefly, a digital signature scheme is defined by three algorithms (see [20]),

1. a key generator,
2. a method for signing, and
3. a method for testing signatures,

such that if the keys are generated correctly using the key generator, then we have the following:

- If the signer signs a message correctly, everyone who knows the signer's public key accepts the signature.

• A polynomially bounded forger cannot make any signature that passes the signature test.

In a fail-stop signature scheme, a protocol is added that allows the (polynomially bounded) signer to prove to third parties that a forged signature is indeed a forgery. It consists of two more algorithms:

4. a method for constructing proofs of forgery, and
5. a method for verifying proofs of forgery (which everyone who knows the public key can carry out).

A proof of forgery is always noninteractive so that it can subsequently be shown to others, and the system can be stopped in consensus. The proof must satisfy two new security requirements:

• The ability to prove forgeries must work independently of the computing power of potential forgers.

• It must be infeasible for the signer to construct signatures that she can later prove to be forgeries.

It can be shown that these two properties imply security against forgery (see section 3.2); hence this security is omitted in the formal definitions.

Until now, nothing has been said about the generation of the secret and public key. In an ordinary digital signature scheme, the primary purpose of the secret key is to enable the signer to make signatures that nobody else can construct, and this key is therefore chosen by either the signer herself or a key-authentication center trusted by the signer. Since it is equally important that fail-stop signatures cannot be forged, the signer still has to take part in choosing the keys. However, since the signer in these schemes is allowed to repudiate (forged) signatures despite the fact that they pass the public signature test, the recipients of signatures must be sure that the signer cannot disavow her own signatures. It is therefore necessary that the recipients or a center trusted by the recipients also participate in the key generation. In particular, such a center is needed if the recipients are not known at the time of choosing the keys. When the signer's public key has been selected, it will usually be stored in a public directory with accepted integrity.

In the following, we first formalize the definition for the case where the signer and a center generate the keys (section 3.1). Section 3.2 shows that every scheme satisfying this definition is secure against forgery, and section 3.3 discusses how the recipients can participate in the key generation.

3.1. Definition. In this section, we consider the situation where the signer and an entity trusted by the recipients generate the keys. This entity is called the *center* and denoted by C.

It should be noted that we use uniform complexity. In particular, the computational assumptions in the next section are described uniformly. It is not difficult to modify the definitions in order to cope with nonuniform complexity, and reductions in the uniform model are automatically valid in the nonuniform model as well.

The security of a cryptographic scheme is usually determined by a security parameter, which specifies the size of the instances of the hard problem used. A fail-stop signature scheme can be broken by either the signer, if she succeeds in constructing a signature that she can later prove to be a forgery, or a forger, if he succeeds in constructing a false signature that the signer cannot repudiate. Since these two attacks have completely different consequences for the participants, it is natural to define the security parameter of a fail-stop signature scheme as a pair (k, σ) of positive integers, where k measures the (computational) security for the recipients and σ determines

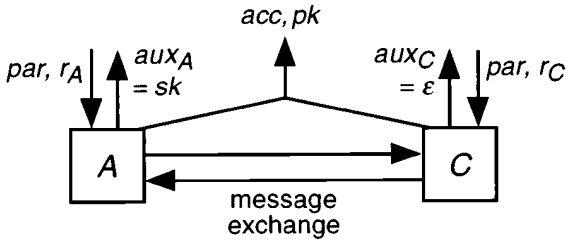


FIG. 1. *Notation used for a correct execution of the key generation G.*

the (unconditional) security for the signer. As is common practice, we shall implicitly assume that these security parameters are represented in unary whenever they are input to an algorithm or protocol.

We allow signing and proving forgeries to be probabilistic and to depend on the history of previously given signatures. Actually, the signatures in our constructions depend only on the *number* of previous signatures. However, the general definition ensures that the lower bounds in section 6 are valid for everything that might reasonably be called a fail-stop signature scheme. We cover all these cases by regarding the random bits as part of the secret key and signing as a deterministic function of the secret key and the sequence of all (previous) messages. This will simplify the notation.

The number of the current message is usually denoted by $i \in \mathbb{N}$.

In addition to σ and k, a fail-stop signature scheme has a parameter $N \in \mathbb{N}$, which is the maximal number of signatures that the signer is willing to construct using the same secret key (hence $1 \leq i \leq N$). As with the security parameters, it is assumed that N is always represented in unary. We often write $par := (k, \sigma, N)$ as an abbreviation of these three parameters.

DEFINITION 3.1. *A* fail-stop signature scheme (*abbreviated* FSS scheme) *with message space* $M \subseteq \{0,1\}^+$ *is a* 5*-tuple* (*G, sign, test, prove, verify*)*, where we have the following:*

- *G is a polynomial-time two-party protocol for generating the keys. The protocol is executed by the signer, A, and C, who both get par as input. Furthermore, each party has a secret random string,* r_A *and* r_C*, respectively. The participants each have a private output channel and there is a* (*broadcast*) *channel for common outputs; see Figure* 1*. The common output is* (acc, pk)*, where* $acc \in \{accept, reject\}$ *and pk is the public key* (*only well defined if* $acc = accept$)*. The common broadcast channel can be realized with a usual broadcast channel if one participant outputs pk and the other outputs acc. We generally denote the outputs on the private output channels of the two participants by* aux_A *and* aux_C*, respectively, and we say that G outputs* (acc, pk, aux_A, aux_C)*. If A is executed correctly,* aux_A *is simply the secret key, denoted by sk. If C is executed correctly,* aux_C *is the empty string,* ε*.*

- *sign is a polynomial-time algorithm that on input the secret key, a message number i, and a message sequence* $\underline{m} = (m_1, \ldots, m_i)$ *from M constructs a signature on* m_i*. Thus sign*(*sk, i,* $\underline{m}$) *denotes the signature on* m_i *if the previously signed messages were* $m_1, \ldots, m_{i-1}$*, and all random bits used are regarded as part of sk* (*and thus originally of* r_A)*. It is called the* correct *signature.*

- *test is a polynomial-time algorithm that on input the public key, a message* $m \in M$*, and a possible signature s on m outputs either ok or notok. If test*$(pk, m, s) =$ *ok, we say that s is an* acceptable *signature on m.*

• *prove is a polynomial-time algorithm that on input the secret key, a message* $m \in M$*, a possible*[1] *signature* s *on* m*, and the history hist of previously signed messages* (*plus their signatures*) *either outputs the string "not a forgery" or a bit string proof* $\in \{0,1\}^*$.

• *verify is a polynomial-time algorithm that on input the public key, a message* $m \in M$*, a possible signature* s *on* m *and a string proof outputs either accept or reject. If the result is accept, the string proof is called a valid proof of forgery.*

This 5*-tuple must satisfy the following basic correctness property* (*security is defined separately*)*. For all* $k, \sigma, N \in \mathbb{N}$*, if* A *and* C *follow the computations prescribed by* G*, each output* $(acc, pk, sk, \varepsilon)$ *of* G *satisfies* $acc =$ *accept and for all message numbers* $i \in \{1, \ldots, N\}$ *and message sequences* $\underline{m} = (m_1, \ldots, m_i)$ *from* M*,*

• *if* $s = sign(sk, i, \underline{m})$*, then* $test(pk, m_i, s) = ok$*, i.e., correct signatures are acceptable.*

The output "*not a forgery*" of *prove* signals that *prove* is not able to construct a valid proof of forgery.

Note that *test* and *verify* are not as general as *sign* and *prove* because we have not allowed them to depend on the history of previous signatures: In general, all signatures may go to different recipients. Hence one cannot assume that a recipient (or a verifier) knows anything about the history. Everything he has to know must therefore be included in the signature. One can make an exception only in applications where all signatures of one signer have a fixed recipient or very few recipients.

Before continuing, we introduce some notation to be used with the key-generation protocol. A signer or center who does not necessarily follow the prescribed protocol is denoted by $\tilde{A}$ or $\tilde{C}$, respectively. $G_{\tilde{A},C}$ and $G_{A,\tilde{C}}$ denote the resulting protocols, and $aux_{\tilde{A}}$ and $aux_{\tilde{C}}$ the private outputs. A cheating participant can also use the input par. We denote by $G_{\tilde{A},C}(par)$ the distribution of the outcome of $G_{\tilde{A},C}$ over the random bits r_C of C and $r_{\tilde{A}}$ of $\tilde{A}$ if the common input is par. Similarly, $G_{A,\tilde{C}}(par)$ denotes the distribution of the outcome of $G_{A,\tilde{C}}$.

DEFINITION 3.2. *An FSS scheme is* secure for the recipients *iff the following holds for all probabilistic polynomial-time algorithms* $\tilde{A}$ *and* $\tilde{A}^*$ (*representing the two steps of an attacking signer*)*: If* C *follows the prescribed protocol in an execution of* G *with* $\tilde{A}$ *and if* $\tilde{A}^*$*, on input the private output* $aux_{\tilde{A}}$ *of* $\tilde{A}$*, constructs a triple* $(m, s,$ *proof*)*, then the probability that proof is a valid proof of forgery tends to zero faster than the inverse of any polynomial in* k*. This probability is over the random* (*uniformly distributed*) *choices of* r_C*,* $r_{\tilde{A}}$ *and the random choices used in* $\tilde{A}^*$*.*

More formally, $\forall \tilde{A}$ *and* $\tilde{A}^*$ (*probabilistic polynomial-time*)*,* $\forall \sigma$*,* N*, and* c*, and for* k *sufficiently large,*

$$\mathrm{P}(acc = accept \wedge verify(pk, m, s, proof) = accept :: \\ (acc, pk, aux_{\tilde{A}}, \varepsilon) \leftarrow G_{\tilde{A},C}(par); (m, s, proof) \leftarrow \tilde{A}^*(aux_{\tilde{A}})) < k^{-c}.$$

(*Remember:* $par = (k, \sigma, N)$.)

We now turn to the definition of the security for the signer. Since we have not required that the signer trusts the center, she should be protected no matter what the center does in the key-generation protocol as long as she follows the protocol herself. Furthermore, the purpose of fail-stop signatures is that the signer should be secure

[1]The algorithm is primarily intended to be used on acceptable signatures. However, it is more convenient for the notation to define it more generally. If s is not an acceptable signature, the output will usually be "*not a forgery.*"

even against arbitrarily powerful forgers. We therefore consider arbitrarily powerful centers $\tilde{C}$. Remember that $[G_{A,\tilde{C}}(par)]$ denotes the set of possible outcomes of the protocol if *par* is given.

DEFINITION 3.3. *Let an FSS scheme with message space $M \subseteq \{0,1\}^+$ and parameters $par = (k, \sigma, N)$ be given, and consider an arbitrarily powerful center $\tilde{C}$. If A and $\tilde{C}$ have executed G and the outcome was $(acc, pk, sk, aux_{\tilde{C}})$ with $acc = accept$, we define the following:*

(a) *The set of possible* histories *is*

$$Hist(sk) := \{((m_1, \ldots, m_j), (s_1, \ldots, s_j)) | 1 \le j \le N \\ \wedge\ (m_i \in M \wedge s_i = sign(sk, i, (m_1, \ldots, m_i)) \text{ for } i = 1, \ldots, j)\}.$$

For a given history hist, let $M(hist)$ denote the set of sign messages in hist.

(b) *The set of* possible secret keys (*from the point of view of an unrestricted forger*) *given $pk, aux_{\tilde{C}}$, and a history hist is*

$$SK_{\tilde{C}}(pk, hist, aux_{\tilde{C}}) \\ := \{sk | (accept, pk, sk, aux_{\tilde{C}}) \in [G_{A,\tilde{C}}(par)] \wedge hist \in Hist(sk)\}.$$

$SK_{\tilde{C}}(pk, hist, aux_{\tilde{C}})$ is equipped with a distribution induced by the random bits of the signer and corresponds to the remaining uncertainty that an attacker has about the real secret key when he tries to choose an "optimal" forgery.

(c) *The set of* successful forgeries *given pk and a history hist is*

$$Forg(pk, hist) := \{f = (m, s) | m \in M \backslash M(hist) \wedge test(pk, m, s) = ok\},$$

i.e., the set of acceptable signatures on messages not contained in the history.

(d) *A value $f = (m, s) \in Forg(pk, hist)$ is a* provable forgery *after a history hist, abbreviated provable$(sk, pk, hist, f)$, iff*

$$verify(pk, m, s, prove(sk, m, s, hist)) = accept,$$

i.e., if applying prove to it yields a valid proof of forgery.

Intuitively, the following definition says that no matter what an arbitrarily powerful center does during the key generation, no matter what messages A signs, and no matter what message and signature the forger selects as a forgery given all this knowledge, A can repudiate the forged signature with high probability. The probability is over the possible secret keys (among which, roughly speaking, the forger must guess which one A really has). For the first similar definition of security against an unrestricted adversary, see [42].

There are two possible sources for a small error probability: One is during key generation, where A might be tricked into accepting a bad key pair; the other is that a forger happens to find exactly A's correct signature.

DEFINITION 3.4. *Let an FSS scheme be given.*

(a) *For any given parameters $k, \sigma, N \in \mathbb{N}$ and any arbitrarily powerful center $\tilde{C}$, define a set $Good_{\tilde{C}}$ of "good" outcomes of the key generation as follows:*

$$(sk, pk, aux_{\tilde{C}}) \in Good_{\tilde{C}} :\Leftrightarrow \forall hist \in Hist(sk), \forall f \in Forg(pk, hist):$$

$$\mathrm{P}(provable(sk', pk, hist, f) :: sk' \leftarrow SK_{\tilde{C}}(pk, hist, aux_{\tilde{C}})) \ge 1 - 2^{-\sigma}.$$

Thus an outcome is called good if it guarantees that after any history, an unrestricted forger still has so much uncertainty about the secret key that his forgery will be provable with very high probability.

(b) *The FSS scheme is* secure for the signer *iff* $\forall\sigma$, $\forall par = (k, \sigma, N)$, *and for any arbitrarily powerful center* $\tilde{C}$,

$$\mathrm{P}((sk, pk, aux_{\tilde{C}}) \notin Good_{\tilde{C}} \wedge acc = accept :: (acc, pk, sk, aux_{\tilde{C}}) \leftarrow G_{A,\tilde{C}}(par)) \leq 2^{-\sigma}.$$

(c) *If both A and C follow G, each output is good, i.e.,* $(sk, pk, \varepsilon) \in Good_C$.

DEFINITION 3.5. *A* secure FSS scheme *is an FSS scheme that is secure for both the signer and the recipients.*

3.2. Relation to ordinary digital signatures: Security against forgery. In the above definition of secure fail-stop signatures, we have not explicitly demanded that it be difficult for a forger to construct acceptable signatures. Clearly, a signature scheme is useless if it is easy to make forgeries, even if they can be repudiated. We now show that Definition 3.5 actually implies that forging is hard for polynomially bounded enemies. More precisely, we show that existential forgery is infeasible even after an adaptive chosen-message attack (see [20]).

In an adaptive chosen-message attack against a signature scheme with security parameter l (which is used as an input in key generation) a forger F does the following, given the public key as an input:

1. Repeat a polynomial number of times (in l): Generate (in some way) a message m and receive the correct signature on m from the signer.
2. Output a pair (m', s'). (This should be a new message–signature pair.)

DEFINITION 3.6. *A signature scheme with security parameter* l *is* secure against an adaptive chosen-message attack *iff for all* $c > 0$ *and for all probabilistic polynomial-time forgers* F *as above, the probability that* F *outputs a pair* (m', s') *such that* m' *is different from all messages chosen in step* 1 *and* s' *is an acceptable signature on* m' *is less than* l^{-c} *for* l *sufficiently large.*

The probability is over the random bits used in the key generation, the random bits of F*, and the random choices of the signatures, if the signing algorithm is probabilistic.*

We now consider the security of fail-stop signatures against a forger who has not participated in the key generation.

THEOREM 3.1. *A secure FSS scheme is secure against an adaptive chosen-message attack by forgers who do not participate in G, i.e., who have only* par *and the result pk of a correct execution of G as inputs; the parameter k of the FSS scheme plays the role of the parameter l in Definition* 3.6.

Proof. Let a secure FSS scheme be given as in Definitions 3.1 and 3.5. Assume further that there exists a probabilistic polynomial-time forger F as above that on input (par, pk) outputs a pair of a new message and an acceptable signature (m', s') with probability $p(par)$. This probability is over all the random bits used in key generation and in F. Since F runs in polynomial time, a signer who tries to cheat a recipient can use the same algorithm:

1. Execute G correctly with C. This yields a key pair (sk, pk).
2. Execute F alternating with the real signing algorithm to obtain a history *hist* and a pair (m', s').
3. Use *prove* to compute a proof of forgery for (m', s'), given sk and *hist*.

Roughly, the argument is as follows: On one hand, the security for the signer against a forger with algorithm F implies that step 3 leads to a valid proof of forgery

with nonnegligible probability. On the other hand, this means that when the signer alone carries out the whole algorithm (i.e., step 1 as $\tilde{A}$ and steps 2 and 3 as $\tilde{A}^*$), it contradicts the security for the recipients.

We now proceed more formally. The history and message–signature pair generated in step 2 have exactly the same distribution as those that the forger would have constructed (i.e., the distribution used in Definition 3.6). This implies two things: First, (m', s') is a successful forgery with probability $p(par)$. Second, whenever this is the case, the signer can repudiate the forgery, i.e., step 3 yields a valid proof of forgery with probability at least $q := 1 - 2^{-\sigma}$, because $(sk, pk, \varepsilon) \in \mathit{Good}$ (see Definition 3.4(a, c)). Here q is the a posteriori probability over the possible secret keys, given any history. We omit the tedious details needed to combine these different types of probabilities formally before one obtains the expected result:

With probability at least $p(par)(1 - 2^{-\sigma}) \geq p(par)/2$, a signer executing steps 1, 2, and 3 obtains an acceptable signature together with a valid proof of forgery for it. Since the FSS scheme is secure for the recipients, this implies that $p(par)$ must be negligible as a function of k. This proves the theorem. □

This theorem can be interpreted as saying that fail-stop signatures are a stronger notion of signatures than that defined in [20] in two ways. The first way is described in the following theorem.

THEOREM 3.2. *Every secure fail-stop signature scheme can be used to construct an (equally efficient) secure digital signature scheme in the sense of* [20].

Proof. The main reason that a fail-stop signature scheme itself does not fulfill the definition in [20] is the interactive key generation. However, one can construct such a scheme as follows: On input (k, N), the signer executes both A and C in the key generation with $\sigma := 1$. The algorithms *sign* and *test* remain the same, and *prove* and *verify* are omitted. Theorem 3.1 implies that this is a secure signature scheme in the sense of [20]. □

As the second way, section 3.3 shows how Theorem 3.1 can be strengthened for the fail-stop signature scheme itself so that unconditional security for the signer is combined with security against an adaptive chosen-message attack by anyone, i.e., without even trusting a center.

3.3. Fail-stop signatures with known recipients. In the definitions in section 3.1, the keys were generated by the signer and a center trusted by the recipients. This section briefly discusses how fail-stop signature schemes work if the recipients themselves participate in the key generation. More precisely, one should distinguish recipients and risk bearers. Risk bearers are those who have a disadvantage if a proof of forgery is accepted and thus a signature becomes invalid. For instance, a recipient might have an insurance that covers his losses if signatures that he had accepted are proved to be forgeries. Then the insurance company—and not the recipient—is the risk bearer and has to trust the key generation process. We continue to say recipient for simplicity.

One recipient. In this case, we can simply replace the trusted center by the recipient. Note that a fail-stop signature scheme with only one recipient is not useless because everyone who knows the public key can still test the signatures and verify the proofs of forgery. The only restriction is that only the intended recipient is guaranteed that the signer cannot repudiate her own signatures. Such a scheme can, e.g., be applied in electronic payment systems for signing the customers' requests to the bank.

Many recipients. The definition of FSS schemes with many known recipients also follows the previous definitions very closely. In such a scheme, all the recipients participate in a key-generation protocol with the signer. Each participant has the input *par* and a secret random string, and the common output is either *reject* or $(accept, pk)$. (We assume that all recipients are satisfied with the same parameter k for their computational security.) The signer gets the secret key corresponding to pk as private output. The security for the signer can be defined as before, and the security for the recipients is essentially defined by requiring Definition 3.2 for each of them. In other words, if a recipient follows the key-generation protocol correctly, he is assured that a (polynomially bounded) signer cannot repudiate her own signatures, even if the signer and the remaining recipients cooperate.

We now show how FSS schemes with many known recipients can be constructed from FSS schemes in the sense of section 3.1. In some FSS schemes, the only task of the center in the key-generation protocol is to select a random string. In these schemes, it is quite easy to replace the center by many recipients because they only have to perform a multiparty coin-flipping protocol.

In the general case, one can apply a protocol for general secure multiparty computations in a certain way. For some details, see [35, 34]. However, in the present state of cryptography, the following simpler method is much more efficient. Its disadvantage is that the keys are long and thus signing and testing are relatively inefficient.

Construction 3.1 (many keys). Let R be the number of recipients. Each recipient executes protocol G with the signer once. This results in R key pairs, $(sk_j, pk_j)_{j=1,\ldots,R}$. The secret key of the signer is defined as $(sk_1, \ldots, sk_R)$ and the public key as $(pk_1, \ldots, pk_R)$. All signatures as well as proofs of forgery consist of R parts, one for each of the R key pairs. Note that everyone can test all parts of each signature or proof of forgery. ◇

The security for the jth recipient is guaranteed because if he carries out his execution of G correctly, it is computationally infeasible to compute valid proofs of forgery for pk_j. The security for the signer is guaranteed because given a successful forgery, she can compute proofs of forgery for it for each key pair with high probability.

As an immediate consequence of Theorem 3.1, this scheme is secure against an adaptive chosen-message attack if at least one of the recipients executes G correctly with the signer—even if the forger cooperates with the remaining $R - 1$ recipients. We can even achieve security if the forger may cooperate with all recipients by letting the signer take part in the key generation in the role of a recipient as well, i.e., she generates an additional key pair (sk_{R+1}, pk_{R+1}) all on her own, which is then treated like any other key pair.

4. Constructions of fail-stop signature schemes. This section first presents a general construction of a fail-stop signature scheme, which is subsequently used in actual instantiations. We describe two such instantiations based on the assumptions that it is hard to compute discrete logarithms and to factor integers, respectively. These are the two best-known concrete computational assumptions used in cryptography, and they are therefore well investigated and fairly trustworthy.

The constructions as described in this section allow only one message to be signed; hence they are called one-time signature schemes. Section 5 presents extensions for signing an arbitrary number of messages.

4.1. Bundling homomorphisms. In order to present our general construction of fail-stop signatures, we first define bundling homomorphisms. They are a special type of cryptographic hash functions and may have other uses in cryptography. Briefly,

a bundling homomorphism h is a homomorphism $h : G \to H$ between two Abelian groups $(G, +, 0)$ and $(H, \cdot, 1)$ that satisfies the following:

- Every image $h(x)$ has at least 2^τ preimages. (The bundling homomorphism is said to be of degree 2^τ.)
- It is infeasible to find collisions, i.e., two different elements that are mapped to the same value by h.

In order to make the second of these requirements precise, we have to consider a family of such functions. The individual functions of this family are indexed by a key K. The key is chosen depending on two security parameters: τ, which determines the bundling degree, and k, which measures the computational security against collision-finding. The parameters τ and k are part of the index to this function, but we only write K instead of (K, τ, k). This leads to the following definition.

DEFINITION 4.1. *A* family of bundling homomorphisms *is a quadruple* $(g, h, \mathcal{G}, \mathcal{H})$, *where we have the following:*

- g, *the key generator, is a probabilistic polynomial-time algorithm that on input parameters* $k, \tau \in \mathbb{N}$ *outputs a value* K. *Let* $\mathcal{K}$ *be the set of all possible keys, i.e., the union of the sets* $[g(k, \tau)]$ *for all* $k, \tau \in \mathbb{N}$.
- $\mathcal{G}$ *and* $\mathcal{H}$ *are families of Abelian groups, one for each key. More formally,* $\mathcal{G} = (G_K, +, 0)_{K \in \mathcal{K}}$ *and* $\mathcal{H} = (H_K, \cdot, 1)_{K \in \mathcal{K}}$.
- h *is a polynomial-time algorithm that on input* $K \in \mathcal{K}$ *and* $x \in G_K$ *outputs a value* $z \in H_K$. *The restriction of* h *to a particular key* K *is abbreviated as* h_K.

This quadruple must satisfy the following properties:

(a) *Each* h_K *is a group homomorphism from* $(G_K, +, 0)$ *to* $(H_K, \cdot, 1)$.

(b) *For all* $k, \tau \in \mathbb{N}$, $K \in [g(k, \tau)]$, *each* $z \in h_K(G_K)$ *has at least* 2^τ *preimages under* h_K.

(c) *The family is collision-resistant: For every* $c > 0$ *and for every probabilistic polynomial-time algorithm* $\tilde{A}$, *the probability that* $\tilde{A}$ *on input* $K \in [g(k, \tau)]$ *outputs a pair* (x, x') *such that* $x \neq x'$ *and* $h_K(x) = h_K(x')$ *is less than* k^{-c} *for* k *sufficiently large. More precisely,* $\forall \tau \forall c \, \exists k_0 \forall k \geq k_0$,

$$\mathrm{P}(h_K(x) = h_K(x') \wedge x \neq x' :: K \leftarrow g(k, \tau); (x, x') \leftarrow \tilde{A}(K)) < k^{-c}.$$

(*A more common name for collision-resistant is collision-free, but collision-resistant emphasizes the computational aspect better.*)

Additionally, there must be polynomial-time algorithms (*which do not need explicit names in the following*) *that on input* K

- *compute the operations in the two groups* $(G_K, +, 0)$ *and* $(H_K, \cdot, 1)$,
- *select elements of* G_K *uniformly at random, and*
- *test membership in* H_K *and* G_K.

Note that k_0 depends on τ in the definition of collision resistance. Actually, all of our constructions satisfy the stronger requirement where k_0 is independent of τ, as long as τ is polynomial in k (because an arbitrarily long input τ would give the collision finder time more than polynomial in k).

4.2. The general construction. The general construction yields a rather special case of the definition of fail-stop signature schemes. First, only one message is signed. Second, the key generation of this scheme (like all previous constructions in the literature) is quite simple. These two properties are now defined formally because the constructions to sign many messages in section 5 can be based on any FSS scheme with these properties.

DEFINITION 4.2. *A* one-time fail-stop signature scheme with prekey *is defined like a general FSS scheme in Definition* 3.1*, except that the parameter* N *is* 1 *and the key-generation protocol* G *is of a special, simpler form: It is constructed from a triple* $(gen_C, (P,V), gen_A)$*, where we have the following:*

- gen_C *is a probabilistic polynomial-time algorithm that on input par generates values prek (the* prekey*) and* w *(called witness).*
- (P,V)*, the prekey verification protocol, is a polynomial-time two-party protocol where* P *gets the input* $(par, prek, w)$ *and* V *only gets* $(par, prek)$*. As a result,* V *outputs accept or reject. Correct outputs of the prekey generation should always be accepted, i.e., if* $(prek, w) \in [gen_C(par)]$*, the output of* V *should be accept. The most efficient special case is where* P *does not do anything, i.e.,* V *decides locally whether or not to accept prek.*
- gen_A *is a probabilistic polynomial-time algorithm that on input a prekey prek outputs a key pair* (sk, pk)*. It is called the main key-generation algorithm.*

The protocol G *is constructed from these subprotocols as follows: First, the center* C *executes* gen_C *and publishes the resulting prekey. Next,* (P,V) *is executed for this prekey by the center* (P) *and the signer* (V)*, where the center has* w *as an additional input. (The purpose of this step is to prove some desired property of prek to the signer.) Finally, the signer* A *carries out* gen_A *on input prek. We say that she generates a key pair based on the prekey prek. The public key in the sense of Definition* 3.1 *is the pair* $(prek, pk)$*. However, we often omit prek in the notation because it is clear from the context.*

The signer, using V, may also accept if *prek* is not a possible outcome of the correct prekey generation; we only use (P,V) to prove those properties of *prek* to the signer that are necessary for the signer's security. This is often much more efficient than proving correct generation. Note that (P,V) is similar to an interactive proof system [19], but our prover is polynomial-time and needs the witness w.

Schemes with prekey have the following advantage if there are several signers. The center can publish the prekey without knowing which signers will take part. Every signer carries out the interactive proof with the center once and can then base many successive secret keys on this prekey without further interaction with the center. This is exploited in section 5. The security is not weakened if many signers base their key pairs on the same prekey because of the following:

- If this led to signers being able to repudiate their own signatures, one cheating signer alone could generate many key pairs based on that prekey and experiment with them locally until she could repudiate a signature, and only then would she publish the corresponding public key.
- If this allowed forgers to make unprovable forgeries with nonnegligible probability (over the choice of the key pairs of all the signers), the probability for each key pair would also be nonnegligible because all key pairs are identically distributed.

We now describe a framework for constructing a one-time FSS scheme with prekey from a family of bundling homomorphisms. A few parameters are left open; they depend on the choice of the family of bundling homomorphisms. We then present two theorems that reduce the security of the scheme to one property of the bundling homomorphisms and those parameters. This property will be proved and the parameters specified in the instantiations in sections 4.3 and 4.4.

Since the message number i is always 1, it will be omitted in the notation, and instead of a message sequence $\underline{m}$ of length 1, a message m is written.

Construction 4.1 (general construction). Let a family of bundling homomor-

phisms with a key generator g be given. Then the various components of a one-time fail-stop signature scheme with prekey are defined as follows:

- Key generation: Let security parameters k and σ be given. The parameter τ for the degree of the bundling homomorphisms is a function of σ, which will be specified later.
- Prekey generation gen_C: The center computes $K \leftarrow g(k, \tau)$ and publishes it, i.e., $prek := K$. This corresponds to choosing one homomorphism h_K of the family. Henceforth, let $h := h_K$, $G := G_K$, and $H := H_K$. Instead of g, an algorithm g' may be used that outputs K with the same probability distribution as g but also outputs a witness w.
- Prekey verification (P, V): The signer must be assured that K is a possible outcome of $g(k, \tau)$, or at least that h is a homomorphism and satisfies Definition 4.1(b). The choice of a predicate between these two extremes that can be proved most efficiently depends on the choice of the family of bundling homomorphisms. A prekey K is called *good* if it fulfills this predicate; otherwise, it is called *bad*. If possible, this protocol is a local test by the signer. Otherwise, it is a zero-knowledge proof from the center to the signer (see [19, 18]—except we require P to be polynomial-time and to use a witness w). The probability that the signer accepts the proof for a bad prekey K must be at most $2^{-\sigma}$ for each K.[2]
- Main key generation gen_A: The signer generates her secret key $sk := (sk_1, sk_2)$ by choosing sk_1 and sk_2 randomly in G, and she computes her public key $pk := (pk_1, pk_2)$, where $pk_i := h(sk_i)$ for $i = 1, 2$.
- The message space M is a subset of $\mathbb{Z}$ depending on the choice of the prekey.
- Signing: The correct signature on a message m in the message space is

$$sign(sk, m) := sk_1 + m\, sk_2.$$

(Multiplication with elements of $\mathbb{Z}$ is, as usual, defined by repeated addition.)

- Test: The algorithm *test*, which determines whether a signature $s \in G$ is acceptable, is defined so that $test(pk, m, s) = ok$ iff $pk_1 \cdot pk_2^m = h(s)$.
- Proof of forgery: Given an acceptable signature $s' \in G$ on m such that $s' \neq sign(sk, m)$, the signer computes $s := sign(sk, m)$ and $proof := (s, s')$.
- Verification of proof of forgery: Given a pair (x, x'), verify that x and x' are elements of G, that $x \neq x'$, and that $h(x) = h(x')$. $\diamond$

This concludes the description of the general construction. The following theorem shows that any instantiation of this construction works and that it is secure for the recipients.

THEOREM 4.1. *For any family of bundling homomorphisms, and with any choice of the parameters that have been left open, the general construction has the following properties:*

(a) *Correct signatures pass the test* (*i.e., Definition* 3.1 *is fulfilled*).

(b) *A polynomially bounded signer cannot construct a signature and a valid proof that it is a forgery* (*i.e., Definition* 3.2 *is fulfilled*).

(c) *If s^* is an acceptable signature on m^* and $s^* \neq sign(sk, m^*)$, the signer obtains a valid proof of forgery* (*this is a step towards fulfilling Definition* 3.4).

(d) *If all values $(sk, (K, pk), aux_{\tilde{C}})$ where K is good are contained in $Good_{\tilde{C}}$, the scheme is secure for the signer* (*another step towards fulfilling Definition* 3.4).

[2] Note that the definition of zero-knowledge proofs from [19] only requires that the probability that a bad value is accepted decreases faster than the inverse of any polynomial, but the actual proofs in [18] have the strictly exponential decrease that we need.

Proof. Part (a) follows from the fact that h is a homomorphism:

$$h(s) = h(sk_1 + m\,sk_2) = h(sk_1) \cdot h(sk_2)^m = pk_1 \cdot pk_2^m.$$

Part (c) is a trivial consequence of the fact that both s and s^* pass the test. Part (d) follows immediately from the requirement that we have made on the error probability of the prekey verification. For part (b), note that a proof of forgery is exactly a collision of the bundling homomorphism h_K, where K is chosen by the center in gen_C with the correct probability distribution. Hence part (b) follows immediately from the collision resistance of the bundling homomorphisms, except that we have to show that the zero-knowledge proof does not make it easier for the signer to find collisions, which is intuitively clear (although a formal proof is lengthy; see [34]). □

This theorem shows that the general construction is secure for the recipients and that it is also secure for the signer if an arbitrarily powerful forger cannot guess a correct signature $sign(sk, m^*)$, except with a very small probability, whenever the prekey is good. In order to estimate the probability with which a forger can find such a signature, we first note that given a public key, at least $2^{2\tau}$ secret keys are possible. Given a correct signature on another message m, the forger has more information about sk, but Theorem 4.2 gives a condition under which this information is not sufficient to guess the correct signature on m^* with too high a probability.

THEOREM 4.2. *Consider Construction* 4.1. *Let parameters k and σ, a good prekey K, and two messages $m \neq m^*$ from the corresponding message space be given. Let*

$$T := \{d \in G | h(d) = 1 \wedge (m^* - m)d = 0\}.$$

Then for all key pairs $(sk, pk) \in [gen_A(K)]$ and all values $s^* \in G$ (a forgery) satisfying $test(pk, m^*, s^*) = ok$, the probability that $s^* = sign(sk, m^*)$, given $s := sign(sk, m)$, is at most $|T|/2^\tau$, i.e., for any center $\tilde{C}$,

$$\mathrm{P}(s^* = sign(sk', m^*) :: sk' \leftarrow SK_{\tilde{C}}(pk, (m, s), aux_{\tilde{C}})) \leq |T|/2^\tau.$$

Note that the probability in this theorem resembles Definition 3.4(a).

Proof. Since K is good, h is at least a homomorphism and satisfies Definition 4.1(b). Note that $aux_{\tilde{C}}$ can contain additional information about K but that the only information that the signer divulges about sk is pk and a correct signature s on one message m. The set of possible keys given this information is

$$\begin{aligned} SK_{\tilde{C}} &= \{(sk'_1, sk'_2) \in G \times G | h(sk'_1) = pk_1 \wedge h(sk'_2) = pk_2 \wedge sk'_1 + m\,sk'_2 = s\} \\ &= \{(s - m\,sk'_2, sk'_2) | h(sk'_2) = pk_2\} \end{aligned}$$

because h is a homomorphism and $s = sk_1 + m\,sk_2$. The size of $SK_{\tilde{C}}$ is therefore at least 2^τ. We must now find out how many of these keys satisfy $s^* = sign(sk', m^*)$, i.e.,

$$(*) \qquad sk'_1 + m^*\,sk'_2 = s^*.$$

Since we only consider keys in $SK_{\tilde{C}}$, we can replace sk'_1 by $s - m\,sk'_2$. Hence $(*)$ is equivalent to

$$(m^* - m)sk'_2 = s^* - s.$$

This equation might be unsolvable, but if there is any solution sk_2'', the set of all solutions in $SK_{\tilde{C}}$ is

$$\{(s - m\,sk_2', sk_2') | h(sk_2') = h(sk_2'') \wedge (m^* - m)(sk_2' - sk_2'') = 0\}.$$

Hence the number of solutions is $|T|$ (where d corresponds to the difference $sk_2' - sk_2''$). Since sk is uniformly distributed in Construction 4.1, all elements of $SK_{\tilde{C}}$ are equally probable, and the attacker is successful with probability at most $|T|/|SK_{\tilde{C}}| \leq |T|/2^\tau$, as claimed. □

COROLLARY. *Theorem* 4.2, *together with Theorem* 4.1(c, d), *shows that the general construction is secure for the signer* (*as in Definition* 3.4) *if* τ *is chosen so that* $|T|/2^\tau \leq 2^{-\sigma}$, *i.e.,* $\tau \geq \sigma \log_2(|T|)$.

Consequently, we must find the maximal size of

$$T_{m'} := \{d \in G | h(d) = 1 \wedge \mathrm{ord}_G(d) | m'\}$$

over all possible differences m' of two messages. The size of this set depends on the chosen family of bundling homomorphisms.

Note that the collision resistance of h was only needed in Theorem 4.1(b), i.e., to ensure the security for the recipients. This is the reason why the center need not prove to the signer that h has this property.

4.3. A scheme based on discrete logarithms. We now construct a family of bundling homomorphisms for which finding collisions is equivalent to computing discrete logarithms.

Let p and q be large primes such that q divides $p - 1$, and let H_q be the unique subgroup of $\mathbb{Z}_p^*$ of order q. (Recall the theorem that all groups $\mathbb{Z}_p^*$ are cyclic and of order $p - 1$.) Remember that all elements of H_q except 1 are generators. We shall assume that it is hard to compute discrete logarithms in H_q (as already sketched in [12]). For elements a and b of H_q, where $a \neq 1$, the discrete logarithm $\log_a(b)$ is defined as the number $e \in \{0, \ldots, q - 1\}$ with $a^e = b$.

Assumption DL. For all probabilistic polynomial-time algorithms D, for all $c \in \mathbb{N}$, and for k sufficiently large, the probability that D, on input two primes p and q, where q is a k-bit prime dividing $p - 1$ and $q > (p - 1)/q$, and two generators a and b of H_q, outputs $\log_a(b)$ is less than k^{-c}. The probability is over the random bits used by D and the uniform random choices of p, q, a, and b with the given constraints. ◇

The scheme presented below actually works for any groups of prime order, but for the sake of concreteness, we shall assume the above setup.

Construction 4.2 (discrete logarithm homomorphisms).

- Key generator g: On input k and τ, it chooses primes p and q as above with $|q|_2 = \max(k, \tau)$ and two random generators a and b of H_q. It outputs the key $K := (p, q, a, b)$.
- Families of groups: Define

$$G_q := \mathbb{Z}_q \times \mathbb{Z}_q$$

with pairwise addition (modulo q). With a slight misuse of the notation, we abbreviate $G_{(p,q,a,b)} =: G_q$ and $H_{(p,q,a,b)} =: H_q$.

- The homomorphisms are defined as

$$h_{(p,q,a,b)}\colon G_q \to H_q; \qquad h_{(p,q,a,b)}(x, y) := a^x b^y.$$

The corresponding algorithm h is clear. ◇

Similar constructions were first used in [6].

THEOREM 4.3. *Under Assumption* DL*, Construction* 4.2 *is a family of bundling homomorphisms.*

Proof. It is easy to see that all necessary operations are efficiently computable. In particular, g generates q first and then searches for a prime p among the numbers of the form $tq+1$. Now the properties from Definition 4.1(a–c) must be verified.

(a) Obviously, each $h_{(p,q,a,b)}$ is a group homomorphism. (Recall that the order of H_q is q.)

(b) For every $z \in H_q$, there are exactly q elements (x, y) of G_q that h maps to z: For each x, there is exactly one y with $b^y = za^{-x}$ because b is a generator.

(c) Assume that a probabilistic polynomial-time algorithm $\tilde{A}$ could compute collisions of h with nonnegligible probability. We then construct an algorithm D that on input (p, q, a, b) computes the discrete logarithm of b with respect to a as follows: First, D runs $\tilde{A}$, and if $\tilde{A}$ outputs a collision, i.e., $(x, y) \neq (x', y')$ with $a^x b^y = a^{x'} b^{y'}$, then D computes $\log_a(b)$ as $(x'-x)(y-y')^{-1} \bmod q$. (Note that $y = y'$ is impossible.) D is successful with the same probability as $\tilde{A}$ and almost equally efficient. Hence it contradicts Assumption DL. □

This theorem implies that we can construct an FSS scheme that is secure for the recipients under the discrete logarithm assumption. Before we go into the details of the security for the signer, the resulting scheme is described, and the parameters that were left open in the general construction are fixed.

Construction 4.3 (the Discrete Logarithm Scheme).

- Key generation: On input k and σ, the parameter τ for the degree of the bundling homomorphisms is set to σ.
- Prekey generation: The center selects primes p and q and generators a and b (with the algorithm g) and publishes them.
- Prekey verification: The signer can verify by herself that p and q are primes and that a and b are generators of H_q by verifying that their order is q. Hence key generation is extremely simple in this scheme.
- Main key generation: The secret key consists of four numbers x_1, y_1, x_2, and y_2 between 0 and $q-1$ (more precisely $(x_1, y_1), (x_2, y_2) \in G_q$), and the corresponding public key is the pair (pk_1, pk_2) with

$$pk_1 := a^{x_1} b^{y_1} \quad \text{and} \quad pk_2 := a^{x_2} b^{y_2}.$$

- The message space is defined as $\{0, 1, \ldots, q-1\}$.
- Signing: The correct signature on a message m from this space is

$$(x, y) := (x_1, y_1) + m(x_2, y_2) = (x_1 + m\ x_2, y_1 + m\ y_2).$$

- Test: A pair (x, y) is an acceptable signature on the message m iff

$$a^x b^y = pk_1 pk_2^m.$$

- Proofs of forgery and their verification: According to the general construction, a proof of forgery is a collision of h. Hence such a proof consists of four numbers. However, the signer can just as well show $e := \log_a(b)$ as a proof of forgery because this is equivalent to the other proof but shorter and easier to verify. (e yields a collision $a^e b^0 = a^0 b^1$.)

We now return to proving that this scheme is also secure for the signer.

THEOREM 4.4. *The Discrete Logarithm Scheme is secure for the signer.*

Proof. According to the end of section 4.2, we have to find the maximal size of the set

$$T_{m'} = \{d \in G_q | h(d) = 1 \wedge \mathrm{ord}(d) | m'\}$$

for all values of m' between 1 and $q-1$ (m' is the difference between two different legal messages; negative values need not be considered separately), given that the prekey is good. Thus q is prime, and the order of all nonzero elements of G_q is q. Hence $(0,0)$ is the only element of $T_{m'}$.

Together with the corollary to Theorem 4.2, this implies that it suffices to choose $\tau := \sigma$ in the proposed scheme, as we did in Construction 4.3. □

The choice of $\tau := \sigma$ means that $|q|_2$ is chosen as $\max(k, \sigma)$. For reasonable parameters k and σ, this means $|q|_2 = k$.

We conclude this section with a short evaluation of the efficiency of the proposed scheme:

- Signing requires two multiplications modulo q.
- Testing a signature requires less than two exponentiations modulo p. This is because the recipient can test the signature (x, y) by computing $a^x b^y\, pk_2^{-m}$ and verifying that it equals pk_1. This corresponds to between k and $2k$ modular multiplications, depending on the message, with a suitable exponentiation algorithm.
- The length of the secret key (for one message) is $4k$.
- The length of the public key is $2|p|_2$ bits.
- The length of a signature on a k-bit message is $2k$.

4.4. A scheme based on factoring. This section shows how the general construction can be used to construct fail-stop signatures where the security for the recipients relies on the intractability of factoring. The family of bundling homomorphisms was defined in [4], using ideas from [20, 17]. A similar homomorphism was already used in [2]. We denote the group of quadratic residues modulo an integer n by $QR_n := \{x \in \mathbb{Z}_n^* \,|\, \exists w\colon\ w^2 \equiv x \bmod n\}$. The basic assumption of this scheme is that it is hard to factor large integers.

Assumption F (see [20]). For all probabilistic polynomial-time algorithms F, for all $c \in \mathbb{N}$, and for k sufficiently large, the probability that F, on input a k-bit integer n which is the product of two primes p and q of equal length and with $p \equiv 3$ and $q \equiv 7 \bmod 8$, outputs one of the factors is less than k^{-c}. The probability is over the random bits used by F and the uniform random choice of n with the given constraints. ◇

Construction 4.4 (factoring homomorphisms).

- Key generator g: On input k and τ, it chooses a k-bit integer $n = pq$ such that p and q are primes with $p \equiv 3$ and $q \equiv 7 \bmod 8$. It outputs $K := (\tau, n)$.
- Families of groups: We slightly abbreviate the groups for a key $K = (\tau, n)$ as

$$H_n := (\pm QR_n)/\{1, -1\} \quad \text{and} \quad G_{\tau,n} := \mathbb{Z}_{2^\tau} \times H_n.$$

The group operation in H_n is induced by multiplication modulo n. Each element of H_n, which is a coset $\{x, -x\}$, will be identified with its smaller member, i.e., a number between 0 and $n/2$. The reason for using the factor group H_n instead of QR_n is that membership in H_n can be tested efficiently: A number between 0 and $n/2$ belongs to H_n iff its Jacobi symbol is $+1$. (Hence it is also easy to test membership in $G_{\tau,n}$.) The operation in $G_{\tau,n}$ is defined by

$$(a, x) \circ (b, y) := ((a + b) \bmod 2^\tau, xy4^{(a+b)\mathrm{div}2^\tau}),$$

and the unit element of $G_{\tau,n}$ is $(0,1)$.

- The homomorphism $h_{\tau,n}$ mapping $G_{\tau,n}$ to H_n is defined by

$$h_{\tau,n}((a,x)) := \pm(4^a x^{2^\tau}),$$

i.e., either $4^a x^{2^\tau}$ or $-4^a x^{2^\tau}$, depending on which of them is smaller than $n/2$. The corresponding (unoptimized) probabilistic polynomial-time algorithm h is clear. ◇

THEOREM 4.5.

(a) *Under Assumption* F*, Construction* 4.4 *is a family of bundling homomorphisms.*

(b) *For any* $\tau \in \mathbb{N}$ *and any odd* $n \in \mathbb{N}$ *(i.e., not only for those chosen according to Construction* 4.4*),* $h_{\tau,n}$ *is a homomorphism satisfying Definition* 4.1(b)*.*

(c) *If* $n = p^r q^s$*, where* p *and* q *are primes with* $p \equiv 3$ *and* $q \equiv 7 \bmod 8$ *and* r *and* s *are odd, then for any* $a \in \mathbb{Z}_{2^\tau}$ *and* $z \in H_n$*, there exists exactly one* $x \in H_n$ *so that* $h_{\tau,n}((a,x)) = z$*. (The reason we consider numbers* n *of this more general form is that we want to use the fairly efficient zero-knowledge protocol from* [21]*; see below.)*

Proof. See [4]. The last claim is only proved for $r = s = 1$, but the same proof also works in the more general case. □

Given this family of bundling homomorphisms, we can construct a fail-stop signature scheme called the *Factoring Scheme*. We only have to fill in the three parameters that were left open in Construction 4.1:

- The message space is $\{0, \ldots, 2^\rho - 1\}$ for any $\rho \in \mathbb{N}$.
- The parameter τ for the bundling degree is computed as $\tau := \sigma + \rho$.
- A prekey n is good if it is of the form defined in Theorem 4.5(c). By Theorem 4.5(b, c), such prekeys fulfill the conditions required in Construction 4.1. The zero-knowledge proof for this predicate is as follows:

1. The center, who has chosen n and therefore knows p and q, uses the zero-knowledge protocol from [21] to prove to the signer that $n = p^r q^s$, where p and q are primes and congruent to 3 modulo 4 and r and s are odd. This proof must be executed such that the probability of cheating the signer into accepting an incorrect n is at most $2^{-\sigma}$.
2. The signer verifies on her own that $n \equiv 5 \bmod 8$ to exclude the case $p \equiv q \bmod 8$.

Given all this, it is straightforward to write down the details of the Factoring Scheme in the same manner as in Construction 4.3. The following theorem completes the security considerations.

THEOREM 4.6. *The Factoring Scheme is secure for the signer.*

Proof. According to the corollary to Theorem 4.2, it only remains to show $|T_{m'}| \leq 2^\rho$ for all m' whenever the prekey is good because then

$$|T_{m'}|/2^\tau \leq 2^{-\sigma}.$$

Note that in $G_{\tau,n}$,

$$(a,x)^{m'} = (0,1) \Rightarrow m'a \bmod 2^\tau = 0 \Rightarrow \mathrm{ord}(a) | m'.$$

Hence

$$T_{m'} \subseteq \{(a,x) \in G_{\tau,n} | h_{\tau,n}((a,x)) = 1 \wedge \mathrm{ord}(a) | m'\}.$$

According to Theorem 4.5(c), for each a, there is exactly one x such that $h_{\tau,n}((a,x)) = 1$. Thus

$$|T_{m'}| \le |\{a \in \mathbb{Z}_{2^\tau} | \mathrm{ord}(a) | m'\}| = \gcd(2^\tau, m').$$

By the choice of the message space, m' is between 1 and $2^\rho - 1$ (m' is the difference between two different legal messages; negative values need not be considered separately), and therefore $\gcd(2^\tau, m') < 2^\rho$. □

As for the efficiency, first note that a multiplication by 4 modulo n can be replaced by two shifts and at most two subtractions and is therefore negligible compared with a general multiplication modulo n. A group operation in $G_{\tau,n}$ is therefore essentially one modular multiplication because the exponent of 4 is 0 or 1. This yields the following:

- Signing: The term $m\,sk_2$ with $sk_2 \in G_{\tau,n}$ can be evaluated with any exponentiation algorithm. Since we just saw that a group operation in $G_{\tau,n}$ is essentially a modular multiplication, this corresponds to an exponentiation modulo n with the exponent m. In other words, it needs between ρ and $(3/2)\rho$ modular multiplications with a suitable exponentiation algorithm.
- Test: A signature $s = (a, x)$ on the message m is acceptable iff

$$pk_1\, pk_2^m = h_{\tau,n}((a,x)) \Leftrightarrow pk_1 = \pm 4^a x^{2^\tau} pk_2^{-m}.$$

Since the message is ρ bits long and $\tau = \rho + \sigma$, we see that a signature can be tested using at most $\tau + \rho = 2\rho + \sigma$ modular multiplications. With a good exponentiation algorithm, one can get close to $\rho + \sigma$ modular multiplications.

- The length of the secret key (for one message) is $2(k + \tau) = 2(k + \rho + \sigma)$ bits.
- The length of the public key is $2k$.
- The signature length (for a ρ-bit message) is $k + \tau = k + \rho + \sigma$ bits.

Thus the two schemes require almost the same amount of computation and storage for keys and signatures. The main difference is that the key generation in the first scheme is simpler. A variant of the Factoring Scheme that does not need a zero-knowledge proof, at the cost of an additional summand k in most of the lengths and numbers of multiplications, is given in [34]. Table 1 compares the two schemes for $k = \rho = |q|_2 \approx |p|_2$; remember that k is also the length of the message.

TABLE 1
Complexity of the two instantiations of the general construction.

	Discrete logarithm	Factoring
sign	2 multiplications	$\approx k$ multiplications
test	$< 2k$ multiplications	$< 2k + \sigma$ multiplications
Length of sk	$4k$	$4k + 2\sigma$
Length of pk	$2k$	$2k$
Length of a signature	$2k$	$2k + \sigma$

5. Signing many long messages. In section 4, constructions of fail-stop signature schemes for one short message were presented. We now extend these constructions so that an arbitrary number of messages of arbitrary length can be signed.

5.1. Overview. The easiest way to sign more than one message is to use a scheme from section 4 and to prepare as many keys as one intends to sign messages.

However, this is not very practical. In particular, the distribution of public keys assumes that each signer has access to a reliable broadcast channel (which may be realized by a certification hierarchy or a kind of phone book in practice). The use of such a channel should be minimized. In fact, some authors even require of ordinary digital signature schemes that after the initial key generation of fixed length, signing can go on "polynomially forever" [1, 30]. Hence one important result of the next two constructions is that they guarantee very short public keys.

The basic idea behind these constructions is tree authentication. The same ideas underlie all published provably secure ordinary digital signature schemes; however, some variations necessitate a new proof each time. Actually, there are two types of tree authentication, that of [29, 26, 27] and that of [28, 20], which we call bottom-up and top-down, respectively, corresponding to how the tree is constructed. The former leads to shorter signatures; the latter is more flexible. We sketch their fail-stop versions in sections 5.3 and 5.4, respectively.

In both of these constructions, the length of the secret key is linear in the number of messages to be signed. We prove in section 6 that this cannot be avoided if one defines "secret key" as in Definition 3.1, i.e., including all of the secret random bits that the signer will ever need. However, we show that this secret key never needs to exist completely at the same time: In section 5.4, we modify the top-down construction so that only a small amount of secret storage is needed.

In addition, we show in section 5.2 how long messages can be hashed before signing so that the length of signatures and keys is independent of the message length. In section 5.5, we present a special variant of the constructions based on discrete logarithms that shortens the secret key by approximately a factor of 2. This will be interesting in comparison with the lower bounds; see section 6. In section 5.6, we sketch more efficient constructions for the case where all the signatures by one signer have the same recipient.

5.2. Collision-resistant hash functions and message hashing. Message hashing means that a hash function is applied to each message before it is signed, which reduces messages of arbitrary length to a fixed length. At first sight, it could seem impossible to do this with fail-stop signatures. Given a correct signature s on a message m, an arbitrarily powerful forger can find another message m' that is mapped to the same string by the hash function. Hence this forger knows that s is also the correct signature on m'. Thus the signer cannot use the idea described in section 1.3 to repudiate the successful forgery (m', s). Fortunately, this is not a problem because such a forgery gives the signer a collision of the hash function, which she can present as a proof of forgery. Under the assumption that the hash function is collision-resistant, the signer cannot construct such a collision by herself.

Collision-resistant hash functions have been formally defined in [10]. Under Assumptions DL and F, efficient collision-resistant hash functions exist that need about one multiplication per message bit and where the length of the output is the length of the modulus used in the hash function [10, 8]. In practice, one may decide to use much faster hash functions whose security cannot be proved under well-known assumptions (or cannot even be defined, e.g., if they have no keys), such as RIPEMD-160 [13] (follow the references for more such functions and known attacks).

If a hash function is used in a fail-stop signature scheme, the key for the hash function, i.e., the particular instance of the hash function, must be chosen by the center (or the recipients according to section 3.3), i.e., in our constructions, it is part of the prekey. The parameters must be chosen so that the output of the hash function

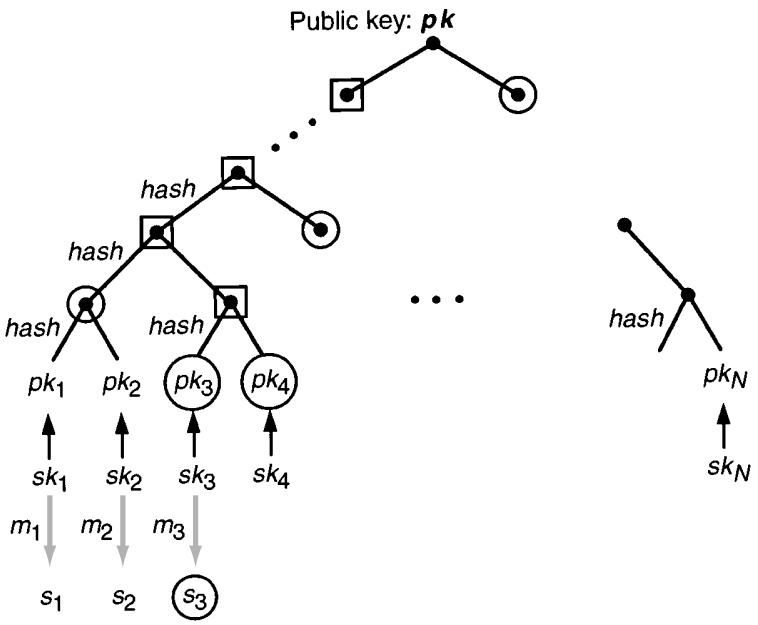


FIG. 2. *Fail-stop signature scheme with bottom-up tree authentication. Thin black arrows denote the relation between a one-time secret key and the corresponding one-time "public" key, broad grey arrows denote one-time signatures, and the tree is constructed by repeatedly hashing pairs of values. For instance, the complete signature on m_3 consists of the encircled nodes. To test it, the recipient reconstructs the nodes in squares.*

fits into the message length of the underlying fail-stop signature scheme. Then it is easy to prove that such a combination of a secure collision-resistant hash function and a secure fail-stop signature scheme is a secure fail-stop signature scheme for messages of arbitrary length.

5.3. Bottom-up tree authentication. The construction of bottom-up tree authentication is sketched in Figure 2. It is based on any collision-resistant hash function and any one-time fail-stop signature scheme with prekey (all constructions in section 4 are of this form). We distinguish signatures and keys in the underlying one-time FSS scheme and the scheme now to be constructed by calling them *one-time* and *complete*, respectively, and put one-time "public" keys in quotes because they are no longer public.

A prekey and an instance of the hash function are chosen once and for all (by the center or the recipients). The signer generates N one-time key pairs (sk_j, pk_j) based on this prekey. She hashes them pairwise in the form of a binary tree. Only the final hash value, i.e., the root of the tree, is published as the complete public key pk of the new scheme.

The complete signature on the jth message, m_j, starts with the one-time signature s_j on m_j using sk_j. Secondly, it contains the corresponding value pk_j. Moreover, to authenticate pk_j, the branch from pk_j to the root is needed; thus the complete signature also contains the other children of the nodes on this branch (see Figure 2). Hence its length is logarithmic in N.

The recipient tests the one-time signature s_j using pk_j, reconstructs the values on the path to the root, and tests if this path ends at the correct public key pk.

A proof of forgery in the new scheme is either a proof of forgery in the one-time scheme or a collision of the hash function.

A complete formal description and a proof can be found in [35]. Here we concentrate on top-down tree authentication instead.

5.4. Top-down tree authentication and a construction with a small amount of secret storage. In this section, we proceed in two steps:

1. We present a natural fail-stop version of top-down tree authentication. It can be based on any one-time FSS scheme with prekey where arbitrarily long messages can be signed and thus, in particular, on the schemes from section 4 together with message hashing. In this construction, the public key is short and only a small amount of secret storage is needed immediately after the keys are chosen. However, a long secret key accumulates as more and more messages are signed.

2. We add measures so that the amount of secret storage is also small all the time. These measures are constructed specifically for the general construction described in section 4.2, i.e., for the efficient schemes based on factoring and discrete logarithms.

As in section 5.3, we use *one-time* and *complete* to distinguish signatures and keys in the underlying scheme and in the scheme to be constructed.

Construction 5.1 (top-down tree authentication). Let a one-time FSS scheme with prekey for the message space $\{0,1\}^+$ be given (see Definition 4.2). We construct a scheme for signing N messages as follows (see Figure 3).

- Key generation:
  - A prekey *prek* for the one-time FSS scheme is chosen and the protocol (P, V) is carried out for it.
  - The signer generates one one-time key pair (sk, pk) based on *prek* and publishes pk (and N, if it is not globally fixed) as the complete public key of the new scheme.
- Signing: Signing takes place in a binary tree with N leaves. The nodes are denoted by bit strings j and labeled with one-time key pairs (sk_j, pk_j). The root is Node ε and labeled with (sk, pk). The children of Node j are Node $j0$ and Node $j1$. All one-time key pairs are randomly generated by the signer based on the same prekey *prek*. A one-time secret key at an inner node is used to sign the pair of one-time "public" keys of its two children; a one-time secret key at a leaf is used to sign a real message. A complete signature s in the new scheme is one branch of these one-time signatures. To sign the first real message $m_{0\ldots0}$, only the one-time keys on the leftmost branch and their immediate other children have to be generated. Figure 3 shows the situation after $m_{0\ldots0}$ has been signed. Generally, a complete signature on m_j consists of the one-time signature on m_j using sk_j and the sequence of pairs of one-time "public" keys with their one-time signatures on the path from m_j to the root (see Figure 3). Thus the tree is gradually constructed from left to right.
- Test: Given a complete signature s, i.e., a branch of a tree, the recipient tests all the one-time signatures in it and checks that the path ends at the correct public key pk.
- Proof of forgery: Given a successfully forged complete signature s^*, i.e., a branch that ends at pk, the signer finds a node j where it "links in" to the correct tree, i.e., the same pk_j is used in s^* and in her current tree, but the message m_j^* signed at this node in s^* has not been signed by her. (Depending on the position of Node j, m_j^* may be a real message or a pair of one-time "public" keys.) She computes a proof of forgery, $proof_j$, for this forged one-time signature s_j^* in the one-time scheme. The complete proof of forgery, *proof*, consists of $proof_j$, pk_j, m_j^*, and s_j^*.

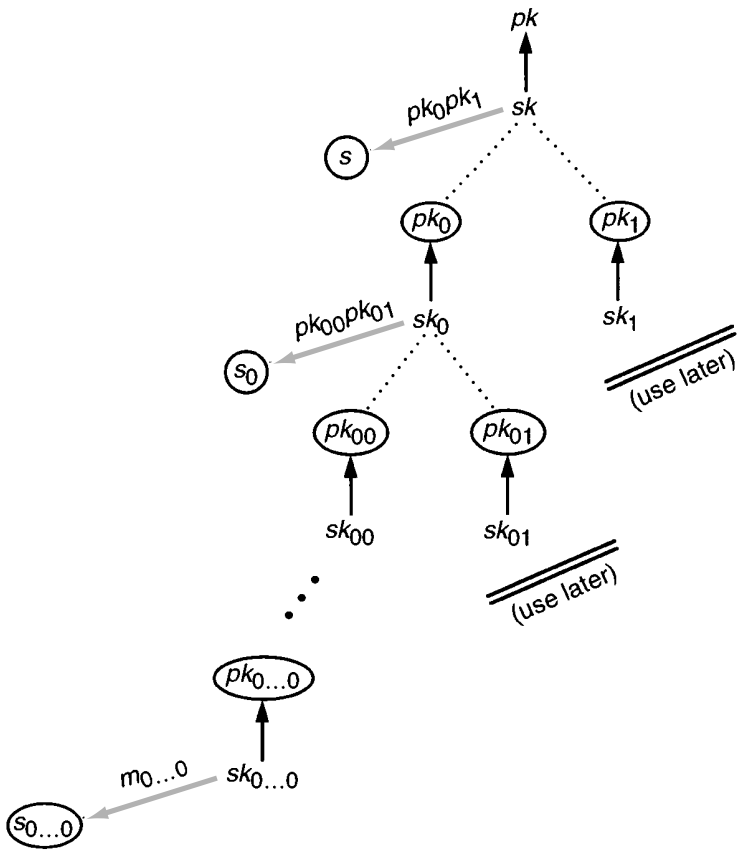


FIG. 3. *Fail-stop signature scheme with top-down tree authentication. Thin black arrows denote the relation between a one-time secret key and the corresponding one-time "public" key, broad grey arrows denote one-time signatures, and dotted lines only indicate a tree but are not related to a computation. For instance, the complete signature on* $m_{0\ldots0}$ *consists of the encircled nodes.*

• Verification of a proof of forgery: Given a value $proof = (proof_j, pk_j, m_j^*, s_j^*)$, verify that it is a valid proof of forgery in the one-time scheme, i.e., compute $verify(pk_j, m_j^*, s_j^*, proof_j)$. ◇

THEOREM 5.1. *If the underlying one-time FSS scheme is secure, Construction* 5.1 *describes a secure FSS scheme for* N *messages.*

Proof. The requirement from Definition 3.1 is obviously fulfilled. The security for the recipients, i.e., the fact that a polynomially bounded signer cannot construct signatures that she can later prove to be forgeries, follows immediately from that of the one-time scheme because any valid proof of forgery in the complete scheme is also a valid proof of forgery in the one-time scheme. (Remember that repeated use of the same prekey does not weaken the security; see the remarks after Definition 4.2.)

As to the security for the signer, first note that the probability that the signer accepts a bad prekey is the same in the one-time and the complete scheme. We show that all outcomes of the key generation based on a good prekey are in $Good_{\tilde{C}}$. For any successful forgery, the forger must select at least one node j where his complete signature links in to the correct tree, i.e., where he must forge a one-time signature. Given the forgery, the signer will find such a node (with probability 1) and can prove

the forgery at that node with the same probability as in the one-time scheme. □

A large amount of secret storage is still needed in Construction 5.1 because all the values sk_j must be stored secretly so that forgeries at any node can be proved. The basic idea used to reduce secret storage in Construction 5.2 is to store those sk_j's that are no longer used for signing in encrypted form (reliably, but not secretly) and to store only the encryption key secretly. However, information-theoretically secure encryption is needed, and a one-time pad (which is the only absolutely secure encryption scheme) is of no use because the key would be just as long as the encrypted message [42]. Hence special care must be taken that each individual sk_j remains secret, although information about the ensemble of sk_j's may become known.

We show how this can be done for the general Construction 4.1, taking into account that a forger has a priori information about the encrypted sk_j's because he may know the corresponding one-time "public" keys and signatures.

Construction 5.2 (scheme with small amount of secret storage). Let a one-time FSS scheme according to Construction 4.1 together with message hashing be given, and apply top-down tree authentication (Construction 5.1) to it with the following modifications:

- In key generation, the signer additionally chooses a value $e \in G$ randomly as an encryption key. She keeps e secret all the time.
- In signing: Whenever the signer has used up a one-time secret key $sk_j = (sk_{j,1}, sk_{j,2})$ by signing a message m_j (which may be a real message or a pair of one-time "public" keys), she proceeds as follows:
  - She encrypts $sk_{j,2}$ as $c_j := sk_{j,2} + e$.
  - She stores m_j, the one-time signature s_j, and the ciphertext c_j reliably but not necessarily secretly.
- For a proof of forgery: If the signer needs a one-time secret key sk_j to prove a forgery, she reconstructs it as follows: She decrypts $sk_{j,2} = c_j - e$ and recomputes $sk_{j,1} = s_j - m\, sk_{j,2}$, where $m := hash(m_j)$. ◇

THEOREM 5.2. *If the underlying one-time FSS scheme is secure, Construction* 5.2 *describes a secure FSS scheme for N messages. At any time, only the encryption key e and the one-time secret keys that have been generated but not yet used for signing* (*marked "use later" in Figure* 3)*, i.e., at most one per level of the tree, must be stored secretly.*

Proof. In addition to Theorem 5.1, we only have to show that the extra information stored nonsecretly, i.e., the ciphertexts c_j, does not help a forger to produce unprovable forgeries.

In every successfully forged complete signature, the signer still finds at least one successfully forged one-time signature s_j^* on a message m_j^*. If m_j^* and a message m_j that the signer has signed at this node are a collision of the hash function, this collision is already a proof of forgery. Otherwise, Theorem 4.1(c) guarantees that the forgery can be proved if s_j^* is different from the correct one-time signature that the signer would have produced for m_j^* at Node j. Thus it remains to be shown that the additional information does not help a forger to find exactly this signature. This signature depends only on sk_j, i.e., not on the values sk_l at other nodes. We consider the worst case, where the signer has already signed a message m_j at Node j. Let $m := hash(m_j)$. In the one-time scheme, the set of possible values sk_j from the point of view of a forger was $SK_{\bar{C},j} = \{(s_j - m\, sk'_{j,2}, sk'_{j,2}) | h(sk'_{j,2}) = pk_{j,2}\}$ with uniform distribution (see the proof of Theorem 4.2). Hence it suffices to show that all of these values are still possible if the forger has seen c_j and all the other ciphertexts c_l.

Let such a value $sk'_{j,2}$ be given. It corresponds to exactly one encryption key $e' := c_j - sk'_{j,2}$. This implies that the other one-time secret keys must be given by

$$sk'_{l,2} := c_l - e' = sk'_{j,2} + c_l - c_j.$$

The only question is whether these are possible secret keys, i.e., whether $h(sk'_{l,2}) = pk_{l,2}$. On one hand,

$$h(sk'_{l,2}) = h(sk'_{j,2}) \cdot h(c_l)/h(c_j) = pk_{j,2} \cdot h(c_l)/h(c_j),$$

and on the other hand,

$$h(c_l) = h(sk_{l,2}) \cdot h(e) = pk_{l,2} \cdot h(e) \quad \text{and} \quad h(c_j) = pk_{j,2} \cdot h(e).$$

Hence $h(sk'_{l,2}) = pk_{l,2}$. □

If this construction is applied to a usual complete tree, the amount of information that must be stored secretly is logarithmic in N (see Table 2 in section 6.5 for an example).

If we use a list-shaped tree, i.e., the left child of each node is a real message, at any time only two sk_j's have been generated but not yet used for signing. Thus the amount of secret storage is constant (as a function of N). However, later signatures are very long. Thus list-shaped trees should only be used with a single recipient; see section 5.6.

One can also use trees of other shapes, e.g., ternary trees or trees where one decides dynamically whether the message m_j signed at a node is a real message or a pair of one-time "public" keys and specifies this by one additional bit in m_j.

5.5. Improvement for the discrete logarithm constructions. In this section, we present a fail-stop signature scheme for signing N messages where the total length of the secret key is $(2N + 2)k$. The smallest upper bound in the previous sections was $4Nk$, achieved by pure repetition of the one-time Discrete Logarithm Scheme and also in bottom-up tree authentication based on it.

The scheme is a variant of pure repetition of the Discrete Logarithm Scheme where half of each one-time secret key is reused in the next signature.

Construction 5.3.

- Key generation:
  - As in Construction 4.3, a prekey (p, q, a, b) is chosen and verified, where p and q are primes with $q|(p-1)$ and a and b are generators of the group H_q, the unique subgroup of $\mathbb{Z}_p^*$ of order q.
  - The signer chooses a secret key

$$sk := ((x_1, y_1), (x_2, y_2), \ldots, (x_{N+1}, y_{N+1}))$$

consisting of numbers between 0 and $q - 1$. The corresponding public key is

$$pk := (pk_1, \ldots, pk_{N+1}) := (a^{x_1} b^{y_1}, \ldots, a^{x_{N+1}} b^{y_{N+1}}).$$

- The message space is restricted to $\mathbb{Z}_q^*$.
- Signing: The correct signature on the jth message, m_j, is the triple

$$s := (j, x_j + m_j x_{j+1}, y_j + m_j y_{j+1}).$$

- Test: A triple (j, x, y) is an acceptable signature on the jth message, m_j, iff
$$a^x b^y = pk_j pk_{j+1}^{m_j}.$$
- Proofs of forgery are constructed and verified as in Construction 4.3. ◇

THEOREM 5.3. *Under Assumption* DL*, Construction* 5.3 *describes a secure FSS scheme.*

Proof. It is clear that correct signatures pass the test because individual signatures are identical to those in Construction 4.3. Similarly, the security for the recipients is fulfilled because proofs of forgery are identical to those in Construction 4.3. Furthermore, every successful forgery f that is not the correct signature, i.e., $f = (m_j^*, (j, x, y))$ with $(j, x, y) \neq sign(sk, j, m_j^*)$, can be proved as before. It remains to show that the reuse of halves of the one-time secret keys does not make it too easy for a forger to guess correct signatures.

It suffices to show this for the worst case, where the signer has already issued N signatures on messages $m_1, \ldots, m_N$, possibly chosen by the forger, and the forger therefore has maximum information. We first determine the size of the set $SK_{\tilde{C}}$ of possible secret keys given these signatures and the public key. $SK_{\tilde{C}}$ is the set of solutions to the equations defining the public key and the signatures. With $e := \log_a(b)$, all of these equations can be written as linear equations over GF(q). They are described by the following $(3N+1) \times (2N+2)$ matrix A. The columns correspond to the elements of sk in the order as above, the first $2N$ rows to the signatures, and the last $N+1$ rows to the components of the public key.

$$A = \begin{pmatrix}
1 & 0 & m_1 & & & & & & & \\
 & 1 & 0 & m_1 & & & & & & \\
 & & 1 & 0 & m_2 & & & & & \\
 & & & 1 & 0 & m_2 & & & & \\
 & & & & & \ddots & & & & \\
 & & \mathbf{0} & & & & & & & \\
 & & & & & & 1 & 0 & m_N & \\
 & & & & & & & 1 & 0 & m_N \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
1 & e & & & & & & & & \\
 & & 1 & e & & & & & & \\
 & & & & & \ddots & & & & \\
 & & \mathbf{0} & & & & & & & \\
 & & & & & & & & 1 & e
\end{pmatrix}.$$

We now show that the rank of A is $2N+1$. For this, consider the following submatrix of A:

$$\begin{pmatrix}
1 & 0 & m_j & 0 \\
0 & 1 & 0 & m_j \\
1 & e & 0 & 0 \\
0 & 0 & 1 & e
\end{pmatrix}.$$

One can easily see that this submatrix has rank 3 and that the third row can be removed. By repeating this step for $j := 1, \ldots, N$, we delete all rows $(0 \ldots 0\, 1\, e\, 0 \ldots 0)$ except for the last one. The resulting matrix clearly has rank $2N + 1$.

Hence $|SK_{\bar{C}}| = q$. We now show that each of these q possible secret keys yields a different correct signature on the new message m_j^*. For this it suffices to show that any additional signature determines the secret key uniquely. The additional signature introduces two new equations corresponding to the rows

$$\begin{array}{ccccccc} 0 \ldots 0 & 1 & 0 & m_j^* & 0 & 0 \ldots 0 \\ 0 \ldots 0 & 0 & 1 & 0 & m_j^* & 0 \ldots 0 \end{array}.$$

One can easily see that the rank of the new matrix is $2N+2$. (Remember that $m_j \neq 0$ for all j.) Hence an additional, forged signature is correct only for one out of the q possible secret keys. This completes the proof. □

Construction 5.3 can be combined with bottom-up tree authentication; this yields public keys as short as in section 5.3 and secret keys as short as in Construction 5.3.

5.6. The case of a single recipient. In some important applications, all signatures of a signer have the same recipient, while any third party is able to settle a dispute (otherwise, one would not need a signature scheme). In this case, significant efficiency improvements are possible because the recipient can store information from previously received signatures. Now the most efficient construction is top-down tree authentication with a list-shaped tree so that only one new node of the tree must be produced, sent, and tested each time. Thus signing and testing are reduced to the following procedures:

- Signing: Assume that the current one-time secret key is sk_j. The signer generates a new one-time key pair (sk_{j+1}, pk_{j+1}); this can be done off-line before the new message m_j is known. The complete signature on m_j consists of the new one-time "public" key pk_{j+1} and a one-time signature with sk_j on the pair (m_j, pk_{j+1}).
- Test: The recipient tests the new one-time signature with his current one-time "public" key pk_j and then sets pk_{j+1} as his current one-time "public" key.

The old one-time secret and "public" keys can be stored as in Construction 5.2. If a third party has to settle a dispute, it must test the signature with respect to the original public key pk. This can be done with overhead logarithmic in the length of the list: With logarithmic search, one first finds an index $j' \leq j$ where the signer and the recipient agree on $pk_{j'}$ but not on the pair $(m_{j'}, pk_{j'+1})$ signed with respect to it. Then it suffices to solve the dispute regarding this one-time signature.

6. Lower bounds on the efficiency of fail-stop signatures. In this section, we try to answer the following two questions:

1. How close to the optimum are the existing constructions?

2. Is there a certain price in efficiency to pay for information-theoretic security instead of computational security?

We do this by giving lower bounds on the length (more precisely, the entropy) of keys and signatures. In practice, these parameters correspond to the communication and storage complexity of both key generation and the transmission of signed messages. Similar lower bounds have been investigated for other cryptographic schemes with information-theoretic security requirements, e.g., encryption schemes [42], au-

thentication codes (starting with [16]; see, e.g., [40]), and unconditionally secure signature schemes [23].[3]

Section 6.1 sketches the information-theoretic background, section 6.2 introduces the random variables, section 6.3 investigates the length of the secret key, section 6.4 investigates the length of signatures and public keys, and section 6.5 answers the questions posed above, i.e., it compares the lower bounds to the known upper bounds and to ordinary digital signature schemes.

6.1. Information-theoretic background. Since a random variable with entropy σ cannot be coded with less than σ bits on average, it is sufficient to prove lower bounds on the entropy of random variables in order to have lower bounds on the length of the values of these random variables.

For the formal theorems and proofs, we assume that the reader is familiar with elementary information theory (see [41] and [15, sections 2.2 and 2.3]). We briefly repeat the most important notions in the notation of [15]. Assume that a common probability space is given. Capital letters denote random variables and small letters their corresponding values, and $\mathrm{P}(X = x)$ is abbreviated as $\mathrm{P}(x)$, etc. The joint random variable of X and Y is written as X, Y. The entropy of a random variable X is

$$\mathrm{H}(X) := -\sum_x \mathrm{P}(x) \log \mathrm{P}(x).$$

$\mathrm{H}(X \mid Y)$ denotes the conditional entropy of X if Y is known, and $\mathrm{I}(X;Y)$ denotes the mutual information between X and Y. They are defined as follows:

$$\mathrm{H}(X \mid Y) : = -\sum_{x,y} \mathrm{P}(x,y) \log \mathrm{P}(x \mid y),$$

$$\mathrm{I}(X;Y) := \mathrm{H}(X) - \mathrm{H}(X \mid Y).$$

Furthermore, the conditional mutual information is defined as

$$\mathrm{I}(X;Y \mid Z) := \mathrm{H}(X \mid Z) - \mathrm{H}(X \mid Y,Z).$$

The following important rules will often be used:

$$\mathrm{H}(X) \geq 0, \qquad \mathrm{H}(X \mid Y) \geq 0, \tag{6.1}$$

$$\mathrm{H}(Y,Z \mid X) = \mathrm{H}(Y \mid X) + \mathrm{H}(Z \mid Y,X), \tag{6.2}$$

$$\mathrm{I}(X;Y) = \mathrm{I}(Y;X). \tag{6.3}$$

Additionally, we often need Jensen's inequality for the special case of the logarithm [14]: If $p_i \geq 0$ and $x_i > 0$ for all i and the sum of the p_i's is 1, then

$$\log\left(\sum_i p_i x_i\right) \geq \sum_i p_i \log(x_i).$$

[3] However, the lower bounds in the following cannot be derived entirely from information-theoretic security requirements; this makes some new proof techniques necessary.

6.2. The random variables. The common probability space for all of the random variables arising in a fail-stop signature scheme is given by the random bits used in key generation; here we made use of the fact that we could define all of the other algorithms as deterministic by regarding all random bits as part of the secret key.

For the lower bounds, we only need the case where all parties execute G honestly, and we always consider a fixed triple of parameters $par = (k, \sigma, N)$. Then the probabilities of sk and pk are uniquely determined. Hence we can define the following random variables:

- SK and PK are the random variables of the secret and public key, respectively.
- If a message sequence $\underline{m} = (m_1, \ldots, m_j)$ with $j \leq N$ has been fixed, random variables S_i for $i = 1, \ldots, j$ are defined as the correct signature on m_i in this context, i.e.,

$$S_i := sign(SK, i, (m_1, \ldots, m_i)).$$

Since the signature is a deterministic function of the secret key and the message sequence, we have

$$\mathrm{H}(S_i \mid SK) = 0.$$

- Similarly, for the given message sequence $\underline{m} = (m_1, \ldots, m_j)$, the random variable of the history up to the ith signature (for $i \leq j$) is defined as

$$Hist_i := ((m_1, \ldots m_i), (S_1, \ldots, S_i)).$$

6.3. Lower bound on the number of secret random bits needed. As mentioned in section 5, the largest difference between fail-stop signatures and ordinary digital signatures is the length of the secret key. We have shown in Theorem 5.2 that the signer need not store a lot of secret information at the same time; however, the overall number of secret random bits chosen was still linear in the number of messages to be signed. We now show that this cannot be avoided.

If all fail-stop signature schemes followed the construction idea in section 1.3, this would be quite easy to see:

1. Even an arbitrarily powerful forger must not be able to guess the signer's correct signature.
2. Hence the entropy of the correct signature is large.
3. Since this holds for each additional signature, even if some signatures are already known, the entropies of the signatures can be added, and therefore the overall entropy of the signer's secrets is large.

Our proof follows this outline; however, only a weaker average version of statement 1 can be derived from the definitions. If we required that applying *prove* to the signer's correct signature never yielded a valid proof of forgery, a formal proof would be easy and would yield

$$\mathrm{H}(SK \mid PK) \geq (N+1)\sigma.$$

However, e.g., in fail-stop signature schemes with message hashing, an arbitrarily powerful forger sometimes *does* know correct signatures on new messages (those with the same hash value as the original message), and the signer can prove them to be forgeries.

Furthermore, note that the desired lower bound cannot possibly be proved from the signer's security alone. As a counterexample, suppose that the signer were allowed

to disavow all signatures in an ordinary digital signature scheme; then she would be unconditionally secure without using many random bits (but the recipient would not be secure at all). This is a problem because Definition 3.2, like all computational cryptographic definitions, is asymptotic, i.e., security is only guaranteed for "k sufficiently large." Thus in a certain sense, we can only derive lower bounds for $k \geq k_0$ for an unknown k_0. This may seem unsatisfactory: No one would have doubted that we need arbitrarily long keys if we made k sufficiently large.

However, the real purpose of lower bounds is to say "whenever we have certain requirements on the security, we have to pay the following price in terms of efficiency." In this section, this means, more precisely, "if the signer wants the probability of unprovable forgery to be at most $2^{-\sigma}$, and the recipients want some security, too, then at least the following number of random bits is needed (as a function of σ and the security for the recipients)."

To quantify the security for the recipients, it suffices for our purpose to consider the probability that the signer can prove that her own signatures are "forgeries" simply by applying the algorithm *prove*. In practice, one will require this probability to be at most, say, 2^{-100}, or, more generally, $2^{-\sigma^*}$ for some σ^*. We will prove the lower bounds as a function of this parameter σ^* (in addition to σ). Now we can proceed formally.

DEFINITION 6.1. *Let a fail-stop signature scheme and parameters σ and N be given. For every message sequence $\underline{m} = (m_1, \ldots, m_{N+1})$, we define a polynomial-time algorithm $\tilde{A}_{\underline{m}}$ that describes how a dishonest signer could try to disavow her own correct signatures. (This algorithm should be rather useless!)*

1. *Execute G correctly (with the center) to obtain sk and pk.*

2. *Compute the signatures $s_1, \ldots, s_N$ on $m_1, \ldots, m_N$ and the histories $hist_1, \ldots, hist_N$ correctly. Then because m_{N+1} is one message too much, compute its signature s_{N+1} as if m_N had not been signed, i.e., $s_{N+1} := sign(sk, N, (m_1, \ldots, m_{N-1}, m_{N+1}))$.*

3. *If $provable(sk, pk, hist_{i-1}, (m_i, s_i))$ for an $i \in \{1, \ldots, N+1\}$, then output the proof $prove(sk, m_i, s_i, hist_{i-1})$.*

We say that k is large enough to provide security level σ^* for the recipients against $\tilde{A}_{\underline{m}}$ *if the success probability of $\tilde{A}_{\underline{m}}$ is at most $2^{-\sigma^*}$.*[4] *This probability is over the random bits of $\tilde{A}_{\underline{m}}$ and the random bits of the center in step 1.*

THEOREM 6.1. *Let a fail-stop signature scheme with actual parameters σ and N and a security level σ^* be given. Let $\sigma' := \min(\sigma, \sigma^*)$. Then for all k large enough to provide security level σ^* for the recipients against the algorithm $\tilde{A}_{\underline{m}}$ for at least one sequence $\underline{m}$ of $N+1$ pairwise-distinct messages,*[5]

$$\mathrm{H}(SK \mid PK) \geq (N+1)(\sigma' - 1).$$

[4]The formal definition of the recipient's security immediately implies the existence of k_0 such that $\tilde{A}_{\underline{m}}$ has this property for all $k \geq k_0$. We have now bypassed the problem that we do not know how large k_0 is because we simply know that it must be large enough in a practical application.

Note that it would be impossible to require k to be large enough to provide the same level of security against all polynomial-time algorithms instead of only $\tilde{A}_{\underline{m}}$ because for a fixed k, we have a finite problem, and all finite problems can be solved perfectly by some constant-time algorithm. However, our $\tilde{A}_{\underline{m}}$ is not only asymptotically polynomial-time but definitely feasible at the given k because it is of the same complexity as *sign* and *prove*. Requirements of the same type are well known in practice, e.g., when one requires k to be large enough to make the success probability of all known factoring algorithms small.

[5]Note that this is a very weak condition on k. The contrary is that the signer can sign an arbitrary message sequence $\underline{m}$ and disavow it with significant probability using $\tilde{A}_{\underline{m}}$.

Since $\underline{m}$ is fixed throughout the proof, we can omit it in the notation, i.e., all random variables S_i and $Hist_i$ are implicitly assumed to belong to this message sequence. We also omit $\underline{m}$ in the histories, i.e., we abbreviate $Hist_i := (S_1, \ldots, S_i)$ for $i = 1, \ldots, N+1$. The first lemma formalizes statement 1 above, i.e., that it is hard to guess correct signatures.

LEMMA 6.1. *With the notation of Theorem* 6.1*, for each* $i \leq N+1$ *and each pair* $(pk, hist_{i-1})$*, fix an optimal guess* $s^*(pk, hist_{i-1})$ *at the correct signature on* m_i *if* pk *and* $hist_{i-1}$ *are known, i.e., for all values* s*,*

$$\mathrm{P}(S_i = s^*(pk, hist_{i-1}) \mid pk, hist_{i-1}) \geq \mathrm{P}(S_i = s \mid pk, hist_{i-1}).$$

Then this guess is still not very good on average:

$$\mathrm{P}(S_i = s^*(PK, Hist_{i-1})) \leq 2^{-\sigma'+1}.$$

Proof. Let i and $(pk, hist_{i-1})$ be given. We have

$$\begin{aligned}
&\mathrm{P}(S_i = s^*(PK, Hist_{i-1})) \\
&\quad = \mathrm{P}(S_i = s^*(PK, Hist_{i-1}) \wedge provable(SK, PK, Hist_{i-1}, (m_i, S_i))) \\
&\qquad + \mathrm{P}(S_i = s^*(PK, Hist_{i-1}) \\
&\qquad\quad \wedge \neg provable(SK, PK, Hist_{i-1}, (m_i, s^*(PK, Hist_{i-1})))) \\
&\quad \leq \mathrm{P}(provable(SK, PK, Hist_{i-1}, (m_i, S_i))) \\
&\qquad + \mathrm{P}(\neg provable(SK, PK, Hist_{i-1}, (m_i, s^*(PK, Hist_{i-1})))).
\end{aligned}$$

The first term is bounded by the success probability of $\tilde{A}_{\underline{m}}$ and therefore by $2^{-\sigma^*}$. The second term can be bounded using the security for the signer (Definition 3.4):

$$\begin{aligned}
&\mathrm{P}(\neg provable(SK, PK, Hist_{i-1}, (m_i, s^*(PK, Hist_{i-1})))) \\
&= \sum_{pk, hist_{i-1}} \mathrm{P}(pk, hist_{i-1}) \mathrm{P}(\neg provable(SK, PK, Hist_{i-1}, (m_i, s^*(PK, Hist_{i-1}))) \\
&\qquad\qquad \mid PK = pk, Hist_{i-1} = hist_{i-1}) \\
&\leq \sum_{pk, hist_{i-1}} \mathrm{P}(pk, hist_{i-1}) 2^{-\sigma} \\
&= 2^{-\sigma}.
\end{aligned}$$

Together, these two bounds yield the lemma. □

Lemma 6.2 uses Lemma 6.1 to give a lower bound corresponding to statement 2 of the informal proof sketch.

LEMMA 6.2. *With the notation of Theorem* 6.1*, for each* $i \leq N+1$*,*

$$\mathrm{H}(S_i \mid PK, Hist_{i-1}) \geq \sigma' - 1.$$

Proof. We can write the conditional entropy as

$$\mathrm{H}(S_i \mid PK, Hist_{i-1}) = -\sum_{s_i,pk,hist_{i-1}} \mathrm{P}(s_i, pk, hist_{i-1}) \log(\mathrm{P}(s_i \mid pk, hist_{i-1}))$$

$$\geq -\log\left(\sum_{s_i,pk,hist_{i-1}} \mathrm{P}(s_i, pk, hist_{i-1})\mathrm{P}(s_i \mid pk, hist_{i-1})\right)$$

using Jensen's inequality. (Note that one *cannot* prove that the individual values $\log(\mathrm{P}(s_i \mid pk, hist_{i-1}))$ are at least σ.) Now it suffices to show that the sum in the logarithm is at most $2^{-\sigma'+1}$. In fact, with Lemma 6.1,

$$\sum_{s_i,pk,hist_{i-1}} \mathrm{P}(s_i, pk, hist_{i-1})\mathrm{P}(s_i \mid pk, hist_{i-1})$$

$$\leq \sum_{s_i,pk,hist_{i-1}} \mathrm{P}(s_i, pk, hist_{i-1})\mathrm{P}(S_i = s^*(pk, hist_{i-1}) \mid pk, hist_{i-1})$$

$$= \sum_{pk,hist_{i-1}} \left(\mathrm{P}(S_i = s^*(pk, hist_{i-1}) \mid pk, hist_{i-1}) \sum_{s_i} \mathrm{P}(s_i, pk, hist_{i-1})\right)$$

$$= \mathrm{P}(S_i = s^*(PK, Hist_{i-1}))$$

$$\leq 2^{-\sigma'+1}. \qquad \square$$

Proof of Theorem 6.1. Theorem 6.1 can be proved from Lemma 6.2 using rule (2) from section 6.1. Recall the abbreviation $Hist_i = (S_1, \ldots, S_i)$ for $i \leq N+1$. First, we show by induction over i that the entropy of all of these signatures together is large, i.e., for all $i \leq N+1$,

$$\mathrm{H}(Hist_i \mid PK) \geq i(\sigma' - 1). \tag{4}$$

For $i = 1$, (4) is Lemma 6.2. If (4) has already been proved for $i-1$, it holds for i because

$$\begin{aligned}\mathrm{H}(Hist_i \mid PK) &= \mathrm{H}(Hist_{i-1} \mid PK) + \mathrm{H}(S_i \mid PK, Hist_{i-1}) \\ &\geq (i-1)(\sigma'-1) + (\sigma'-1) \\ &= i(\sigma'-1).\end{aligned}$$

We now use the fact that signing is deterministic, i.e., SK uniquely determines $Hist_{N+1}$. This implies $\mathrm{H}(Hist_{N+1} \mid PK, SK) = 0$, and therefore

$$\begin{aligned}\mathrm{H}(SK \mid PK) &= \mathrm{H}(SK, Hist_{N+1} \mid PK) - \mathrm{H}(Hist_{N+1} \mid PK, SK) \\ &= \mathrm{H}(SK, Hist_{N+1} \mid PK) \\ &\geq \mathrm{H}(Hist_{N+1} \mid PK) \\ &\geq (N+1)(\sigma'-1). \qquad \square\end{aligned}$$

6.4. Length of signatures and public keys. Signatures and public keys are not much longer in current fail-stop signature schemes than in ordinary digital signature schemes. Hence the lower bounds are also very small. The basic idea why fail-stop signatures might be longer at all is as follows:

1. First, there must be at least 2^σ acceptable signatures; otherwise, the correct signature could be guessed too easily.

2. Second, it must be hard for a forger to guess acceptable signatures at all. Thus the density of the set of acceptable signatures within the signature space should be small, e.g., at most $2^{-\sigma^*}$ for some σ^*.

Hence we expect the size of the signature space to be at least $2^{\sigma+\sigma^*}$. We will prove a slightly lower bound but more generally on the entropy of each signature.

We deal with the asymptotic character of the security against forgery in the same way as in section 6.3, i.e., we fix one simple algorithm that a forger might use to guess acceptable signatures.

DEFINITION 6.2. *Let a fail-stop signature scheme and parameters σ and N be given. For every message sequence $\underline{m} = (m_1, \ldots, m_i)$ with $i \leq N$, we define a (very efficient and stupid) polynomial-time forging algorithm $\tilde{F}_{\underline{m}}$. On input pk, do the following:*

1. *Choose a new key pair (sk^*, pk^*) by carrying out both roles of G correctly.*

2. *Compute a signature $s_i := sign(sk^*, i, \underline{m})$ with the new key and output (m_i, s_i) as a proposed forgery for the given key pk.*

We say that k is large enough to provide security level σ^* against forgery by $\tilde{F}_{\underline{m}}$ *if the probability that $\tilde{F}_{\underline{m}}$ outputs an acceptable signature on m_i is at most $2^{-\sigma^*}$.*[6]

THEOREM 6.2. *Let a fail-stop signature scheme with actual parameters σ and N and a security level σ^* be given. Let $\sigma' := \min(\sigma, \sigma^*)$. Then for all k large enough to provide security level σ^* both against forgery by $\tilde{F}_{\underline{m}}$ and against $\tilde{A}_{\underline{m}}$ for at least one sequence $\underline{m}$ of $i \leq N$ pairwise-distinct messages,*

$$\mathrm{H}(S_i) \geq \sigma' + \sigma^* - 1$$

and

$$\mathrm{H}(PK) \geq \sigma^*.$$

The following lemma formalizes step 2 of the informal proof sketch. The fact that the number of acceptable signatures, given the public key, is much smaller than the complete signature space is generalized as follows: The public key contains a lot of information about the correct signature.

LEMMA 6.3. *With the notation of Theorem* 6.2*,*

$$\mathrm{I}(S_i; PK) \geq \sigma^*.$$

[6] The security against forgery, Theorem 3.1, implies the existence of k_0 such that all $k \geq k_0$ have this property.

Proof. By the definitions and Jensen's inequality,

$$\begin{aligned}
\mathrm{I}(S_i; PK) &= - \sum_{s_i, pk : \mathrm{P}(s_i, pk) \neq 0} \mathrm{P}(s_i, pk) \log(\mathrm{P}(s_i)\mathrm{P}(pk)/\mathrm{P}(s_i, pk)) \\
&\geq - \log \left(\sum_{s_i, pk : \mathrm{P}(s_i, pk) \neq 0} \mathrm{P}(pk, s_i)\mathrm{P}(s_i)\mathrm{P}(pk)/\mathrm{P}(s_i, pk) \right) \\
&= - \log \left(\sum_{s_i, pk : \mathrm{P}(s_i, pk) \neq 0} \mathrm{P}(s_i)\mathrm{P}(pk) \right).
\end{aligned}$$

It suffices to show that the argument of the logarithm is bounded above by $2^{-\sigma^*}$. First, the fact that all correct signatures pass the test, i.e.,

$$\mathrm{P}(s_i, pk) \neq 0 \Rightarrow test(pk, m_i, s_i) = ok,$$

yields

$$\begin{aligned}
\sum_{s_i, pk : \mathrm{P}(s_i, pk) \neq 0} \mathrm{P}(s_i)\mathrm{P}(pk) &= \sum_{s_i, pk} \mathrm{P}(s_i)\mathrm{P}(pk) 1_{\mathrm{P}(s_i, pk) \neq 0} \\
&\leq \sum_{s_i, pk} \mathrm{P}(pk)\mathrm{P}(s_i) 1_{test(pk, m_i, s_i) = ok}.
\end{aligned}$$

Now we use the security against $\tilde{F}_{\underline{m}}$, which can be written formally as

$$\begin{aligned}
\mathrm{P}(test(pk, m_i, s_i) = ok :: (acc, pk, sk, \varepsilon) \leftarrow G; (acc^*, pk^*, sk^*, \varepsilon) \leftarrow G; \\
s_i := sign(sk^*, i, \underline{m})) \leq 2^{-\sigma^*}.
\end{aligned}$$

This means that if a public key pk is chosen according to the distribution of PK and, independently of this, a signature is chosen according to the distribution of S_i, this signature is acceptable with respect to pk with probability at most $2^{-\sigma^*}$. Hence

$$\sum_{s_i, pk} \mathrm{P}(pk)\mathrm{P}(s_i) 1_{test(pk, m_i, s_i) = ok} \leq 2^{-\sigma^*}.$$

This is exactly what remained to be shown. □

Proof of Theorem 6.2. Lemma 6.3 implies $\mathrm{H}(S_i) - \mathrm{H}(S_i \mid PK) \geq \sigma^*$, and a special case of Lemma 6.2 implies $\mathrm{H}(S_i \mid PK) \geq \sigma' - 1$. (Here we exploit the security against $\tilde{A}_{\underline{m}}$.) As a consequence, $\mathrm{H}(S_i) \geq \mathrm{H}(S_i \mid PK) + \sigma^* \geq \sigma' + \sigma^* - 1$. Furthermore, $\mathrm{H}(PK) \geq \mathrm{I}(S_i; PK)$. □

For the case with a prekey (see section 4.2), we obtain slightly stronger results by using a forging algorithm $\tilde{F}^*_{\underline{m}}$ that generates its new key pair (sk^*, pk^*) based on the same prekey *prek*. In this case, the same proof yields

$$\mathrm{H}(S_i \mid Prek) \geq \sigma' + \sigma^* - 1 \quad \text{and} \quad \mathrm{H}(PK \mid Prek) \geq \sigma^*.$$

If, as usual in such schemes, pk is a function of sk and *prek*, we obtain one more result about the secret key.

TABLE 2
Comparison of constructions and lower bounds for fail-stop signatures.

Length of	Construction 5.1	Construction 5.2	Construction 5.3	Bottom-up	Single recipient	Lower bound
sk	$8Nk$	$8Nk$	$(2N+2)k$	$4Nk$	$4Nk$	$(N+2)(\sigma'-1)$
secret storage	$8Nk$	$4k\log(N)$	$(2N+2)k$	$4Nk$	$10k$	?
pk	$2k$	$2k$	$(N+1)k$	$2\sigma^*$	$2k$	σ^*
signature	$6k\log(N)$	$6k\log(N)$	$2k+\log(N)$	$6k+2\sigma^*\log(N)$	$4k$	$\sigma'+\sigma^*-1$

THEOREM 6.3. *In a fail-stop signature scheme with prekey and where the public key is a function of the secret key and the prekey, with the notation of Theorems* 6.1 *and* 6.2*,*

$$\mathrm{H}(SK \mid Prek) \geq (N+2)(\sigma'-1).$$

Proof. By rule (2), we get

$$\begin{aligned}\mathrm{H}(SK \mid Prek) &= \mathrm{H}(SK, PK \mid Prek) - \mathrm{H}(PK \mid SK, Prek)\\ &= \mathrm{H}(PK \mid Prek) + \mathrm{H}(SK \mid PK, Prek) - 0\\ &= \mathrm{H}(PK \mid Prek) + \mathrm{H}(SK \mid PK).\end{aligned}$$

In the last line, we used the fact that the prekey is part of the public key. Hence

$$\mathrm{H}(SK \mid Prek) \geq \sigma^* + (N+1)(\sigma'-1) \geq (N+2)(\sigma'-1). \qquad \square$$

6.5. Comparison. This section answers the two questions raised at the beginning of this section by evaluating how well our constructions meet the lower bounds and by comparing the lower bounds with similar bounds on ordinary digital signature schemes.

Table 2 approximately shows the length of the secret key (including all of the random bits needed later), the maximal amount of secret storage needed, the length of the public key and the length of a signature for Constructions 5.1, 5.2, and 5.3, bottom-up tree authentication, and the case of a single recipient. The table also shows the corresponding lower bounds; remember that all schemes use a prekey. All constructions are assumed to be based on the Discrete Logarithm Scheme from Construction 4.3 with parameters as in Table 1. Remember that $2k$ bits are needed to represent an element of the group G, k bits are needed for an element of H, and the total number of one-time secret keys in the tree in Figure 3 is $2N-1$. The bottom-up construction is shown for the case where the recipients trust a fast hash function with output length $2\sigma^*$.

It should be mentioned that [22] also presents a scheme for signing N messages where the signature length is only $2k$, and the key lengths and secret storage are the same as in Construction 5.3. However, the complexity of signing then grows linearly in N, and $\log(N)$ will be dominated by k anyway.

The difference between k in the upper bounds and σ and σ^*, and thus $\sigma' = \min(\sigma, \sigma^*)$, in the lower bounds is as follows: Usually, σ and σ^* would be chosen of equal size, say 100, whereas k must be at least 500 to give sufficient computational

security. If one believes that variants of the discrete logarithm problem exist that cannot be solved in subexponential time, e.g., on nonsupersingular elliptic curves (cf. [24]), one can choose k very close to σ^*, and then the lower bounds are met by the upper bounds except for small factors.

By disregarding the difference between k, σ, and σ^*, the current upper bound on the length of the secret key is $(2N+2)\sigma$ in contrast to the lower bound $(N+2)(\sigma-1)$. For the signature length, both bounds are 2σ, and for the public key, the upper bound is 2σ and the lower bound σ. However, having both public keys and signatures of length independent of N was only achieved if the signatures of each signer have a single recipient (or very few recipients). If every new signature may have a new recipient, there is a tradeoff: If the public key is of constant length, the signatures grow logarithmically in N because of the tree authentication. It is an interesting open question whether this can be avoided.

If we compare fail-stop signatures with ordinary digital signatures, the most obvious difference in terms of complexity is that the number of secret random bits needed for fail-stop signatures is linear in the number of messages to be signed. The lower bound on the length of fail-stop signatures is $\sigma'+\sigma^*-1$; for ordinary digital signatures, σ^* is a lower bound. Hence the two types of schemes differ by a factor of about 2. For public keys, the same lower bound σ^* holds for ordinary digital signature schemes, and in fact, even in the constructions, public keys of fail-stop signature schemes are not longer than in ordinary digital signature schemes.

7. Conclusion and outlook. We have defined fail-stop signature schemes formally and shown that their security is in fact stronger than that of ordinary digital signature schemes. In particular, one can use fail-stop signatures either as ordinary digital signatures or as signatures with dual security, where the signers instead of the recipients are secure in an information-theoretic sense. Based on the definition, we proved general lower bounds on the length of the keys and signatures in these schemes.

We presented a general construction of fail-stop signature schemes and showed how to instantiate this construction under either of the well-known assumptions that factoring large integers or computing discrete logarithms is hard. These constructions come quite close to the lower bounds; see Table 2.

There are some recent additional results about fail-stop signature schemes. In [11], it was shown that fail-stop signature schemes exist if statistically hiding bit-commitment schemes of a certain type exist. This implies that they exist if one-way permutations exist, which is a potentially weaker assumption than the existence of claw-free pairs of permutations (see section 1.4). Furthermore, a practical construction from arbitrary collision-resistant hash functions was given, and the existence of certain types of fail-stop signature schemes and bit-commitment schemes was shown to be equivalent. In [33], the open question from section 6.5, whether a fail-stop signature scheme exists where the length of both public keys and signatures is independent of the number N of messages to be signed, was answered positively under a strong computational assumption (with a construction based on the one-way accumulators from [3]).

In conjunction with a theorem from [23] which shows that the length of unconditionally secure signatures (see section 1.5) is linear in the number of participants who can test them (whereas the length of fail-stop signatures and ordinary digital signatures does not depend on this number), our constructions show that fail-stop signatures provide the most viable way of protecting signers unconditionally.

Acknowledgments. It is a pleasure to thank many people for their contributions, primarily, Eugène van Heijst, who was a coauthor of preliminary versions of parts of this paper and would have been a coauthor again if he were still working in this field, and Michael Waidner, who allowed us to use some material from joint publications here. We also thank Ivan Damgård and Andreas Pfitzmann for several good ideas and Joachim Biskup, Gerrit Bleumer, David Chaum, and the anonymous referees for interesting comments.

REFERENCES

[1] M. Bellare and S. Micali, *How to sign given any trapdoor permutation*, J. Assoc. Comput. Mach., 39 (1992), pp. 214–233.

[2] J. Benaloh, *Verifiable secret-ballot elections*, Ph.D. thesis, Yale University, New Haven, CT, 1987.

[3] J. Benaloh and M. de Mare, *One-way accumulators: A decentralized alternative to digital signatures*, in Proc. 1993 Eurocrypt, Lecture Notes in Comput. Sci. 765, Springer-Verlag, Berlin, 1994, pp. 274–285.

[4] G. Bleumer, B. Pfitzmann, and M. Waidner, *A remark on a signature scheme where forgery can be proved*, in Proc. 1990 Eurocrypt, Lecture Notes in Comput. Sci. 473, Springer-Verlag, Berlin, 1991, pp. 441–445.

[5] J. Boyar, D. Chaum, I. Damgård, and T. Pedersen, *Convertible undeniable signatures*, in Proc. 1990 Crypto, Lecture Notes in Comput. Sci. 537, Springer-Verlag, Berlin, 1991, pp. 189–205.

[6] J. Bos, D. Chaum, and G. Purdy, *A voting scheme*, unpublished manuscript, presented at the rump session of 1988 Crypto, Santa Barbara, CA, 1988.

[7] D. Chaum and H. van Antwerpen, *Undeniable signatures*, in Proc. 1989 Crypto, Lecture Notes in Comput Sci. 435, Springer-Verlag, Berlin, 1990, pp. 212–216.

[8] D. Chaum, E. van Heijst, and B. Pfitzmann, *Cryptographically strong undeniable signatures, unconditionally secure for the signer*, in Proc. 1991 Crypto, Lecture Notes in Comput. Sci. 576, Springer-Verlag, Berlin, 1992, pp. 470–484.

[9] D. Chaum and S. Roijakkers, *Unconditionally secure digital signatures*, in Proc. 1990 Crypto, Lecture Notes in Comput. Sci. 537, Springer-Verlag, Berlin, 1991, pp. 206–214.

[10] I. B. Damgård, *Collision free hash functions and public key signature schemes*, in Proc. 1987 Eurocrypt, Lecture Notes in Comput. Sci. 304, Springer-Verlag, Berlin, 1988, pp. 203–216.

[11] I. B. Damgård, T. P. Pedersen, and B. Pfitzmann, *On the existence of statistically hiding bit commitment schemes and fail-stop signatures*, in Proc. 1993 Crypto, Lecture Notes in Comput. Sci. 773, Springer-Verlag, Berlin, 1994, pp. 250–265.

[12] W. Diffie and M. Hellman, *New directions in cryptography*, IEEE Trans. Inform. Theory, 22 (1976), pp. 644–654.

[13] H. Dobbertin, A. Bosselaers, and B. Preneel, *RIPEMD*-160: *A strengthened version of RIPEMD*, in Proc. 3rd Fast Software Encryption Workshop, Lecture Notes in Comput. Sci. 1039, Springer-Verlag, Berlin, 1996, pp. 71–82.

[14] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. II, 2nd ed., John Wiley, New York, 1971.

[15] R. Gallager, *Information Theory and Reliable Communication*, John Wiley, New York, 1968.

[16] E. N. Gilbert, F. J. MacWilliams, and N. J. A. Sloane, *Codes which detect deception*, Bell System Technical J., 53 (1974), pp. 405–424.

[17] O. Goldreich, *Two remarks concerning the Goldwasser–Micali–Rivest signature scheme*, in Proc. 1986 Crypto, Lecture Notes in Comput. Sci. 263, Springer-Verlag, Berlin, 1987, pp. 104–110.

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

[19] S. Goldwasser, S. Micali, and C. Rackoff, *The knowledge complexity of interactive proof systems*, SIAM J. Comput., 18 (1989), pp. 186–207.

[20] S. Goldwasser, S. Micali, and R. L. Rivest, *A digital signature scheme secure against adaptive chosen-message attacks*, SIAM J. Comput., 17 (1988), pp. 281–308.

[21] J. van de Graaf and R. Peralta, *A simple and secure way to show the validity of your public key*, in Proc. 1987 Crypto, Lecture Notes in Comput. Sci. 293, Springer-Verlag, Berlin, 1988, pp. 128–134.

[22] E. van Heyst and T. P. Pedersen, *How to make efficient fail-stop signatures*, in Proc. 1992 Eurocrypt, Lecture Notes in Comput. Sci. 658, Springer-Verlag, Berlin, 1993, pp. 366–377.

[23] E. van Heijst, T. P. Pedersen, and B. Pfitzmann, *New constructions of fail-stop signatures and lower bounds*, in Proc. 1992 Crypto, Lecture Notes in Comput. Sci. 740, Springer-Verlag, Berlin, 1993, pp. 15–30.

[24] N. Koblitz, *Elliptic curve implementation of zero-knowledge blobs*, J. Cryptology, 4 (1991), pp. 207–213.

[25] L. Lamport, *Constructing digital signatures from a one-way function*, Report CSL-98, SRI International, Menlo Park, CA, 1979.

[26] R. C. Merkle, *Protocols for public key cryptosystems*, in Proc. 1980 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, 1980, pp. 122–134.

[27] R. C. Merkle, *Protocols for public key cryptosystems*, in Secure Communications and Asymmetric Cryptosystems, G. J. Simmons, ed., AAAS Selected Symposium 69, Westview Press, Boulder, CO, 1982, pp. 73–104.

[28] R. C. Merkle, *A digital signature based on a conventional encryption function*, in Proc. 1987 Crypto, Lecture Notes in Comput. Sci. 293, Springer-Verlag, Berlin, 1988, pp. 369–378.

[29] R. C. Merkle, *A certified digital signature* (*That antique paper from* 1979), in Proc. 1989 Crypto, Lecture Notes in Comput. Sci. 435, Springer-Verlag, Berlin, 1990, pp. 218–238.

[30] M. Naor and M. Yung, *Universal one-way hash functions and their cryptographic applications*, in Proc. 21st Symposium on Theory of Computing (1989 STOC), ACM, New York, 1989, pp. 33–43.

[31] B. Pfitzmann, *Für den Unterzeichner unbedingt sichere digitale Signaturen und ihre Anwendung*, Diploma thesis, Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe, Karlsruhe, Germany, 1989 (in German; summarized in English in [35]).

[32] B. Pfitzmann, *Fail-stop signatures: Principles and applications*, in Proc. 8th World Conference on Computer Security, Audit, and Control (1991 Compsec), Elsevier, Oxford, UK, 1991, pp. 125–134.

[33] B. Pfitzmann, *Fail-stop signatures without trees*, Hildesheimer Informatik-Berichte 16/94 (ISSN 0941-3014), Institut für Informatik, Universität Hildesheim, Hildesheim, Germany, 1994.

[34] B. Pfitzmann, *Digital Signature Schemes: General Framework and Fail-Stop Signatures*, Lecture Notes in Comput. Sci. 1100, Springer-Verlag, Berlin, 1996.

[35] B. Pfitzmann and M. Waidner, *Formal aspects of fail-stop signatures*, Technical Report 22/90, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, 1990.

[36] B. Pfitzmann and M. Waidner, *Fail-stop signatures and their application*, in Proc. 9th Worldwide Congress on Computer and Communications Security and Protection (1991 Securicom), Paris, 1991, pp. 145–160.

[37] B. Pfitzmann and M. Waidner, *Unconditional Byzantine agreement for any number of faulty processors*, in Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science (1992 STACS), Lecture Notes in Comput. Sci. 577, Springer-Verlag, Berlin, 1992, pp. 339–350.

[38] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Comm. Assoc. Comput. Mach., 21 (1978), pp. 120–126; reprinted in Comm. Assoc. Comput. Mach., 26 (1983), pp. 96–99.

[39] J. Rompel, *One-way functions are necessary and sufficient for secure signatures*, in Proc. 22nd Symposium on Theory of Computing (1990 STOC), ACM, New York, 1990, pp. 387–394.

[40] R. Safavi-Naini and L. Tombak, *Optimal authentication systems*, in Proc. 1993 Eurocrypt, Lecture Notes in Comput. Sci. 765, Springer-Verlag, Berlin, 1994, pp. 12–27.

[41] C. E. Shannon, *Communication in the presence of noise*, Proc. Inst. Radio Engineers, 37 (1949), pp. 10–21.

[42] C. E. Shannon, *Communication theory of secrecy systems*, Bell System Technical J., 28 (1949), pp. 656–715.

[43] M. Waidner and B. Pfitzmann, *The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability*, in Proc. 1989 Eurocrypt, Lecture Notes in Comput. Sci. 434, Springer-Verlag, Berlin, 1990, p. 690.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 331–349, April 1997

002

THE VERTEX-DISJOINT MENGER PROBLEM IN PLANAR GRAPHS*

HEIKE RIPPHAUSEN-LIPA†, DOROTHEA WAGNER‡, AND KARSTEN WEIHE‡

Abstract. We consider the problem of finding a maximum collection of vertex-disjoint paths in undirected, planar graphs from a vertex s to a vertex t. This problem is usually solved using flow techniques, which lead to $\mathcal{O}(nk)$ and $\mathcal{O}(n\sqrt{n})$ running times, respectively, where n is the number of vertices and k the maximum number of vertex-disjoint (s,t)-paths. The best previously known algorithm is based on a divide-and-conquer approach and has running time $\mathcal{O}(n \log n)$. The approach presented here is completely different from these methods and yields a linear-time algorithm.

Key words. graph algorithms, disjoint paths, planar graphs

AMS subject classifications. 08C85, 68Q20, 05C38, 68R10, 90C35

PII. S0097539793253565

1. Introduction.

The general Menger problem is a classical problem in both structural and algorithmic graph theory [2, 9, 15]. In general, it consists in finding the maximum number of vertex-disjoint or edge-disjoint paths in a graph from some designated vertex to another one. This problem has many practical applications, for example, in the design of reliable communication networks and the design of integrated circuits. Because of this practical background, the special case where the underlying graph is undirected and planar is particularly interesting.

In this paper, we consider the *vertex-disjoint* Menger problem in undirected, planar graphs. This means the following. Let $G = (V, E)$ be an undirected, planar graph, and let $s, t \in V$, $s \neq t$. A vertex $v \in V$ is called an *inner vertex* of path p if p contains v and v is neither of the end vertices of p. Two (s,t)-paths are said to be *internally vertex-disjoint*, or simply *vertex-disjoint*, if neither path contains an inner vertex of the other path. The problem is to find as many pairwise vertex-disjoint (s,t)-paths as possible.

The general vertex-disjoint Menger problem, directed or undirected, can be solved by reduction to a bipartite maximum-flow problem with unit capacities [3, 5]. When using the famous labeling algorithm [1, 4], this approach yields an $\mathcal{O}(km)$ algorithm, where k is the maximum number of (s,t)-paths and m is the number of edges. When the algorithm of Malhotra, Kumar, and Maheshwari is used instead, $\mathcal{O}(n^{1/2}m)$ is achieved, where n is the number of vertices [10, 11].

For planar graphs, these upper bounds reduce to $\mathcal{O}(kn)$ and $\mathcal{O}(n^{3/2})$, respectively, since $m \in \mathcal{O}(n)$. The best previously known algorithm for the special case of undirected, planar graphs was introduced by Suzuki, Akama, and Nishizeki and requires $\mathcal{O}(n \log n)$ time [16]. This algorithm is based on divide-and-conquer techniques that were originally developed in [13] and [6] for the maximum-flow problem in undirected planar graphs.

* Received by the editors August 12, 1993; accepted for publication (in revised form) May 4, 1995. This research was supported by Deutsche Forschungsgemeinschaft grant Mö 446/1–3.
http://www.siam.org/journals/sicomp/26-2/25356.html

† Technische Universität Berlin, Sekr. MA 6–1, Strasse des 17.Juni 136, 10623 Berlin, Germany (ripphaus@math.tu-berlin.de).

‡ Fakultät für Mathematik und Informatik, Universität Konstanz, Postfach 5560/D188, 78434 Konstanz, Germany ({dorothea.wagner, karsten.weihe}@uni-konstanz.de, http://www.informatik.uni-konstanz.de/~{wagner, weihe}). (Any communication should be addressed to the third author.)

In this paper, we present an *asymptotically optimum* algorithm for the vertex-disjoint Menger problem, that is, with running time $\mathcal{O}(n)$. Moreover, we shall state that the algorithm is not too hard to implement and that the constant factor for the linear-time bound is reasonably small.

Besides the practical background mentioned above, our result can be applied to design a linear-time algorithm for a special case of the problem of finding vertex-disjoint Steiner trees for several sets of vertices. Namely, the underlying graph is undirected and planar and each terminal is incident to either of two designated faces. This special case was investigated by Suzuki, Akama, and Nishizeki [16]. They proposed an $\mathcal{O}(n \log n)$ algorithm, where the solution of an auxiliary vertex-disjoint Menger problem is the bottleneck task. In [14], a linear-time algorithm for this particular Steiner tree problem was introduced which uses the new linear-time algorithm for the vertex-disjoint Menger problem as a subroutine.

When the number of (s,t)-paths, say k, can be fixed in advance, the vertex-disjoint Menger problem in planar graphs can trivially be solved in linear time using the flow-theoretic approach. (Even vertex-disjoint paths between k different pairs of vertices, k fixed, can be found in linear time [12].)

On the other hand, the problem is straightforwardly solvable in linear time whenever an instance is (s,t)-planar, which means that s and t have a face of G in common. In fact, in this case, we may apply the following simple algorithm: For convenience, G is embedded such that s is the lowest and t the highest vertex, respectively. In the beginning, the solution is empty. Then while there is a path from s to t in G, add the "rightmost" one to the solution and (temporarily) drop all vertices and edges on this path and on its right side.

In contrast, in this paper, k is unbounded and s and t may be in general position.

The straightforward algorithm for the (s,t)-planar case demonstrates a special technique for routing and flow problems in planar graphs, which is the basis of our algorithm as well. Throughout the paper, we will call this technique *right-first search*. (In previous work, it is also called the *uppermost path algorithm* [7].) This is a depth-first search where in each search step, all arcs leaving the current vertex are searched "from right to left." More precisely, in each step, the next candidate arc for going forward, if any, is the counterclockwise next arc after the leading arc in the incidence list of the leading vertex. Clearly, the straightforward algorithm for the (s,t)-planar case amounts to a repeated application of right-first search, where each search starts with s and disregards all vertices and arcs hit in previous searches.

Very recently, the *edge*-disjoint version of the problem considered here could be solved in linear time as well [17]. A generalization of the vertex-disjoint Menger problem, where each vertex may belong to more than one path but each vertex has a fixed capacity, was also considered by Khuller and Naor [8].

The paper is organized as follows. In section 2, the new algorithm to solve the planar, vertex-disjoint Menger problem in linear time is introduced. Then in section 3, we will prove its correctness. In section 4, we will prove some properties of the results of our algorithm, which are useful for applications to vertex-disjoint Steiner trees such as those mentioned above.

2. The algorithm. We will now introduce a linear-time algorithm for the vertex-disjoint Menger problem. A formal description of the algorithm is given in Fig. 2, and an example is illustrated in Figs. 3–6. For technical reasons, we will consider a certain *directed* version of the Menger problem instead of the *undirected* problem itself. In contrast to *undirected edges*, which we denote by $\{v,w\}$, we denote the *arc directed*

from v to w by (v, w).

DIRECTED VERSION. *We transform the input graph $G = (V, E)$ into a directed graph $G_0 = (V, A_0)$: we replace each edge $\{v, w\} \in E$ with $v, w \in V \setminus \{s, t\}$ by the arcs (v, w) and (w, v), each edge $\{s, v\} \in E$ by (s, v) only, and each edge $\{v, t\}$ by (v, t) only. The problem is now to find as many vertex-disjoint (s, t)-paths as possible, that is, paths* directed *from s to t.*

Obviously, there is a linear-time algorithm for the original Menger problem if there is a linear-time algorithm for this directed version.

OVERALL ASSUMPTIONS. *Without loss of generality, we assume that t is on the outer face boundary, that G is connected, and that G is embedded in the plane* (*at least* combinatorially*, which means that all edges are sorted according to the same fixed, geometric embedding in the plane*).

Let $a_1, \ldots, a_k$ denote the arcs leaving s, in arbitrary order. The core of our algorithm for the directed version is a loop over $1, \ldots, k$. In the ith iteration, we try to draw a cycle-free (s, t)-path. This is done by a right-first search which starts with a_i. (The final path resulting from the ith iteration, if any, need not start with a_i.) The ith iteration is finished when we either reach t or return to s. Obviously, in the latter case, this is done by a backtrack step since by construction of G_0, no arc enters s.

During such a right-first search, we have to consider conflicts of the search path with itself and with paths drawn in previous searches. That is, the search meets a vertex which already belongs to some path. In other words, a conflict is a situation where one vertex has *two* in-going arcs. We resolve each conflict through a backtrack step with one of these in-going arcs. The main idea is to handle all conflicts in a way that enables us to *remove each arc from G_0 once we perform a backtrack step on it.* Since any step of such a right-first search is a forward step or a backtrack step, the number of search steps *in total* is thus linear in the size of the instance. Moreover, we will show how to realize these search steps in constant time, which immediately yields the overall worst-case bound. In principle, the following conflicts may occur (see Fig. 1):

1. The search path hits another path from the left side.
2. The search path hits itself from the left side.
3. The search path hits another path from the right side.
4. The search path hits itself from the right side.

The second and fourth cases mean that the leading vertices and arcs of the search path form a cycle. In the second case, this is a clockwise cycle if and only if s is enclosed in the cycle, and vice versa in the fourth case.

In the first two cases, we simply perform a backtrack step with the search path, where we remove the leading arc not only from the search path but also from A_0. After that, we continue our right-first search with the remainder of the search path. In particular, we need not distinguish between the first and second types of conflict in our algorithm.

In the third case, we resolve the conflict in the following way (see Fig. 1, part (3)). Let r denote the search path and let p denote the path hit by r at, say, vertex v. Moreover, let p_1 be the subpath of p from s to v and p_2 be the subpath of p from v to t. We now concatenate r with p_2 and let p_1 be the "new" search path. After that, the situation is the same as in the first case, and we resolve the conflict in the same manner. That is, we then perform a backtrack step with the new search path p_1, remove the leading arc of p_1 from G_0, and continue our search with the remainder of p_1.

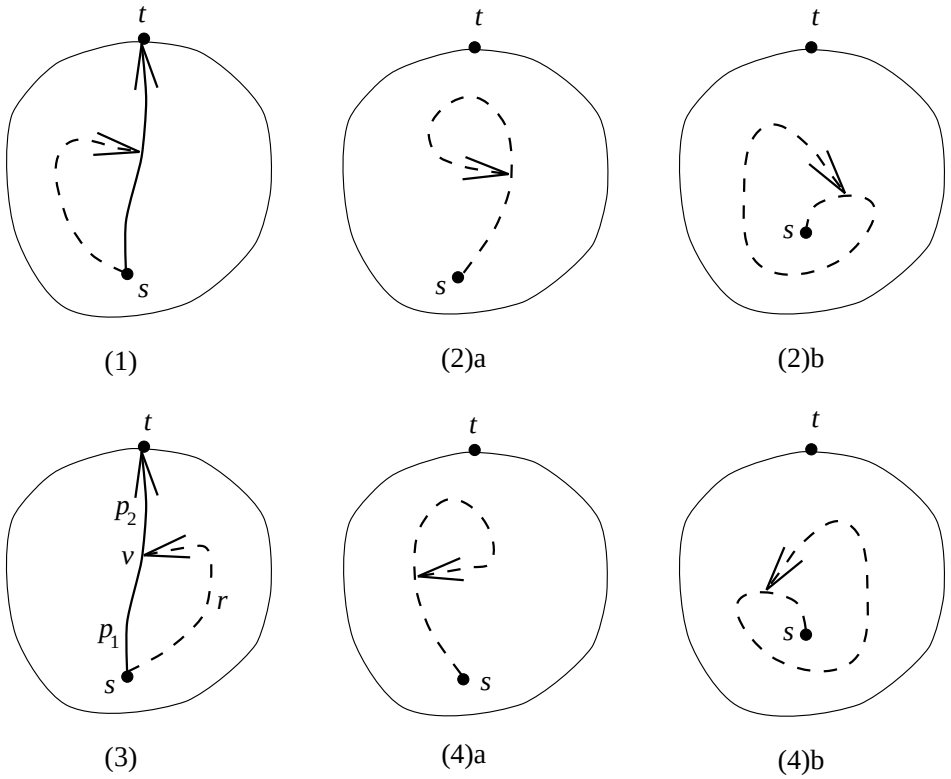


FIG. 1. *The four different cases of conflicts. The search path is dashed, and the other path involved in the conflict, if any, is solid.*

We are not able to cope with conflicts of the fourth type. Therefore, we want to avoid those conflicts *in advance*. In particular, we need not distinguish between the third and fourth types of conflict in our algorithm as well. To avoid all conflicts of the fourth type in advance, we maintain a global *time counter* and, for each vertex, a local *time stamp*. The global time counter is increased by 1 whenever the search path changes, that is, when either a new iteration of the overall loop starts or a conflict of the third type occurs. The time stamp of a vertex is set to the current value of the global time counter whenever this vertex becomes the leading vertex of the search path (in fact, before doing anything else with this vertex).

By the following modification, we prevent all conflicts of the fourth type. Consider a single forward step of the algorithm, let v be the leading vertex of the search path, and let (v, w) be the arc chosen to leave v. First, we compare the time stamps of v and w and make a case distinction. If the time stamps of v and w are different, we actually use (v, w) for going forward. On the other hand, if the time stamps of v and w are equal, we do *not* use (v, w) for going forward in our search, but we remove this arc *immediately*. In this case, we proceed with the arc leaving v that appears next after (v, w) in right-first order (if any).

In our correctness proof, we will show in particular the following two facts to justify this procedure (see Figs. 7 and 8):

1. If a conflict of the fourth type were to occur, it would be preceded and thus "announced" by a conflict of the second type where the same cycle is involved, but

PROCEDURE 2.1. *make_single_step* (VAR $v \in V$);
time_stamp $[v]$:= *time_counter*;
IF the search path hits some path at v from the left side
THEN (* conflict types 1 and 2 *) remove the last arc, say (w, v), from the search path and set $v := w$;
ELSEIF the search path hits some path at v from the right side
THEN (* conflict type 3 *)
let p_1 be the subpath of this path from s to v and let p_2 be the remaining subpath;
concatenate the search path with p_2;
make p_1 the new search path;
remove the last arc, say (w, v), from p_1 and set $v := w$;
time_counter := *time_counter* + 1;
ELSEIF at least one arc leaving v is not yet searched (except of the reverse of the in-going arc)
THEN let (v, w) be the counterclockwise first such arc after the in-going arc around v;
IF *time_stamp* $[v]$ = *time_stamp* $[w]$
THEN remove (v, w)
ELSE append (v, w) to the search path and set $v := w$;
ELSE remove the last arc, say (w, v), from the search path and set $v := w$;

PROCEDURE 2.2. *right_first_search* $((s, u) \in A)$;
time_counter := *time_counter* + 1;
let $s \to u$ be the new search path;
$v := u$;
REPEAT
make_single_step (v);
UNTIL $v \in \{s, t\}$;

ALGORITHM 2.3.
time_counter := 0;
time_stamp $[s]$:= 1;
FOR $v \in V \setminus \{s\}$ DO *time_stamp* $[v]$:= 0;
FOR $i := 1$ TO k DO $right\_first\_search(a_i)$;

FIG. 2. *Formal description of the algorithm.*

in the reverse direction. Let (w, v) be the arc removed to resolve this announcing conflict.

2. When we later select (v, w) for going forward from v, the global time counter has not changed in the meantime.

Hence (v, w) is removed at the latter stage and *not* used for going forward from v. Clearly, this makes the "announced" conflict of the fourth type impossible because (v, w) is an arc of the cycle involved.

The reader might wonder why (v, w) is not removed at the former stage when (w, v) is removed. For this, however, we must know whether the conflict at the former stage is of the first or the second type because, of course, we must not remove (v, w) in a conflict of the first type. This means that we had to maintain additional information to distinguish the individual paths from each other. However, updating

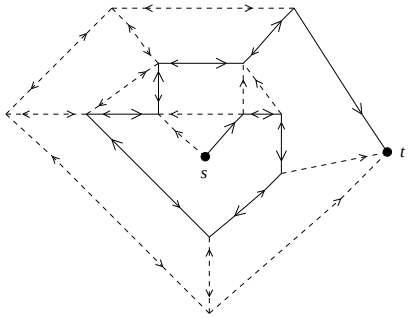


FIG. 3. *The directed, symmetric graph $G_0 = (V, A_0)$ after the first iteration, along with the path p drawn in the first iteration (solid). Two opposite arrows indicate the presence of two opposite arcs. The orientation of p from s to t is indicated by emphasized arrows. Note that there is no longer an arc leaving p on the right side because all of these arcs have been searched—and hence removed—in the first iteration.*

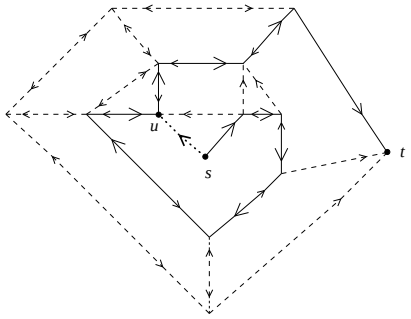


FIG. 4. *After one step of the second iteration. The search path, that is, $s \to u$ (dotted), hits the path drawn in the first iteration from the right side at vertex u.*

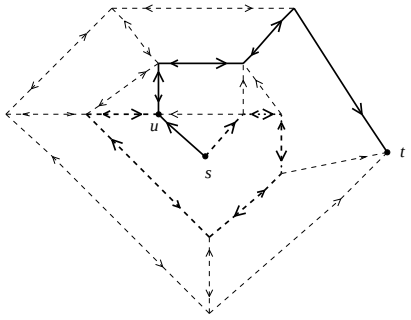


FIG. 5. *The subpath of p from s to u becomes the new search path (dotted), and the previous search path is concatenated with the subpath of p from u to t (solid).*

this information after each conflict of the third type takes too much time. In contrast, the time stamps can be maintained efficiently enough.

The principle of announcing conflicts is the reason why the algorithm does not apply to directed planar graphs in general. In fact, we need as a prerequisite that for each clockwise cycle in the graph, the reverse counterclockwise cycle exists as well.

The following theorem summarizes the discussion.

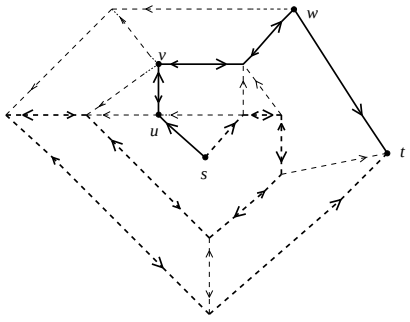


FIG. 6. *All further conflicts in the second iteration are conflicts where the search path hits the other path from the left side: twice at v and, later, once at w.*

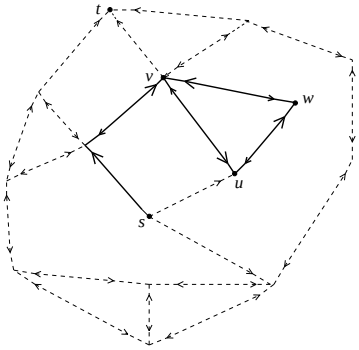


FIG. 7. *Another instance, where a conflict of the second type occurs (cycle $v \to u \to w \to v$) which "announces" a conflict of the fourth type with the reverse cycle involved.*

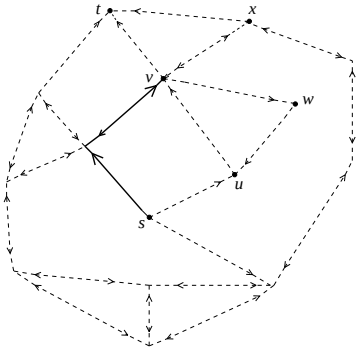


FIG. 8. *After three backtrack steps. Now (v, w) is the next arc to be considered. Note that the time stamps of v and w are equal at this stage, which means that (v, w) is not chosen for going forward but is removed from G_0 immediately, and (v, x) is chosen instead. If the algorithm were to choose (v, w) and proceed as usual, the search would now run into the cycle $v \to w \to u \to v$. This would be a disaster because resolving this conflict by removing (u, v) decreases the cardinality of an optimum solution. On the other hand, removing the other in-going arc of v would leave a cycle (namely $v \to w \to u \to v$) in the solution. This cycle* must *be cracked later on because we need (u, v) for any optimum solution. However, detecting and cracking cycles like this throughout the rest of the algorithm seems hopeless.*

THEOREM 2.4. *Algorithm* 2.3 *can be implemented such that it requires linear time in the worst case.*

Proof. For each vertex v, we manage a cyclic incidence list, where each item corresponds to an arc leaving this vertex and its reverse arc. Of course, there are pointers between corresponding items in the incidence lists of adjacent vertices. This enables us to perform each forward and backtrack/remove step in constant time. It remains to show how to perform two further tasks in constant time: finding the next arc for going forward and determining whether a conflict is from the left side (type 1 or 2) or the right side (type 3 or 4). Remember that we must distinguish neither type 1 from type 2 nor type 3 from type 4.

To see how to find the next possibility for going forward from $v \in V \setminus \{s,t\}$ in constant time, we note that at any stage, the set of all arcs leaving v that have not yet been dropped forms a connected interval of the original incidence list of v and the next arc to be considered is always the counterclockwise first arc in this interval. Clearly, maintaining a variable pointer to this arc does the job.

Now we show how to decide in constant time whether a conflict is from the left or the right side. To this end, in the overall initialization phase, we number the arcs incident to vertex $v \in V \setminus \{s,t\}$ clockwise, starting anywhere. Additionally, we store the number, say, i_v of the current in-going arc and the number, say, j_v of the current out-going arc. Suppose a conflict occurs at v, and let l be the number of the leading arc of the search path with respect to the numbering around v. If $i_v < j_v$, this is a conflict from the *left* side if and only if $i_v < l < j_v$, and if $i_v > j_v$, it is a conflict from the *right* side if and only if $i_v > l > j_v$. Altogether, this proves Theorem 2.4. □

3. Correctness. Essentially, the correctness proof consists of three parts, which are handled separately in three subsections. Before going into the details, we first outline the ideas behind our proof.

Let $\overline{a}_1, \ldots, \overline{a}_m$ be the arcs that are removed *in this order* during the algorithm. Further, let $A_i := A_{i-1} \setminus \{\overline{a}_i\}$ for $i = 1, \ldots, m$. (Recall that $G_0 = (V, A_0)$ is the initial directed graph constructed from G in the beginning of section 2.) Finally, let $G_i := (V, A_i)$, $i = 1, \ldots, m$, that is, G_i is the rest of G_0 after $\overline{a}_1, \ldots, \overline{a}_i$ are removed.

Notice that the final set of paths provided by the algorithm is maximum for G_m because any arc of G_0 that leaves s and still belongs to G_m is occupied by one of the final paths. However, we have to show that our solution is optimum for G_0, not for G_m. For this, we prove the following invariant facts by induction on the forward and backtrack/remove steps of the algorithm. The second fact immediately yields the claim.

INVARIANT 3.1. *The following two facts are maintained as invariants by Algorithm* 2.3.

1. *No conflict of the fourth type occurs.*
2. *Optimum solutions in G_0, G_1, $G_2, \ldots, G_m$, respectively, have the same cardinality. In other words, for each $i = 1, \ldots, m$, there is an optimum solution for G_{i-1} that does not contain $\overline{a}_i$.*

In section 3.1, we show that Invariant 3.1(1) is maintained at least as long as Invariant 3.1(2) is. Of course, Invariant 3.1(2) is trivially maintained when $\overline{a}_i$ is a dead end, that is, no arc leaves the head of $\overline{a}_i$ in G_{i-1}. Therefore, it suffices to show maintenance of Invariant 3.1(2) for the case where $\overline{a}_i$ is removed either because of a conflict or because of equal time stamps.

However, our case distinction is slightly different, mainly for proof-technical rea-

sons. Note that often an arc (v, w) that is removed because $time\_stamp[v] = time\_stamp[w]$ would be removed anyway in the very next search step, namely because of a conflict at w or because w is a dead end. An arc (v, w) is called *unusual* if it is removed because $time\_stamp[v] = time\_stamp[w]$ at some stage of the algorithm but would *not* have been removed anyway in the very next search step. In any other case, it is called *usual*. Maintenance of Invariant 3.1(2) is proved separately for unusual arcs in section 3.2 and for usual arcs in section 3.3. The former proof relies on the assumption that Invariant 3.1(1) is valid.

Sections 3.1 and 3.2 are necessary to prove that our procedure for avoiding conflicts of the fourth type works well. The heart of the proof is section 3.3. To get an idea of this, remember the simple algorithm for the (s, t)-planar case given in the introduction. This algorithm works because it always "packs" the paths as tightly to the right boundary as possible. That is, no path can be drawn more to the right unless another path is removed. The key insight for the *non*-(s, t)-planar case is that, in principle, we may use a very similar argument, although there is no such boundary to lean on at all. In fact, in some sense, such boundaries are "constructed" by Algorithm 2.3 during its execution. This is formalized in the following lemma.

LEMMA 3.2. *At any stage of Algorithm* 2.3*, no arc leaves any path on the right side.*

Proof. All of those arcs have been searched and hence removed at previous stages. □

3.1. Proof of Invariant 3.1(1). In this section, we will show by contradiction that no conflict of the fourth type occurs during the execution of Algorithm 2.3 as long as Invariant 3.1(2) is valid. Therefore, throughout this section we assume that at least one conflict of the fourth type *does* occur. Let C be the cycle formed by the leading part of the search path.

Note that by construction of G_0, neither s nor t belongs to C because no arc enters s and no arc leaves t. Moreover, since t is without loss of generality assumed to be incident to the outer face, t is outside C.

LEMMA 3.3. *The source s cannot be inside C as long as Invariant* 3.1(1) *is valid.* (*Fig.* 1*, part* 4(b)).

Proof. Suppose s is inside C (see Fig. 9).

Lemma 3.2 implies that at this stage, there is no longer an arc pointing from C to the exterior of C. Since t belongs to the exterior of C, there is no longer an (s, t)-path at this stage. This means that the cardinality of an optimum situation is zero for the current graph. However, since G_0 is connected, the cardinality of an optimum solution for G_0 is positive, but this contradicts Invariant 3.1(2) for the stage immediately before the assumed conflict occurs. □

In the rest of this section, we prove that the case where s is outside C (see Fig. 1, part 4(a)) is also impossible. Therefore, for a contradiction, let S_{first} denote the first stage where a conflict of the fourth type occurs with s outside the cycle C involved. Let vertex w_0 be the leading vertex of the search path at that stage and let $C = (w_0 \to w_1 \to w_2 \to \cdots \to w_{l-1} \to w_l = w_0)$. We will investigate the "past history" of the conflict at stage S_{first}.

LEMMA 3.4. *For all $i = 1, \ldots, l-1$, occupying arc (w_{i-1}, w_i) did not cause a conflict.*

Proof. Let S be the stage where (w_{i-1}, w_i) is occupied. Suppose this causes a conflict at w_i. This is no conflict of the first or the second type because otherwise the arc occupied last, namely (w_{i-1}, w_i), would be removed at S to resolve the conflict,

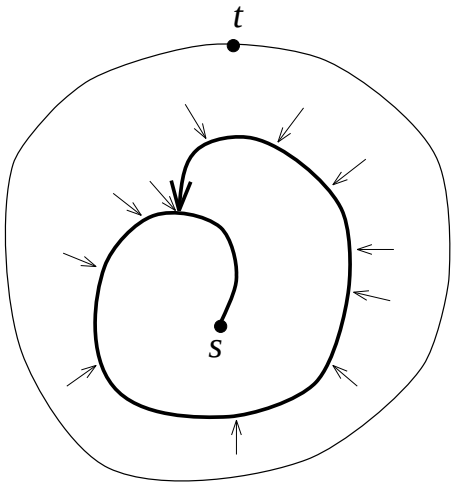


FIG. 9. *The situation in the proof of Lemma* 3.3. *No arc points from the cycle C to its exterior, only possibly the other way round.*

and this would contradict the assumed conflict at stage S_{first}. Moreover, this is not a conflict of the fourth type either, for the following reason: the assumption that s is inside the cycle formed by the leading arcs of the search path contradicts Lemma 3.3, and the assumption that s is outside this cycle contradicts the specific choice of S_{first} to be the first stage where such a conflict occurs. In summary, this is a conflict of the third type.

Let p denote the (s,t)-path hit by the search path at stage S, and let a be the arc of p leaving w_i. Then either a belongs to the exterior of $\mathcal{C}$ or the interior of $\mathcal{C}$ or is just (w_i, w_{i+1}). We consider all three cases separately (see Fig. 10).

Case 1. The arc a belongs to the exterior of $\mathcal{C}$ (see Fig. 10, part (1)).

Since this is a conflict of the third type, (w_{i-1}, w_i) appears counterclockwise around w_i after the in-going arc and before the out-going arc of w_i with respect to p. Since a is this out-going arc, (w_i, w_{i+1}) leaves p on the right side. By Lemma 3.2, (w_i, w_{i+1}) has already been removed at stage S, which contradicts the assumed situation at the later stage S_{first}.

Case 2. The arc a belongs to the interior of $\mathcal{C}$ (see Fig. 10, parts (2) and (3)).

Note that p must leave $\mathcal{C}$ and enter the exterior of $\mathcal{C}$ at least once after w_i in order to reach t. (Recall that t belongs to the exterior of any directed cycle of G_0.) In particular, let w_j be the first vertex after w_i on p where this happens. Then the in-going arc of w_j with respect to p either belongs to the interior of $\mathcal{C}$ or is just (w_{j-1}, w_j) or (w_{j+1}, w_j) (see Fig. 10, part (2) for the first case and Fig. 10, part (3) for the third case). In the first two cases, (w_j, w_{j+1}) has already been removed until stage S because of Lemma 3.2. Clearly, this is again a contradiction. Therefore, consider the last case. Then there is $\mu \in \{1, \ldots, l\}$ such that $(w_\mu, w_{\mu-1})$, $(w_{\mu-1}, w_{\mu-2}), \ldots, (w_{j+1}, w_j)$ belong to p but $(w_{\mu+1}, w_\mu)$ does not. By the specific choice of w_j, the in-going arc of w_μ with respect to p does not belong to the exterior but to the interior of $\mathcal{C}$. Therefore, $(w_\mu, w_{\mu+1})$ has already been removed because of Lemma 3.2. This is the same contradiction as in the first case.

Case 3. $a = (w_i, w_{i+1})$ (see Fig. 10, part (4)).

Then there is $\nu \in \{1, \ldots, l\}$ such that all of (w_i, w_{i+1}), $(w_{i+1}, w_{i+2}), \ldots, (w_{\nu-1}, w_\nu)$

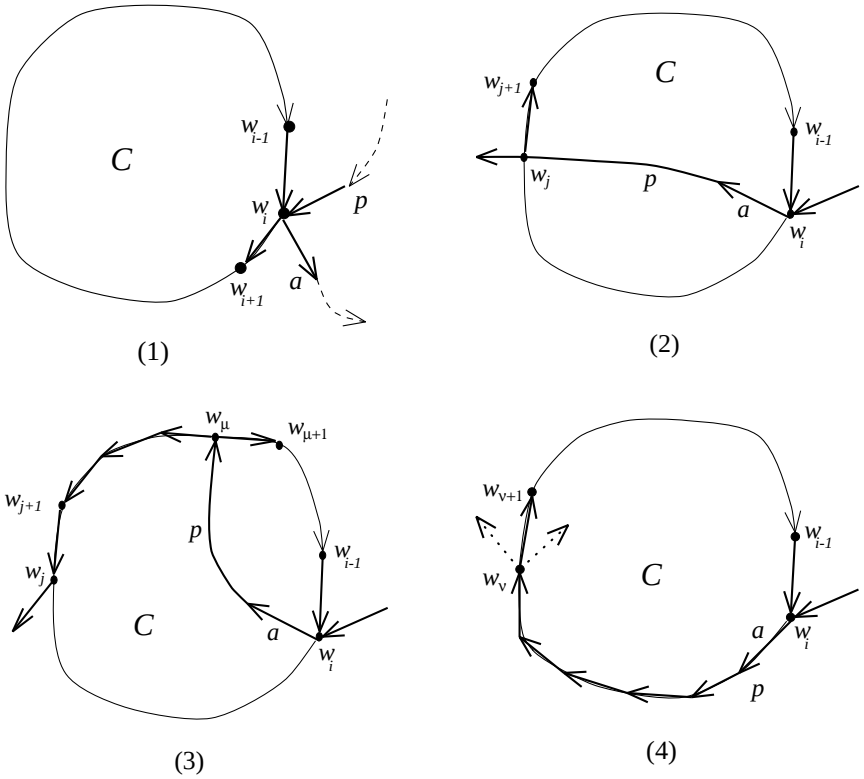


FIG. 10. *Typical situations in the different cases considered in the proof of Lemma* 3.4 *for conflicts of the third type: Case* 1: (1); *Case* 2: (2) *and* (3); *Case* 3: (4).

belong to p but $(w_\nu, w_{\nu+1})$ does not. Thus the out-going arc of w_ν with respect to p belongs to either the exterior of $\mathcal{C}$ or the interior of $\mathcal{C}$. If this arc belongs to the *exterior* of $\mathcal{C}$, then $(w_\nu, w_{\nu+1})$ has already been removed by Lemma 3.2. On the other hand, if this arc belongs to the *interior* of $\mathcal{C}$, the proof is literally the same as for Case 2 (just replace i by ν). □

To continue, we need a more general, technical lemma, which will also be used later. A vertex is called *released* at some stage if it was occupied by some path immediately before this stage and is not occupied anymore (because of a backtrack step in fact).

LEMMA 3.5. *When a vertex v is released at some stage S, all arcs leaving v have been removed even before, except possibly the reverse of the current in-going arc. If the in-going arc has changed at least once before S,* all *arcs have been removed.*

Proof. When vertex v is released, all of the arcs that we could use for going forward from v have already been searched and hence removed. This means all arcs leaving v except possibly the reverse of the in-going arc. This proves the first claim. Therefore, consider the second claim. Because of the first claim, we only have to show that the reverse of the in-going arc is removed as well. Let a denote the in-going arc of v at stage S. Suppose the in-going arc of v has changed at least once at some stage S' preceding stage S. Then either v is released at S' or a conflict of the third type occurs at this stage. In particular, let S' be the last such stage before S and let a' be the arc entering v that was the in-going arc of v at stage S'.

If v was released at this stage, the first claim implies that all arcs leaving v except the reverse of a' have been removed before S'. In particular, the reverse of a has also been removed, which implies the second claim. Now assume there was a conflict of the third type at S'. Then there is already an (s,t)-path p occupying v, and a points to p at v from the right side. Now Lemma 3.2 implies that the reverse of a has been removed as well. □

LEMMA 3.6. *All arcs* (w_i, w_{i-1}), $i = 1, \ldots, l$, *are removed before stage* S_{first}.

Proof. The proof is by induction on $i = l, l-1, \ldots, 1$. Because of Lemma 3.2, arc (w_l, w_{l-1}) is removed before stage S_{first} since this arc leaves the search path of stage S_{first} on the right side. Therefore, assume that arc (w_i, w_{i-1}), $i > 1$, is removed at some stage S_i preceding S_{first}. It suffices to show that S_{i-1} precedes S_i, that is, (w_{i-1}, w_{i-2}) is removed even before (w_i, w_{i-1}). To this end, we distinguish the different possible reasons for removing (w_i, w_{i-1}) from each other.

First, suppose (w_i, w_{i-1}) is removed as an unusual arc. Since w_i has a positive time stamp at S_i, the time stamp of w_{i-1} at S_i must also be positive. Hence there is a stage S preceding S_i where w_{i-1} was the leading vertex of the search path. In particular, let S be the last such stage. Since (w_i, w_{i-1}) is removed at S_i as an unusual arc, w_{i-1} is not occupied at stage S_i. Therefore, w_{i-1} was released at stage S. Clearly, (w_i, w_{i-1}) cannot be the leading arc of the search path at stage S because otherwise (w_i, w_{i-1}) would be removed even at S, not at S_i. Thus by the first part of Lemma 3.5, (w_{i-1}, w_i) has already been removed at S, which contradicts the assumed conflict at S_{first}.

Now suppose that (w_i, w_{i-1}) is removed because of a conflict (or, completely analogously, we have $\mathit{time\_stamp}\,[w_i] = \mathit{time\_stamp}\,[w_{i-1}]$, but (w_i, w_{i-1}) would be removed anyway in the very next step because of a conflict). Then w_{i-1} is occupied by some path at stage S_i and, by Lemma 3.4, w_{i-1} is released at some intermediate stage S between S_i and S_{first}. Since (w_i, w_{i-1}) no longer exists at stage S, the leading arc of the search path must be another arc entering w_{i-1} at that stage. Consequently, the first part of Lemma 3.5 implies that (w_{i-1}, w_i) has already been removed at S, which is again a contradiction to the assumed situation at S_{first}.

Finally, if removing (w_i, w_{i-1}) releases w_{i-1} (or, analogously, the time stamps are equal but w_{i-1} had been released anyway in the very next step), the claim that S_{i-1} precedes S_i is immediate from the first part of Lemma 3.5.

This completes the case distinction and the proof of Lemma 3.6. □

Now we are able to prove that any conflict of the fourth type is "announced" by the reverse conflict of the second type.

LEMMA 3.7. *Arc* (w_1, w_0) *is removed before stage* S_{first} *because of a conflict of the second type at* w_0*, with the reverse of* C *being the cycle involved.*

Proof. Consider the stage S_1 where (w_1, w_0) is removed. By Lemma 3.6, S_1 precedes S_{first}. Suppose (w_1, w_0) is *not* removed because of a conflict of the second type at w_0 with the reverse of C involved. Then at stage S_1, there must be a vertex w_i, $i \in \{1, \ldots, l-1\}$, such that all of $(w_i, w_{i-1}), (w_{i-1}, w_{i-2}), \ldots, (w_1, w_0)$ belong to the search path but (w_{i+1}, w_i) does not. Because of Lemma 3.4, vertex w_i is released at some intermediate stage between S_1 and S_{first}. Lemma 3.5 implies that (w_i, w_{i+1}) has already been removed before S_{first}, for the following reason. If the in-going arc of w_i has never changed, all arcs leaving w_i except the reverse of the in-going arc are removed, and by assumption, the in-going arc of w_i at S_1 is *not* (w_{i+1}, w_i). On the other hand, if the in-going arc of w_i *did* change, the second claim of Lemma 3.5 implies that *all* arcs leaving w_i, and (w_i, w_{i+1}) in particular, have been removed. Clearly, the

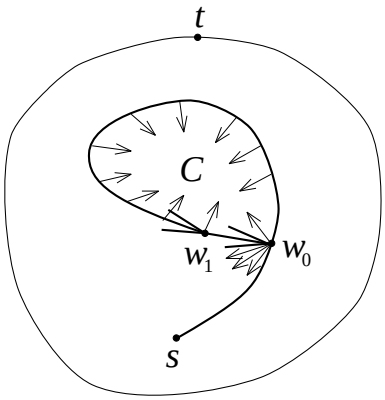


FIG. 11. *The situation in the proof of Lemma* 3.8. *By Lemma* 3.2, *no arc points from* C *to the* exterior *of* C (*except possibly arcs leaving* w_0 *as indicated*). *Hence the search does not reenter the exterior of* C *before* (w_0, w_1) *is seen.*

fact that (w_i, w_{i+1}) is removed before S_{first} is a contradiction. □

Lemma 3.7 now enables us to prove that no conflict of the fourth type occurs at all.

LEMMA 3.8. *While Invariant* 3.1(2) *is valid, no conflict of the fourth type occurs.*

Proof. Suppose a conflict of the fourth type *does* occur. Let C again be the cycle involved. Then s is outside C by Lemma 3.3 (see Fig. 11). Because of our right-first strategy, Lemma 3.2 implies that after (w_1, w_0) is removed, the search cannot enter the exterior of C before (w_0, w_1) is seen. Therefore, we consider (w_0, w_1) before we reach t or any path different from the search path. Hence the global time counter cannot change in the meantime unless a conflict of the fourth type has occurred before S_{first}. (Clearly, the global time counter would change if a conflict of the fourth types occurred because the algorithm does not distinguish between the third and fourth types of conflict.) However, this contradicts the definition of S_{first}. Consequently, (w_0, w_1) is removed because $time\_stamp\,[w_0] = time\_stamp\,[w_1]$, which makes the assumed situation at S_{first} impossible. □

3.2. Unusual arcs. We will now prove that for $i = 1, \ldots, m$, there is an optimum solution for G_{i-1} that does not contain $\overline{a}_i = (v, w)$, where $\overline{a}_i$ is an unusual arc. More precisely, we will show that there is no *cycle-free* (s, t)-path in G_{i-1} that contains $\overline{a}_i$. Clearly, this suffices.

Throughout this section, we need some additional terminology. Let S_2 be the stage where $\overline{a}_i = (v, w)$ is removed. Since v clearly has a positive time stamp at stage S_2 and since (v, w) is unusual, w also has a positive time stamp at S_2. Therefore, w was at least once the leading vertex of the search path before S_2. Let S_1 be the last stage before S_2 where w was the leading vertex of the search path. Moreover, since (v, w) is unusual, w was released at stage S_1. Thus Lemma 3.5 implies that (w, v) has been removed even before S_1 because (v, w) was never before the in-going arc of w. Let S_0 be the stage immediately after (w, v) is removed. In summary, S_0 precedes or equals S_1 and S_1 strictly precedes S_2.

Consider the search paths r_0, r_1, and r_2, at stages S_0, S_1, and S_2, respectively. All three paths start with s, r_0 and r_1 end with w, and r_2 ends with v. The paths r_1 and r_2 coincide up to, say, vertex u. Notice that $u \neq s$ because otherwise the global

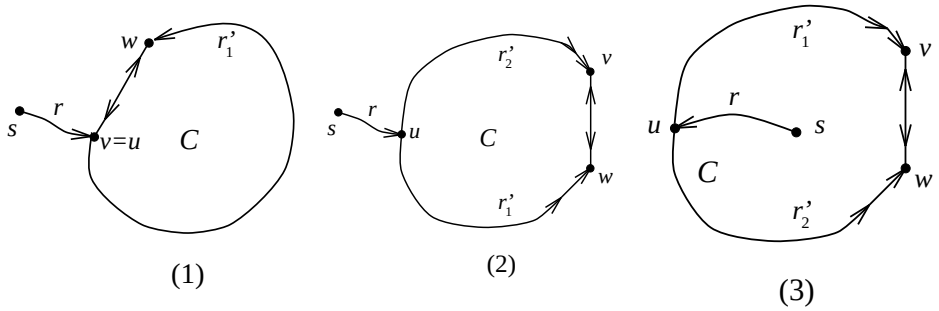


FIG. 12. *The situations in Lemmas* 3.10 (*part* (1)) *and* 3.11 (*part* (2)). *Lemma* 3.9 *states that the situation in part* (3) *is impossible.*

time counter would change between stages S_1 and S_2 and hence (v, w) would not be unusual. Let r be the common subpath from s to u, let r'_1 be the remainder of r_1, and let r'_2 be the remainder of r_2. Finally, let C be the cycle formed by r'_1, r'_2, and $\overline{a}_i$ (see Fig. 12). Lemma 3.5 implies that r'_1 and r'_2 are vertex-disjoint except for u, which means that C is actually a single, simple cycle.

LEMMA 3.9. *Vertex s belongs to the exterior of cycle C.*

Proof. Assume that s is not in the exterior of C. Then s either lies on C or is in the interior of C. The former case implies $u = s$, which contradicts the above remarks. Therefore, focus on the latter case. In the latter case, any (s, t)-path must contain an arc a that leaves C and points to the exterior of C because t is in the exterior of C. However, any such arc is considered for going forward between stages S_1 and S_2 unless it is removed even before S_1, for the following reasons. For an arc leaving r_2 on the right side, this follows from Lemma 3.2, and for an arc leaving r'_1 on the left side, it follows from Lemma 3.5. Therefore, all of those arcs are removed before S_2. Hence the cardinality of an optimum solution for G_{i-1} is zero, whereas it is positive for G_0 since G is connected. This contradicts the induction hypothesis, namely that Invariant 3.1(2) is valid before $\overline{a}_i = (v, w)$ is removed. □

Now we make a case distinction. First, we consider the case where r'_2 is trivial, that is, r'_2 consists solely of u or, equivalently, $u = v$. We will show that in this case, there is no cycle-free (s, t)-path containing $\overline{a}_i$ in G_{i-1} (Lemma 3.10). After that, we will show that otherwise no (s, t)-path in G_{i-1} contains $\overline{a}_i$ at all (Lemma 3.11).

LEMMA 3.10. *If r'_2 is trivial, no cycle-free (s, t)-path in G_{i-1} contains $\overline{a}_i$ (see Fig.* 12, *part* (1)).

Proof. Lemma 3.2 implies that at stage S_1, there is no longer an arc leaving C and pointing to its exterior except possibly arcs leaving v because all other arcs leave r'_1 on the right side. Therefore, any (s, t)-path that contains $\overline{a}_i = (v, w)$ must finally return to v in order to reach t. Consequently, such a path is *not* cycle-free. □

LEMMA 3.11. *If r'_2 is* not *trivial, no (s, t)-path in G_{i-1} contains $\overline{a}_i$ (see Fig.* 12, *part* (2)).

Proof. Let a and a' denote the in-going arcs of w with respect to r_0 and r_1, respectively. We make a case distinction. We show that the claim is true in the second case and that the other two cases do not occur.

Case 1. The global time counter does not change between stages S_0 and S_1.

Since (v, w) is unusual, the global time counter does not change between stages S_1 and S_2 either. Therefore, the global time counter does not change at all between stages S_0 and S_2. This means that vertex v and the last inner vertex of r_2, say v',

have equal time stamps when the last arc, that is, (v', v), is added to r_2. In this case, however, arc (v', v) would immediately be removed and *not* added to r_2. This contradicts the assumed situation at stage S_2.

Case 2. $a \neq a'$.

In particular, the in-going arc of w has changed at least once before stage S_1, where w is released. Hence the second claim of Lemma 3.5 implies that there is no longer an arc leaving w at stage S_1, which means that no (s,t)–path contains (v,w) at all.

Case 3. $a = a'$ and the global time counter does change between stages S_0 and S_1.

Since a is occupied at S_0 and at S_1, a is occupied all the time between S_0 and S_1. In particular, a is occupied at, say, any intermediate stage S where the global time counter changes.

First, we show that at this stage S, a belongs to some (s,t)-path p in the current graph which solely consists of occupied arcs. To this end, we make a case distinction. If the global time counter changes at S because a new search phase starts, the current solution at S decomposes into (s,t)-paths. In particular, a belongs to such a path. On the other hand, if the global time counter changes at S because of a conflict, it must be a conflict of the third type because by the induction hypothesis, the fourth type has not occurred so far. In this case, the current solution consists of some vertex-disjoint (s,t)-paths and the search path. We assume that a belongs to the search path because otherwise the same argument as in the former case applies. Therefore, let x denote the vertex where the conflict occurs. Then we define p to be the concatenation of the search path with the subpath from x to t of the (s,t)-path involved in the conflict. Obviously, p is an (s,t)-path, contains a, and consists solely of occupied arcs.

In any case, the path p must leave C somewhere so as to enter the exterior of C and to reach t. Let y be the first vertex of p after w where this happens, that is, y belongs to C, but the out-going arc a'' of y with respect to p belongs to the exterior of C. By Lemma 3.2, a'' does not leave C on the right side of r_1. Hence a'' leaves C on the left side of the concatenation $r_2 + (v,w)$, and y belongs to r_2' (see Fig. 13).

Since y is the first vertex after w where p enters the exterior of C, p must cross the interior of C from w to y at least once. Let z be the last vertex before y where p leaves the interior of C and enters C. If p runs clockwise (resp., counterclockwise) along C from z to y, the out-going arc of y (resp., z) with respect to $r_2' + (v,w)$ leaves p on the right side. Hence Lemma 3.2 implies that this arc has already been removed before stage S, which contradicts the assumed situation at stage S_2. If p runs neither clockwise nor counterclockwise along C from z to y, we have $y = z$, and of course, we obtain the same contradiction for the out-going arc of $y = z$ with respect to r_2'. □

3.3. Usual arcs. We are now going to prove that for $i = 1, \ldots, m$, there is an optimum solution for G_{i-1} that does not contain $\overline{a}_i$, if, say, $\overline{a}_i = (v,w)$ is a usual arc.

LEMMA 3.12. *If $\overline{a}_i = (v,w)$ is a usual arc, there is an optimum solution for G_{i-1} that does not contain $\overline{a}_i$.*

Proof. Since, (v,w) is usual, the removal of this arc either releases w or resolves a conflict at w (or we have $time\_stamp[v] = time\_stamp[w]$, and either of these situations would otherwise occur). Clearly, if w is released by removing $\overline{a}_i$, we are immediately done since w is a dead end and no (s,t)-path contains $\overline{a}_i$ at all. Thus we focus our attention on the case of a conflict. Let S be an optimum solution for G_{i-1}. We assume that $\overline{a}_i = (v,w)$ is contained in some path p belonging to S because otherwise there is nothing to show. Let p_1 denote the subpath of p from s to w.

Consider the situation immediately before (v,w) is removed. In this situation,

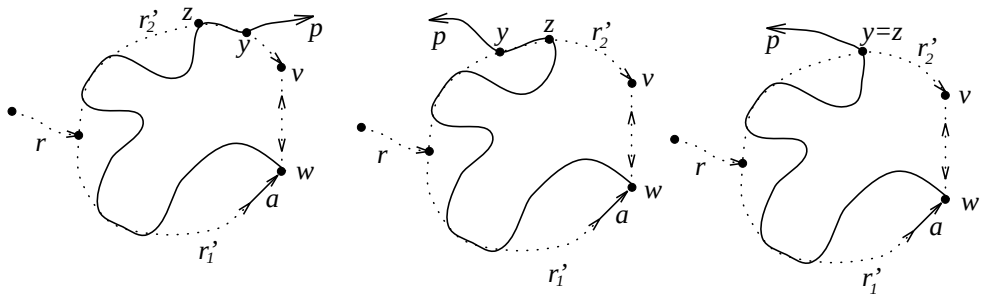


FIG. 13. *The situation in Case* 3 *in the proof of Lemma* 3.11. *Compare this with Fig.* 12*, part* (2). *The three cases of how* z *and* y *may be related are distinguished from each other.*

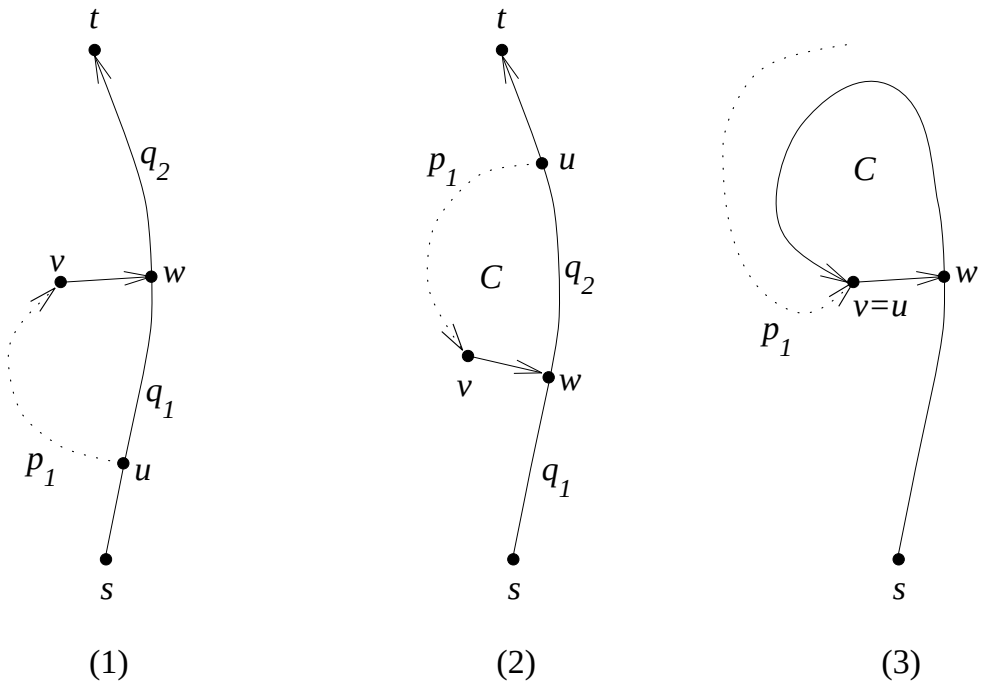


FIG. 14. *The different possible situations in the proof of Lemma* 3.12: (1) u *belongs to* q_1; (2) u *belongs to* q_2; (3) *conflict of the second type.*

there is a unique path q in the current set of paths that contains w as an inner vertex. For example, in a conflict of the second type, q is the search path itself. However, in *any* case, the arc (v, w) points to q from the left side. Let q_1 and q_2 denote the subpath of q from s to w and the remaining subpath, respectively.

There are two different possibilities of how p_1 may be related to q (see Fig. 14).

1. The last vertex of p_1 before w, say u, that is also occupied by q belongs to q_1.
2. The last vertex of p_1 before w, say u, that is also occupied by q belongs to q_2.

Note that all conflicts of the second type belong to the latter class because then $u = v$ is shared by p_1 and q_2. We will now consider both cases separately.

1. *The last vertex of* p_1 *before* w*, say* u*, that is also occupied by* q *belongs to* q_1

(*see Fig.* 14*, part* (1)).

Let $\tilde{q}_1$ denote the subpath of q_1 between u and w. Assume that some path $p' \neq p$ belonging to S shares some vertex with $\tilde{q}_1$. Since p_1 contains (v, w), p_1 separates the *left* side of $\tilde{q}_1$ from t, which implies that p' must leave q_1 somewhere on the *right* side of q_1. This contradicts Lemma 3.2. Therefore, no other path belonging to S hits $\tilde{q}_1$, and we obtain an optimum solution again when we replace the subpath of p_1 from u to w by $\tilde{q}_1$ in S and, since the new path may not be cycle-free, remove all cycles from the new path. The resulting solution does not contain $\overline{a}_i$.

2. *The last vertex of* p_1 *before* w*, say* u*, that is also occupied by* q *belongs to* q_2 (*see Fig.* 14*, parts* (2) *and* (3)).

Let $\tilde{q}_2$ be the subpath of q_2 between w and u. We simply replace the subpath of p_1 between u and w by the reverse of $\tilde{q}_2$ (and remove cycles if necessary). For this we only have to show that all arcs on this reverse path still exist. Note that $\tilde{q}_2$ and the subpath of p_1 from u to w together form a counterclockwise cycle which excludes s. Hence we may apply the following lemma to the *reverse* cycle and obtain that if any arc of the reverse path of $\tilde{q}_2$ is removed, at least one arc of p_1 or $\tilde{q}_2$ is removed as well. Clearly, this contradicts the assumed situation. □

LEMMA 3.13. *Let* $C = (w_0 \to w_1 \to \cdots \to w_l = w_0)$ *be a clockwise cycle of* G_0 *which excludes* s*. If some arc contained in* C *is removed at some stage of the algorithm, at least one of the reverse arcs of* C *has been removed previously.*

Proof. Let, say, arc (w_i, w_{i+1}) be removed at some stage. At this moment, (w_i, w_{i+1}) is the leading arc of the search path. This path must enter C at some vertex. Let w_j be the last of all vertices where the search path enters C from outside. (This happens at least once since C excludes s.) If the path is not continued with (w_j, w_{j-1}), the claim follows from Lemma 3.2 because then (w_j, w_{j-1}) has already been removed. Otherwise, the search path runs along C in the counterclockwise direction and must hence leave C once more at some vertex w_μ so as to enter C again and then run along C in the clockwise direction through (w_i, w_{i+1}). Since w_j was the last vertex where the search path entered C from the outside, it must enter the *interior* of C at w_μ. Then, however, $(w_\mu, w_{\mu-1})$ has already been removed, which again follows from Lemma 3.2. □

Summarizing this section, we obtain the following result.

THEOREM 3.14. *Algorithm* 2.3 *is correct.*

4. Useful insight. As already mentioned in the introduction, this section is devoted to certain properties of the result of our algorithm which have turned out to be very useful in some applications. These properties reflect our idea to always draw paths as far to the right as possible. In fact, we will show that the solutions provided by our algorithm are in some sense "rightmost." For a formal description, however, we first need some terminology.

Let p be a cycle-free (s, t)-path in the undirected input graph G. An *elementary transformation* of p is done as follows. Let F be a face of G such that p contains at least one edge of F and all vertices and edges of F that are contained in p form a connected interval I of the boundary of F. In the elementary transformation, all edges of I are removed from p and all other edges of the boundary of F are inserted in p instead. In other words, if the interior of F is immediately on the right side of p before transforming p in this way, it is on the left side from then on and vice versa. The result is again a cycle-free (s, t)-path; we have simply routed p along F the other way round.

An elementary transformation is called a *homotopic transformation*, if neither s

nor t is incident to F. An elementary transformation is called *right-turning* if t is incident to F and F appears (before the transformation) on the right side of p as p is seen from s to t.

Two (s,t)-paths are *homotopic* if one path can be constructed from the other path by homotopic transformations only. An (s,t)-path p is said to be *more right than* another (s,t)-path q if p can be constructed from q using a sequence of homotopic and right-turning transformations with at least one right-turning transformation. Clearly, "homotopic" defines an equivalence relation on all (s,t)-paths and "more right" defines a strict partial ordering. An (s,t)-path p is called *rightmost* if there is no path more to the right than p, that is, p is a minimal element in that partial ordering.

An undirected input instance (G,s,t) is called *complete* if the cardinality of an optimum solution equals the number of arcs incident to s. In other words, the vertices adjacent to s form a minimum (s,t)-separator and all arcs leaving s are occupied.

Two maximum solutions $(p_1,\dots,p_k)$ and $(p'_1,\dots,p'_k)$ for a complete instance are called *homotopic* if p_i is homotopic to p'_i for $i=1,\dots,k$. Finally, a maximum solution $(p_1,\dots,p_k)$ for a complete instance is *rightmost* if for $i=1,\dots,k$, there is no solution $(p'_1,\dots,p'_k)$ such that p'_i is more to the right than p_i. Note that the rightmost solution is in general not unique (if it exists at all); any solution homotopic to a rightmost solution is rightmost as well. Now we are able to state our result.

THEOREM 4.1. *The solution determined by Algorithm* 2.3 *for a* complete *instance is a rightmost solution.* (*In particular, a rightmost solution is actually guaranteed to exist.*)

Proof. We first define elementary, homotopic, and right-turning transformations for *directed* graphs in a similar manner. Therefore, let $\vec{G}=(\vec{V},\vec{A})$ be a directed graph and $s,\ t\in\vec{V}$, $s\neq t$. For any pair (v,w) and (w,v) of corresponding arcs in $\vec{G}$, we assume for convenience that there is an additional face separating (v,w) from (w,v). Let p be a cycle-free (s,t)-path in $\vec{G}$ that possibly contains not only forward arcs but also backward arcs. Then an elementary transformation is again done as described above, that is, by routing p along an incident face F the other way around. This yields another (s,t)-path, which may again contain forward *and* backward arcs. An (s,t)-path p containing only forward arcs is called *rightmost directed* if there is no other path that contains only forward arcs and is more to the right than p.

Now it is obvious that a rightmost path in G immediately induces a rightmost directed path in G_0 and vice versa. Also, by Lemma 3.2, any of the paths $p_1,\dots,p_k$ is rightmost directed in the final graph G_m. In other words, the resulting solution is rightmost in G_m. Therefore, it suffices to show that for any cycle-free solution S for G_{i-1}, there is a homotopic solution that does not contain $\overline{a}_i$. This implies that at least one rightmost solution of G_0 will "survive" throughout Algorithm 2.3.

Consider the situation where arc $\overline{a}_i$ is removed. Let $p\in S$ be a path in G_{i-1} containing $\overline{a}_i$. First, note that $\overline{a}_i$ cannot be unusual because otherwise p cannot be cycle-free by Lemmas 3.10 and 3.11. On the other hand, if $\overline{a}_i$ is usual, we may construct from p a path that is feasible in G_i by applying the construction used in the proof of Lemma 3.12 (see Fig. 14). Now it is easy to see that in each of the distinct cases in that proof, the resulting solution is homotopic to S. □

REFERENCES

[1] P. ELIAS, A. FEINSTEIN, AND C. SHANNON, *Note on maximum flow through a network*, IRE Trans. Inform. Theory, IT-2 (1956), pp. 117–119.

[2] S. EVEN, *Graph Algorithms*, Pitman, Boston, 1979.

[3] S. Even and R. Tarjan, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.
[4] L. Ford and D. Fulkerson, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.
[5] L. Ford and D. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.
[6] R. Hassin and D. Johnson, *An $\mathcal{O}(n \log^2 n)$ algorithm for maximum flows in undirected planar networks*, SIAM J. Comput., 14 (1985), pp. 612–624.
[7] A. Itai and Y. Shiloach, *Maximum flows in planar networks*, SIAM J. Comput., 8 (1979), pp. 135–150.
[8] S. Khuller and J. Naor, *Flow in planar graphs with vertex capacities*, Algorithmica, 11 (1994), pp. 200–225.
[9] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley, New York, 1990.
[10] V. Malhotra, M. Kumar, and S. Maheshwari, *An $\mathcal{O}(|V^3|)$ algorithm for finding maximum flows in networks*, Inform. Process. Lett., 7 (1978), pp. 277–278.
[11] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, Englewood Cliffs, NJ, 1982.
[12] B. Reed, N. Robertson, A. Schrijver, and P. Seymour, *Finding disjoint trees in graphs on surfaces*, in Graph Structure Theory: Proc. AMS–IMS–SIAM Joint Summer Research Conference on Graph Minors, AMS, Providence, RI, 1993, pp. 295–301.
[13] J. Reif, *Minimum s-t-cut of a planar undirected network in $\mathcal{O}(n \log^2 n)$ time*, SIAM J. Comput., 12 (1981), pp. 71–81.
[14] H. Ripphausen-Lipa, D. Wagner, and K. Weihe, *Linear time algorithm for disjoint two–face paths problems in planar graphs*, in Algorithms and Computation, 4th International Symposium (ISAAC '93), K. Ng, P. Raghavan, N. Balasubramanian, and F. Chin, eds., Lecture Notes in Comput. Sci. 762, Springer-Verlag, Berlin, 1993, pp. 343–352.
[15] H. Ripphausen-Lipa, D. Wagner, and K. Weihe, *Efficient algorithms for disjoint paths in planar graphs*, in DIMACS Series in Discrete Mathematics and Computer Science, Vol. 20, W. Cook, L. Lovász, and P. Seymour, eds., Springer-Verlag, Berlin, pp. 295–354.
[16] H. Suzuki, T. Akama, and T. Nishizeki, *Finding Steiner forests in planar graphs*, in Proc. 1st ACM–SIAM Symposium on Discrete Algorithms (SODA '90), SIAM, Philadelphia, 1990, pp. 444–453.
[17] K. Weihe, *Edge-disjoint (s,t)-paths in undirected planar graphs in linear time*, in Proc. 2nd European Symposium on Algorithms (ESA '94), J. Leeuwen, ed., Lecture Notes in Comput. Sci. 855, Springer-Verlag, Berlin, 1994, pp. 130–140.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 350–368, April 1997

003

RANDOMIZED DISTRIBUTED EDGE COLORING VIA AN EXTENSION OF THE CHERNOFF–HOEFFDING BOUNDS*

ALESSANDRO PANCONESI† AND ARAVIND SRINIVASAN‡

Abstract. Certain types of routing, scheduling, and resource-allocation problems in a distributed setting can be modeled as edge-coloring problems. We present fast and simple randomized algorithms for edge coloring a graph in the synchronous distributed point-to-point model of computation. Our algorithms compute an edge coloring of a graph G with n nodes and maximum degree Δ with at most $1.6\Delta + O(\log^{1+\delta} n)$ colors with high probability (arbitrarily close to 1) for any fixed $\delta > 0$; they run in polylogarithmic time. The upper bound on the number of colors improves upon the $(2\Delta - 1)$-coloring achievable by a simple reduction to vertex coloring.

To analyze the performance of our algorithms, we introduce new techniques for proving upper bounds on the tail probabilities of certain random variables. The *Chernoff–Hoeffding bounds* are fundamental tools that are used very frequently in estimating tail probabilities. However, they assume stochastic independence among certain random variables, which may not always hold. Our results extend the Chernoff–Hoeffding bounds to certain types of random variables which are not stochastically independent. We believe that these results are of independent interest and merit further study.

Key words. edge coloring, distributed algorithms, parallel algorithms, probabilistic algorithms, Chernoff–Hoeffding bounds, stochastic dependence, λ-correlation, correlation inequalities, large deviations

AMS subject classifications. 05C85, 60C05, 60F10, 60G50, 68Q22, 68Q25, 68R10

PII. S0097539793250767

1. Introduction.

An important limitation for a distributed network without global memory is *locality of computation*: since sending messages to faraway nodes is expensive, communication should only take place between nearby nodes. Models of parallel computation like the PRAM abstract this problem of locality away by assuming the existence of a global shared memory with fast concurrent access. We are interested in studying how fast individual processors can compute their portion of the output in a message-passing distributed system with such "local" information alone. The model we consider is the synchronous distributed point-to-point model, in which the processors are arranged as the vertices of an n-vertex graph $G = (V, E)$ and where all communication is via the edges of G alone. In this model, we study the edge-coloring problem, a basic combinatorial problem with many applications to distributed computing. Edge colorings can be used to model certain types of jobshop-scheduling, packet-routing, and resource-allocation problems in a distributed setting. For example, the problem of scheduling I/O operations in a certain parallel architecture can be modeled as follows (see Jain et al. [9]). We are given a set of processes $\mathcal{P}$ and a set of resources $\mathcal{R}$ such that each process $p \in \mathcal{P}$ needs a subset

* Received by the editors June 23, 1993; accepted for publication (in revised form) May 10, 1995. A preliminary version of this work appears as "Fast randomized algorithms for distributed edge coloring" in *Proc. ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1992, pp. 251–262. This research was conducted while the authors were with the Department of Computer Science, Cornell University, Ithaca, NY 14853 and was supported in part by NSF PYI award CCR-89-96272 with matching support from UPS and Sun Microsystems.

http://www.siam.org/journals/sicomp/26-2/25076.html

† Informatik, Freie Universität Berlin, Takustrasse 9, 14195 Berlin, Germany (ale@inf.fu-berlin.de). The research of this author was supported by an Alexander von Humboldt research fellowship.

‡ Department of Information Systems and Computer Science, National University of Singapore, Singapore 119260, Republic of Singapore (aravind@iscs.nus.sg).

$f(p) \subseteq \mathcal{R}$ of the resources where (i) each process p needs every resource in $f(p)$ for a unit of time each and (ii) p can use the resources in $f(p)$ in any order. From this we can construct a bipartite graph $G_{\mathcal{P},\mathcal{R}} = (\mathcal{P}, \mathcal{R}, E_{\mathcal{P},\mathcal{R}})$, where $E_{\mathcal{P},\mathcal{R}} = \{(p,r) |\ p \in \mathcal{P} \wedge r \in f(p)\}$. An edge coloring of $G_{\mathcal{P},\mathcal{R}}$ with c colors yields a schedule for the processes to use the resources within c time units. Optimal colorings correspond to optimal schedules.

Edge coloring can also be used in distributed models in situations where broadcasts are infeasible or undesirable: an edge coloring of the network results in a schedule for each processor to communicate with at most one neighbor at every step; at time step i, processors communicate via the edges colored i only. Using a "small" number of colors reduces the wastage of time in this schedule.

1.1. Related work. Note that Δ colors are necessary to edge color a graph with maximum degree Δ. Vizing showed that it is always possible to edge color a graph with $\Delta + 1$ colors and gave a polynomial-time algorithm to compute such a coloring [22] (see, for instance, Bollobás [3]). Efforts to parallelize Vizing's theorem have failed so far. The best known algorithm is an RNC algorithm of Karloff and Shmoys using $\Delta + O(\Delta^{1/2+\epsilon})$ colors for any fixed $\epsilon > 0$; this algorithm has been derandomized in NC (see Berger and Rompel [2] and Motwani, Naor, and Naor [15]). In the distributed model, the best edge-coloring algorithm known prior to this work was to apply a vertex-coloring algorithm to the line graph $L(G)$ of G. There are fast (polylogarithmic) randomized vertex-coloring algorithms that use $(\Delta+1)$ and Δ colors, which translate to $(2\Delta-1)$- and $(2\Delta-2)$-edge-coloring algorithms, respectively (see Luby [13] and Panconesi and Srinivasan [17]). In the deterministic case, there are no known $(2\Delta-1)$-edge-coloring algorithms of polylogarithmic running time; the best running time is $2^{O(\sqrt{\log n})}$, which is asymptotically better than any fixed root of n but which grows faster than any polylogarithmic function of n [17]. Interestingly, distributed Δ-edge coloring for bipartite graphs requires $\Omega(\mathit{diameter}(G))$ time even with randomization [17], whereas this can be done in $O(\log n)$ time deterministically in the PRAM model [11].

1.2. Our contributions. In this paper, we present fast and simple randomized algorithms to edge color G with at most $1.6\Delta + O(\log^{1+\delta} n)$ colors with high probability for any fixed $\delta > 0$, where Δ is the maximum degree of the vertices of G. At the heart of our analysis is an extension of the *Chernoff–Hoeffding bounds*, which are key tools in bounding the tail probabilities of the sums of independent random variables (see Chernoff [4], Hoeffding [8], and Raghavan [18]).

Our edge-coloring algorithm is based on a very simple randomized algorithm to color bipartite graphs, which can be explained in a few lines. Given a bipartite graph $G = (A, B, E)$ with maximum degree Δ, each vertex in B picks distinct colors from $\{1, 2, \ldots, \Delta\}$ at random for its edges without replacement, i.e., edges incident to the same vertex in B get different colors. Then each vertex $v \in A$ checks for each color α if more than one of its incident edges has color α and, if so, chooses one of them at random as the *winner*, and all the other edges of color α which are incident to v are decolored. The key claim is that for every vertex, the number of decolored edges incident to it is at most $\Delta(1+\epsilon)/e$ with high probability for any fixed $\epsilon > 0$, where e is the base of natural logarithms. Assuming that this holds, we can repeat the above iteration with a set of $\Delta(1+\epsilon)/e$ *fresh* colors, and so on. In spite of its simplicity, the algorithm requires an interesting probabilistic analysis; this is based upon an extension of the Chernoff–Hoeffding bounds to a certain case of dependence among the random variables, which we call λ-correlation. We believe that these results have

the potential for further applications and merit further study.

A preliminary version of this work appeared in [16], where we showed how to edge color using at most $1.6\Delta + O(\log^{2+\delta} n)$ colors. By presenting a tighter analysis of the tail probabilities, we improve this to $1.6\Delta + O(\log^{1+\delta} n)$ colors here.

In section 2, we define the basic notation used, and in section 3, we describe our main analytical tool—the extension of the Chernoff–Hoeffding bounds. Section 4 presents our algorithm, whose performance is analyzed in section 5. Some extensions and applications of this work are described in section 6.

2. Notation. A *message-passing distributed network* is an undirected graph $G = (V, E)$ where vertices, or nodes, correspond to processors and edges correspond to bidirectional communication links. Each node has its unique ID. There is no shared memory and processors can communicate only by sending messages through the network. The network is *synchronous*, i.e., computation takes places in a sequence of *rounds*; in each round, each node does any amount of local computation, sends messages to its neighbors in the graph, and reads messages sent to it by its neighbors. The time complexity of a distributed algorithm, or *protocol*, is given by the number of rounds needed to compute a given function.

Though each node has no knowledge about the topology of the entire network, it knows upper bounds n and Δ on the total number of nodes and maximum degree of the network, respectively. We also sketch an alternative algorithm if Δ and n are unknown, but the constant factor in the $O(\log^{1+\delta} n)$ term in the number of colors used is higher in this case.

Notice that in this model the cost of sending a message from one vertex to another is proportional to the length of a shortest path between the two vertices. Hence if we want a protocol to run for t rounds, then each vertex can communicate only with vertices at distance at most t from it. This is not so in the PRAM model, where the shared memory allows any two processors to communicate in one unit of time. Lower bounds for distributed computation imposed by this locality have been presented by Linial [12]. Also, as mentioned before, distributed Δ-edge coloring for bipartite graphs requires $\Omega(\mathit{diameter}(G))$ time, even with randomization [17]. In particular, we cannot two-color the *vertices* of a bipartite graph G distributively in $o(\mathit{diameter}(G))$ time.

Given an undirected graph $G = (V, E)$, we denote by Δ its maximum degree, i.e., the maximum number of edges incident with any node; by d_u we denote the degree of vertex u, by $N(u)$ we denote the set of neighbors of u, and by $\delta(u)$ we denote the set of edges incident with u.

Given a positive integer n, $[n]$ denotes the set $\{1, 2, \ldots, n\}$. The permanent of a (possibly nonsquare) matrix M with c columns and r rows, where $c \le r$, is defined as the natural extension of the permanent of square matrices. Let $\mathcal{P} = \{\pi \mid \pi : [c] \to [r],\ \pi \text{ is one–to–one}\}$. Then

$$\mathit{perm}(M) \doteq \sum_{\pi \in \mathcal{P}} \prod_{i=1}^{n} M_{\pi(i),i}.$$

An event $\mathcal{A}$ is said to happen *with high probability* (w.h.p.) if $\Pr(\mathcal{A}) \ge 1 - 1/f(n)$ for some superpolynomial function $f(n)$ (i.e., $n^c = o(f(n))$ for all fixed $c > 0$).

In our algorithms, we will use Luby's vertex-coloring algorithm [13] as a subroutine. When applied to the line graph of G, the algorithm computes a $(2\Delta(G)-1)$-edge coloring of G, with its running time being $O(\log n)$ w.h.p. The algorithm only needs local information—a vertex only needs to know its own degree. Another property

of the algorithm that we will use is that vertices can be added dynamically to the graph, each vertex u with its own palette of $\deg(u)+1$ colors, and the algorithm still works as claimed (the running time is $O(\log n)$ from the time of the last insertion). Other algorithms could be used as well, but we refer to this algorithm of Luby for conciseness.

3. The Chernoff–Hoeffding bounds extension. In this section, we introduce our extension of the Chernoff–Hoeffding bounds, which are important tools used in estimating the tail probabilities of random variables. Given n *independent* random variables $X_1, X_2, \ldots, X_n$, these bounds are used in deriving an upper bound on the upper tail probability $\Pr(X \geq (1+\epsilon)\mu)$, where $X = \sum_{i=1}^{n} X_i$, $\mu = E[X]$, and $\epsilon > 0$. We extend these bounds to a certain case of dependency among the X_i's, which we call λ-correlation.

Let us review Chernoff's approach to upper bound the upper tail probability of a random variable X when X is the sum of *independent* binary random variables $X_1, X_2, \ldots, X_n$ [4]. (This idea is apparently originally due to Bernstein [8].) The idea is to use Markov's inequality on the random variable e^{tX} for an arbitrary $t > 0$ and minimize with respect to t, that is, to use the fact that

$$\begin{aligned}\Pr(X > (1+\epsilon)\mu) &= \Pr(e^{tX} > e^{t(1+\epsilon)\mu}) \\ &\leq \frac{E[e^{tX}]}{e^{t(1+\epsilon)\mu}}\end{aligned}$$

and minimize the last ratio for $t > 0$. This is achieved by finding a good upper bound for the numerator $E[e^{tX}]$ by using the fact that X is the sum of *independent* random variables. It is standard (see, e.g., Raghavan [18]) to use this to show that in this case, if $X_i \in \{0,1\}$ for each i, then

$$\min_{t>0} \frac{E[e^{tX}]}{e^{t(1+\epsilon)\mu}} \leq F(\mu,\epsilon) \doteq \left[\frac{e^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right]^{\mu}. \tag{1}$$

Hoeffding [8] considered a more general case where X is the sum of n independent and *bounded* random variables $X_i \in [a_i, b_i]$, and he used the above approach to show that if $E[X] = \mu$, then for $\epsilon > 0$,

$$\min_{t>0} \frac{E[e^{tX}]}{e^{t(1+\epsilon)\mu}} \leq G(\mu,\epsilon,\vec{a},\vec{b}) \doteq \exp\left[-\frac{2\,\mu^2\epsilon^2}{\sum_{i\in[n]}(b_i-a_i)^2}\right]. \tag{2}$$

The bounds (1) and (2) will be used in our proofs. Henceforth, we refer to these bounds of Chernoff and Hoeffding as the CH bounds. If ϵ is a fixed positive quantity no greater than 1 (which will be true in all of our applications), then $F(\mu,\epsilon) \leq e^{-\epsilon^2\mu/3}$. Hence if $\mu = \Omega(\log^{1+\delta} n)$, then $F(\mu,\epsilon)$ is the inverse of a superpolynomial function of n, for any fixed $\delta > 0$. (Similar considerations apply to $G(\mu,\epsilon,\vec{a},\vec{b})$.) This fact makes the CH bounds a powerful tool for deriving strong performance guarantees for randomized algorithms and will be used repeatedly in this paper.

3.1. The general case. We now introduce λ-correlation and prove the general extension of the CH bounds. In section 3.2, we will then discuss an important special case of the results of this section.

Our proof is based on the observation that if we can upper bound each term $E[X^k]$ of the Maclaurin expansion of $E[e^{tX}]$ by $\lambda\, E[\hat{X}^k]$, where $\hat{X}$ is the sum of independent

random variables, and if $E[e^{t\hat{X}}] \leq B$, then $E[e^{tX}] \leq \lambda B$. We start with the following definition.

DEFINITION 3.1. *Let $X_1, X_2, \ldots, X_n$ be bounded random variables such that $X_i \in [a_i, b_i]$ and let $X = \sum_{i\in[n]} X_i$. The X_i's are* λ-correlated *if there exists a collection of independent twin random variables $\hat{X}_i \in [a_i, b_i]$ such that*

(i) *$E[X] \leq E[\hat{X}]$, where $\hat{X} = \sum_{i\in[n]} \hat{X}_i$, and*

(ii) *for all $I \subseteq [n]$ and positive integers s_i, $i \in I$,*

$$E\left[\prod_{i\in I} X_i^{s_i}\right] \leq \lambda \prod_{i\in I} E[\hat{X}_i^{s_i}].$$

Our main theorem is now the following.

THEOREM 3.2. *Let X be the sum of λ-correlated random variables $X_1, X_2, \ldots, X_n$, where $X_i \in [a_i, b_i]$, and let $\hat{X}$ be the sum of the n twin variables $\hat{X}_i$. Then*

$$\Pr(X > (1+\epsilon)\, E[\hat{X}]) \leq \lambda G(E[\hat{X}], \epsilon, \vec{a}, \vec{b}).$$

Proof. Let $\mu = E[\hat{X}]$. As in the classical proof, we start by introducing a positive parameter t and by applying Markov's inequality to the variable e^{tX}:

$$\begin{aligned}\Pr(X > (1+\epsilon)\mu) &= \Pr(e^{tX} > e^{t(1+\epsilon)\mu}) \\ &\leq \frac{E[e^{tX}]}{e^{t(1+\epsilon)\mu}}.\end{aligned}$$

By the hypotheses of the boundedness of X, we may apply linearity of expectation to an infinite series:

$$E[e^{tX}] = E\left[\sum_{k=0}^{\infty} \frac{t^k X^k}{k!}\right] = \sum_{k=0}^{\infty} \frac{t^k E[X^k]}{k!}.$$

Now $X^k = (\sum_{i=1}^n X_i)^k$ is a sum of terms of the form $\prod_{i\in I} X_i^{s_i}$ for some $I \subseteq [n]$ and positive integers s_i. Hence by linearity of expectation and the assumption of λ-correlation,

$$E[X^k] \leq \lambda E[\hat{X}^k].$$

Thus

$$E[e^{tX}] \leq \lambda \sum_{k=0}^{\infty} \frac{t^k E[\hat{X}^k]}{k!} = \lambda E[e^{t\hat{X}}].$$

By the already discussed result of Hoeffding [8], when $\hat{X}$ is the sum of n independent bounded random variables $\hat{X}_i \in [a_i, b_i]$,

$$\min_{t>0} \frac{E[e^{t\hat{X}}]}{e^{t(1+\epsilon)\mu}} \leq G(\mu, \epsilon, \vec{a}, \vec{b}). \qquad \square$$

In this paper, we will use the special case of Theorem 3.2 where $X_i \in [0, 1]$, $i \in [n]$. In this case, $F(\mu, \epsilon)$ is also an upper bound for the upper tail of X.

COROLLARY 3.3. *Let X be the sum of n λ-correlated random variables $X_i \in [0, 1]$. Then*

$$\Pr(X > (1+\epsilon)E[\hat{X}]) \leq \lambda F(\mu, \epsilon).$$

Proof. Let $E[\hat{X}] = \mu$. When $\hat{X}$ is the sum of n independent random variables $\hat{X}_i \in [0,1]$, Hoeffding (cf. Theorem 1 of [8]) shows that if $t \in (0, 1-\mu/n)$ and $\epsilon = nt/\mu$, then

$$\Pr\left(\frac{\hat{X}-\mu}{n} \geq t\right) \leq \min_{s>0} \frac{E[e^{s\hat{X}}]}{e^{s(1+\epsilon)\mu}} \leq \left(\frac{\mu}{\mu+nt}\right)^{\mu+nt} \left(1+\frac{n\,t}{n-\mu-nt}\right)^{n-\mu-nt}.$$

By the proof of Theorem 3.2, we see that $E[e^{sX}] \leq \lambda E[e^{s\hat{X}}]$ for any $s > 0$. Thus by applying the standard approximation $1 + x \leq e^x$ for $x = nt/(n-\mu-nt)$, we get

$$\Pr(X \geq (1+\epsilon)\mu) \leq \lambda \left(\frac{e^\epsilon}{(1+\epsilon)^{(1+\epsilon)}}\right)^\mu = \lambda F(\mu, \epsilon). \qquad \square$$

3.2. Binary random variables. An important special case of Definition 3.1 is when $X_i \in \{0,1\}$. In this case, the condition on the expectations simplifies considerably to become

$$\Pr\left(\bigwedge_{i\in I} X_i = 1\right) \leq \lambda \prod_{i\in I} \Pr(\hat{X}_i = 1)$$

for all $I \subseteq [n]$.[1] This special case is interesting in its own right and hence we record it as the following theorem.

THEOREM 3.4. *Let $X_1, X_2, \ldots, X_n$ be given* 0–1 *random variables with $X = \sum_i X_i$. If there exist* independent *random variables $\hat{X}_1, \hat{X}_2, \ldots, \hat{X}_n$ with $\hat{X} = \sum_i \hat{X}_i$ and $E[X] \leq E[\hat{X}]$ such that for all $I \subseteq [n]$,*

$$\Pr\left(\bigwedge_{i\in I} X_i = 1\right) \leq \lambda \prod_{i\in I} \Pr(\hat{X}_i = 1),$$

then

$$\Pr(X > (1+\epsilon)E[\hat{X}]) \leq \lambda F(E[\hat{X}], \epsilon).$$

The statement follows immediately from Corollary 3.3. Notice that λ-correlation follows if the X_i's are "negatively correlated" in the following sense:

$$\Pr\left(\bigwedge_{i\in I} X_i = 1\right) \leq \prod_{i\in I} \Pr(X_i = 1)$$

for all $I \subseteq [n]$. We now present an example where precisely this kind of situation arises and which will also be used later in this paper.

Suppose we have n balls that are thrown uniformly and independently at random into n bins, and we want to estimate the number B of empty (missed) bins. Let B_i be an indicator random variable that is 1 if bin i is empty and 0 otherwise. For any $i \in [n]$,

$$\Pr(B_i = 1) = \left(1 - \frac{1}{n}\right)^n \leq \frac{1}{e}.$$

[1] This was defined as "self-weakening with parameter λ" in [16].

It follows that $E[B] = E[\sum_i B_i] \leq n/e$. The B_i's are 1-correlated. To see this, consider a subset $J \subseteq [n]$ and any $i \in [n] - J$. Then

$$\begin{aligned}\Pr\left(B_i = 1 \mid \bigwedge_{j\in J} B_j = 1\right) &= \left(1 - \frac{1}{n - |J|}\right)^n \\ &\leq \left(1 - \frac{1}{n}\right)^n \\ &= \Pr(B_i = 1).\end{aligned}$$

By straightforward induction, this implies that for all $I \subseteq [n]$,

$$\Pr\left(\bigwedge_{i\in I} B_i = 1\right) \leq \prod_{i\in I} \Pr(B_i = 1).$$

Thus the B_i's are 1-correlated, where we may take the twin variables $\hat{B}_i$ to be i.i.d. 0–1 random variables with $\Pr(B_i = 1) = 1/e$ for each i. Hence not only is $E[B] \leq n/e$, but by Theorem 3.4,

$$Pr\left(B > \frac{(1+\epsilon)n}{e}\right) \leq F\left(\frac{n}{e}, \epsilon\right).$$

Remarks. The above fact can also be given a completely different (and simple) proof via the theory of martingales using Azuma's inequality (see, for example, Alon, Spencer, and Erdős [1] or McDiarmid [14].) We have presented this proof to illustrate our techniques. Also, Jain has proved the following lemma [19].

Let $a_1, a_2, \ldots, a_n$ be n random trials (not necessarily independent) such that the probability that trial a_i "succeeds" is bounded above by p regardless of the outcomes of the other trials. Then if X is the random variable that represents the number of "successes" in these n trials and Y is a binomial variable with parameters (n, p), then $\Pr[X \geq k] \leq \Pr[Y \geq k]$, $0 \leq k \leq n$.

The assumptions of Jain's lemma are strictly stronger than those of 1-correlation. For instance, in the balls and bins example,

$$\Pr\left(B_n = 1 | \bigwedge_{i\in[n-1]} B_1 = 0\right) = \frac{n-1}{n+1},$$

which for $n \geq 3$ is greater than $\Pr(B_n = 1)$ ($\approx 1/e$). Note, however, that our result does not subsume Jain's lemma since his result upper bounds $\Pr(X \geq k)$ by the true binomial upper bound, while we only upper bound it by the CH bound.

4. The edge-coloring algorithm. We now present our randomized distributed edge-coloring algorithm. The algorithm uses an idea of Karloff and Shmoys to reduce the problem of edge coloring general graphs to that of edge coloring a special class of bipartite graphs [10]. The Karloff–Shmoys scheme uses the fact that bipartite graphs can be edge colored optimally in the PRAM model of computation, which is provably impossible in our distributed model [17]. Instead, we use a distributed subroutine that computes a "good" coloring. Also different is the handling of the "leftover" graphs at the end of the recursion, which we color by making use of Luby's vertex-coloring algorithm.

The input to the algorithm is a distributed network $G = (V, E)$ and some fixed $\epsilon, \delta > 0$. In addition, each node knows upper bounds n and Δ on $|V|$ and the maximum degree $\Delta(G)$ of G, respectively. This information is not necessary but yields better multiplicative constants. The case where Δ and n are unknown is sketched towards the end of section 5.1.

The algorithm is recursive and computes an edge coloring of G using at most $1.6\Delta + O(\log^{1+\delta} n)$ colors and runs in $O(\log n)$ time; both of these bounds hold w.h.p. Let THRESHOLD $= \log^{1+\delta} n$, and $new(\Delta) = \Delta/2 + \sqrt{\Delta \log^{1+\delta/2} n}$; the algorithm is as follows.

If $\Delta \leq 16$THRESHOLD, **then** edge color G with $2\Delta - 1$ colors using Luby's algorithm and exit;[2] **else** execute the following:

1. Compute a random partition of $V(G)$ into black and white vertices. (All vertices flip a fair coin independently and in parallel.) Let $G[B]$ be the subgraph induced by the black vertices, $G[W]$ be the subgraph induced by the white vertices, and $G[B, W]$ be the bipartite subgraph formed by the edges having endpoints of different colors.

2. Edge color $G[B, W]$ using our bipartite edge-coloring algorithm described below with the parameters $new(\Delta)$, ϵ, and δ.

3. Set $\Delta := new(\Delta)$ and recurse on $G[B]$ and $G[W]$ using the *same* set of fresh new colors on both graphs with the same parameters ϵ and δ as before. (*Remark.* Though the bipartite algorithm modifies its first parameter $new(\Delta)$ in the course of its execution, we assume that it is passed "by value," i.e., that the value of $new(\Delta)$ referred to here and in step 2 above is the same.)

Remark. $new(\Delta)$ is meant to be an upper bound on $G[B]$, $G[W]$, and $G[B, W]$. It is easily seen via the standard CH bounds that it is indeed so w.h.p. if $\Delta \geq$ THRESHOLD and hence if $\Delta \geq 16$THRESHOLD [10].

We now present our main algorithm—a distributed algorithm to color the bipartite graphs produced above.

4.1. Distributed edge coloring of bipartite graphs. Given a bipartite graph $G = (A, B, E)$, we assume that each vertex knows whether it belongs to A or B. This is an important assumption because such information cannot be computed fast distributively as mentioned in section 2, but it is verified for the bipartite graphs generated by the Karloff–Shmoys scheme. Henceforth, we will refer to vertices in A as the *top* vertices and to the vertices in B as the *bottom* vertices.

Given parameters Δ_C, ϵ, and δ such that Δ_C is an upper bound on the degree of G, the algorithm takes $O(\log n)$ time and colors the bipartite graph G with at most $1.6\Delta_C + O(\log^{1+\delta} n)$ colors w.h.p., as long as $\delta > 0$ is any constant (ϵ is used in the algorithm). During any iteration of the algorithm, Δ_C is meant to be an upper bound on the degree of the current graph; we will prove later that *this holds w.h.p. as long as* $\Delta_C \geq \log^{1+\delta} n =$ THRESHOLD. From the remark in section 4, we can assume that this is valid when the bipartite algorithm is called first. As we will briefly discuss in section 5.1, this is not needed but gives better constants. The algorithm is as follows.

1. *Part* I. While $\Delta_C \geq$ THRESHOLD, do the following:

Let G_C be the current graph. Pick a set χ of Δ_C *fresh new* colors.

(i) (random proposal of bottom vertices) In parallel and independently of the other vertices in B, each vertex $v \in B$ assigns a temporary color to each edge in $\delta(v)$

[2] When $\Delta = O(\text{polylog}(n))$, we can compute a $2\Delta - 1$ coloring *deterministically* in $O(\text{polylog}(n))$ time using an algorithm based on the idea of removing maximal matchings. We prefer to use Luby's algorithm here for conciseness.

with uniform probability *without replacement*, i.e., edge e_1 is assigned color $\alpha \in \chi$ with probability $1/\Delta_C$, e_2 is assigned $\beta \in \chi - \{\alpha\}$ with probability $1/(\Delta_C - 1)$, and so on.

(ii) (lottery of top vertices) (*Remark*. The coloring so far is consistent around any vertex $v \in B$ but can be inconsistent around a vertex $u \in A$.) For each $u \in A$, let $C_u(\alpha)$ be the set of edges in $\delta(u)$ with temporary color α. Each vertex $u \in A$ selects a *winner* uniformly at random in $C_u(\alpha)$ for each nonempty $C_u(\alpha)$. All other edges, the *losers*, are decolored and assigned $\perp$.

(iii) Set $\Delta_C := \Delta_C(1+\epsilon)/e$. $G_\perp$, the subgraph of G_C induced by the losers (i.e., by the $\perp$-edges), becomes the new current graph.

2. *Part* II. Let G_r be the remaining graph. Edge color G_r with at most $2\Delta(G_r)-1$ colors by executing Luby's vertex coloring algorithm on the line graph of G_r.

Since we use new colors in each iteration, it is clear that when the algorithm terminates, G has been edge colored legally. It is also apparent that the algorithm works based on local information alone. We now turn to placing bounds on the number of colors used and the running time.

5. Analysis of the algorithm. Since the analysis is fairly involved, we first present a higher-level description of it.

5.1. The basic structure of the analysis. Our key claim will be that in every iteration of Part I of the bipartite edge-coloring algorithm, the maximum degree of the graph shrinks by a factor of at least $(1+\epsilon)/e$ w.h.p., as long as $\Delta_C \geq$ THRESHOLD. That is,

$$\Delta(G_\perp) \leq (1+\epsilon)\frac{\Delta(G_C)}{e}$$

w.h.p. for any fixed $\epsilon > 0$. The condition $\Delta_C \geq$ THRESHOLD ensures that the failure probability given by the extension of the CH bounds is the inverse of a superpolynomial function. Hence w.h.p., no vertex will violate the degree condition. The reason for setting THRESHOLD $= \log^{1+\delta} n$ will be apparent from the probabilistic analysis.

Once the key claim is established, we can bound both the total number of colors used, and the running time of the algorithm. To bound the number of colors used, observe that if the degree of the graph shrinks at every iteration by at least a $(1+\epsilon)/e$ factor w.h.p., then the maximum degree of G_r is at most $\log^{1+\delta} n$ w.h.p.

Hence if $\Delta_C \geq$ THRESHOLD, then w.h.p., the number of colors used by the bipartite algorithm is at most

$$BC(\Delta_C) \leq \Delta_C + \frac{\Delta_C}{e}(1+\epsilon) + \cdots + \frac{\Delta_C}{e^k}(1+\epsilon)^k + 2\log^{1+\delta} n,$$

where k is the smallest integer such that $\Delta_C(1+\epsilon)^k/e^k \leq \log^{1+\delta} n$. Thus for a suitable $\epsilon' > 0$ which depends on ϵ and which can be made arbitrarily small, $BC(\Delta_C)$ is at most

$$\begin{aligned} BC(\Delta_C) &\leq \left(\frac{e}{e-1} + \epsilon'\right)\Delta_C + \left(2 - \frac{e}{e-1} - \epsilon'\right)\log^{1+\delta} n \\ &< 1.585\Delta_C + 0.4\log^{1+\delta} n \\ &< 1.59\Delta_C \end{aligned}$$

when $\Delta_C > 8\ \log^{1+\delta} n$. The running time of the algorithm is $O(\log n)$ because Part I takes $O(\log \Delta_C)$ time and Part II, i.e., Luby's algorithm, takes $O(\log n)$ time.

Note that if $\Delta \geq 16\text{THRESHOLD}$ in the main algorithm, then $\Delta_C > 8 \log^{1+\delta} n$ is true for the bipartite algorithm and hence the above analysis is valid; also note that if $\Delta \leq 16 \log^{1+\delta} n$, then we use Luby's subroutine directly in our main algorithm.

Thus if $\Delta \geq 16\text{THRESHOLD}$ in the main algorithm, then the recurrence for the total number of colors used is

$$\begin{aligned} TC(\Delta) &\leq BC\left(\frac{\Delta}{2} + \sqrt{\Delta \log^{1+\delta/2} n}\right) + TC\left(\frac{\Delta}{2} + \sqrt{\Delta \log^{1+\delta/2} n}\right) \\ &< 1.59\Delta + o(\Delta) \\ &< 1.6\Delta. \end{aligned}$$

If $\Delta \leq 16 \log^{1+\delta} n$, then the main algorithm uses Luby's subroutine directly to get a $2\Delta - 1 \leq 32 \log^{1+\delta} n$ edge coloring. Hence the total number of colors to color *any* graph is at most $1.6\,\Delta + 32 \log^{1+\delta} n$ for any fixed $\delta > 0$ w.h.p.

5.1.1. A truly distributed algorithm. We now sketch the modifications needed to handle the case when both Δ and n are unknown. Each node u initially computes the value $\Delta_u = \max_{v \in N(u)} \deg(v)$. The recursion of the Karloff–Shmoys scheme and the loop of Part I of the bipartite subroutine are then repeated for $c \log \Delta_u$ times for a constant $c > 0$ chosen large enough. A vertex u is said to be *active* as long as no more than $c \log \Delta_u$ rounds have elapsed; it is *inactive* otherwise. An edge incident on an inactive node is inactive. It is convenient to think of Luby's algorithm as run directly by the edges. An inactive and yet uncolored edge f will wait until all of its neighboring edges are either colored or inactive, at which point it starts running Luby's algorithm with a palette of $\deg(f) + 1$ fresh colors, where $\deg(f)$ denotes the number of inactive edges incident upon f. There are two main observations to prove the correctness of and the bounds on the number of colors used by this modified algorithm. First, all of the neighbors of a vertex u will stay active for at least $c \log \deg(u)$ rounds. Hence all vertices such that $\deg(u) = \Omega(\log^{1+\delta} n)$ will be able (w.h.p.) to color enough edges to reduce their degree until it drops to $O(\log^{1+\delta} n)$. Second, as discussed in section 2, Luby's algorithm still works correctly when vertices (in our case, edges) are added dynamically.

The high-probability analysis carries through with these modifications. Simple calculations show that with these modifications, the total number of colors used increases to at most $1.6\Delta + 160 \log^{1+\delta} n$. We omit the calculations for this modified algorithm, which are similar to those presented here for the case where n and Δ are known.

We now return to the case where n and Δ are known, and we turn to the task of proving the key claim. We wish to show that given a graph G and Δ such that $\Delta \geq \Delta(G)$ and $\Delta \geq \text{THRESHOLD}$, then after one iteration of Part I of the bipartite algorithm, the maximum degree of the new graph, $\Delta(G_\perp)$, is at most $(1 + \epsilon)\Delta/e$ w.h.p. for any fixed $\epsilon > 0$. It turns out that the analysis is considerably easier for the top vertices than for the bottom vertices. We begin with the easy part.

5.2. Analysis of the top vertices. Let u be a generic top vertex with incident edges $i = (u, v_i)$. Recall that a *loser* is an edge which, after having gotten a tentative color in the random proposal, lost the lottery and got decolored. Therefore, the new degree of u is given by the number of losers incident with it.

From the point of view of a top vertex, the random proposal and the lottery are equivalent to the following random experiment. For each edge i incident on u, we introduce a ball i, and for each color k, we introduce a bin k; the assignment of a

tentative color to an edge by the algorithm is equivalent to throwing each ball into one of the Δ bins independently and uniformly at random since the bottom vertices assign tentative colors with uniform probability and independently of the other bottom vertices. Recalling that we have at most Δ balls and exactly Δ bins,

$$\begin{aligned}\sharp losers &= \sharp balls - \sharp winners \\ &\leq \sharp bins - \sharp nonempty\ bins \\ &= \sharp empty\ bins.\end{aligned}$$

Let X be a random variable denoting the number of losers. To estimate X and its tail distribution, we will study the random variable $B = \sharp empty\ bins$. For this purpose, we introduce Δ many indicator random variables B_i, where $B_i = 1$ if bin i is empty and 0 otherwise; hence $B = \sum_{i\in[\Delta]} B_i$. Notice that $X \leq B$ always. The variable B was studied in section 3, where it was shown that $E[B] \leq \Delta/e$ and that the B_i's are 1-correlated, which implies that $\Pr(B > (1+\epsilon)\Delta/e) \leq F(\Delta/e, \epsilon)$. Since $E[X] \leq E[B]$ and $\Pr(X > (1+\epsilon)\Delta/e) \leq \Pr(B > (1+\epsilon)\Delta/e)$, we get the following result.

THEOREM 5.1. *Let u be a top vertex and X be the random variable denoting the number of losers incident on it. Then $E[X] \leq \Delta/e$ and*

$$\Pr\left(X > \frac{(1+\epsilon)\Delta}{e}\right) \leq F\left(\frac{\Delta}{e}, \epsilon\right).$$

5.3. Analysis of the bottom vertices. In this section, we analyze what happens to the new degree of a generic bottom vertex v_b. This case is considerably harder to handle than the previous one because of the way in which the random variables describing the process are correlated. For a top vertex, the dependency among the variables was playing for us; given that some edges incident on a top vertex are losers, the probability of having another loser decreases. For a bottom vertex, the situation is reversed: having some edges lose the lottery might even make the probability of having another loser increase. The problem can be seen in the following situation. Let $x_1 = v_b$ and x_2 be bottom vertices, and let y_1 and y_2 be top vertices which induce a four-cycle, i.e., there is an edge $e_{i,j} = (x_i, y_j)$ for $i, j = 1, 2$. Suppose we are given that $e_{1,1}$ got tentative color α and lost the lottery and that $e_{1,2}$ got tentative color β; we will argue intuitively that given this, the probability of $e_{1,2}$ losing the lottery has increased. Since $e_{1,1}$ lost, the probability of $e_{2,1}$ getting tentative color α increases, which implies that the probability of $e_{2,2}$ getting tentative color β also increases, and this increases the probability of $e_{1,2}$ losing the lottery.

For the sake of the analysis, we modify the algorithm as follows: instead of performing all random proposals in parallel, suppose that the bottom vertices perform their random proposals sequentially, in some order. This does not modify the probability distributions because the choices are still done independently. We want to focus our attention on the last vertex v_b performing the random proposal. We will use the fact that when v_b performs its random proposal, all edges not incident on v_b already have a tentative color. By symmetry, any upper bound on the probabilities we can find for v_b will hold for all bottom vertices.

Let $i \in [d_{v_b}]$ denote an edge incident with the bottom vertex v_b, with the other endpoint of i being u_i. We introduce the indicator random variables

$$X_i = \begin{cases} 1, & i \text{ loses the lottery,} \\ 0 & \text{otherwise,} \end{cases}$$

and we want to study the expectation and tail probability distribution of $X = \sum_{i\in[d_{v_b}]} X_i$. Computing the expectation is easy.

LEMMA 5.2. $E[X] \le \Delta/e$.

Proof. Let v_b be the bottom vertex. It is sufficient to show that $\Pr(X_i = 1) \le 1/e$ for all $i \in [d_{v_b}]$. From the analysis of the top vertices, we know that the expected number of losers incident with u_i is at most Δ/e and hence that $\sum_{j\in\delta(u_i)} \Pr(j \text{ loses}) \le \Delta/e$. By symmetry, $\Pr(j \text{ loses}) \le 1/e$ for all $j \in \delta(u_i)$, and hence $\Pr(X_i = 1) \le 1/e$. □

We now study the tail probability distribution of X. Our goal is to show that $X \le (1+\epsilon)\Delta/e$ w.h.p. for any fixed $\epsilon > 0$. Establishing this claim will take several lemmas.

We use a method different from the preliminary version of this work [16] to present stronger results. We first invoke a result of Schmidt, Siegel, and Srinivasan [20], which was in fact motivated in part by [16] and in particular by the notion of λ-correlation.

For $z = (z_1, z_2, \ldots, z_n) \in \Re^n$, define a family of symmetric polynomials $q_j(z)$, $0 \le j \le n$, where $q_0(z) \equiv 1$ and for $1 \le j \le n$,

$$q_j(z) \doteq \sum_{1\le i_1<i_2\cdots<i_j\le n} z_{i_1} z_{i_2} \cdots z_{i_j}.$$

THEOREM 5.3 (see [20]). *Let $Y_1, Y_2, \ldots, Y_n$ be arbitrary (not necessarily independent) 0–1 random variables with $Y = \sum_{i=1}^n Y_i$. Then for any $a > 0$ and any positive probability event Z,*

$$\Pr(Y \ge a|Z) \le \min_{\ell=1,\ldots,a} \frac{E[q_\ell(Y_1, Y_2, \ldots, Y_n)|Z]}{\binom{a}{\ell}}.$$

Proof. The actual theorem of [20] is stated unconditionally without reference to Z, but the above conditional extension is easily derivable from its proof as follows. Since the Y_i's are binary, it is easily seen that for any $\ell \le a$, given that Z occured, $(Y \ge a)$ implies $(q_\ell(Y_1, Y_2, \ldots, Y_n) \ge \binom{a}{\ell})$. Thus by Markov's inequality,

$$\Pr(Y \ge a|Z) \le \frac{E[q_\ell(Y_1, Y_2, \ldots, Y_n)|Z]}{\binom{a}{\ell}}. \qquad \square$$

To bound the upper tail of X, we will define an event $\mathcal{A}$ such that $\mathcal{A}$ happens w.h.p. and such that for a suitably chosen k,

$$\frac{E[q_k(X_1, \ldots, X_{d_{v_b}})|\mathcal{A}]}{\binom{\Delta(1+\epsilon)e^{-1}}{k}} = e^{-\Omega(\Delta\epsilon^2)}. \tag{3}$$

In combination with Theorem 5.3, this will show that $\Pr(X \le (1+\epsilon)\Delta/e)$ almost surely because

$$\begin{aligned}\Pr\left(X > \frac{(1+\epsilon)\Delta}{e}\right) &= \Pr\left(X > \frac{(1+\epsilon)\Delta}{e} \mid \mathcal{A}\right)\Pr(\mathcal{A}) \\ &\quad + \Pr\left(X > \frac{(1+\epsilon)\Delta}{e} \mid \mathcal{A}^c\right)\Pr(\mathcal{A}^c)\end{aligned} \tag{4}$$

$$\leq \Pr\left(X > \frac{(1+\epsilon)\Delta}{e} \mid \mathcal{A}\right) + \Pr(\mathcal{A}^c)$$
$$\leq \min_{i\in[d_{v_b}]} \frac{E[q_i(X_1,\ldots,X_{d_{v_b}})|\mathcal{A}]}{\binom{\Delta(1+\epsilon)e^{-1}}{i}} + \Pr(\mathcal{A}^c)$$
$$\leq \frac{E[q_k(X_1,\ldots,X_{d_{v_b}})|\mathcal{A}]}{\binom{\Delta(1+\epsilon)e^{-1}}{k}} + \Pr(\mathcal{A}^c),$$

which is small by assumption. (Note that $\mathcal{A}$ happens w.h.p.) To prove the upper bound (3), we will focus on a generic term $\Pr(\bigwedge_{i\in I} X_i = 1|\mathcal{A})$ in

$$E[q_k(X_1,\ldots,X_{d_{v_b}})|\mathcal{A}] = \sum_{I\subseteq[d_{v_b}],|I|=k} E\left[\prod_{i\in I} X_i|\mathcal{A}\right] = \sum_{I\subseteq[d_{v_b}],|I|=k} \Pr\left(\bigwedge_{i\in I} X_i = 1|\mathcal{A}\right),$$

which will also suggest to us a suitable choice for the event $\mathcal{A}$.

Consider then a generic subset $I = \{w_1, w_2, \ldots, w_k\} \subseteq [d_{v_b}]$ of edges incident on the bottom vertex v_b (corresponding to the neighbors $\{u_{w_i}\}$ of v_b), and let us see how to compute $\Pr(\bigwedge_{i\in I} X_i = 1)$. Without loss of generality, we assume $I = [k]$. Recall that we are analyzing the situation where v_b is the last vertex to perform its random proposal. This means that prior to the assignment of a tentative color to edge $i = (v_b, u_i)$, all other edges incident on u_i already have their tentative color. Using the balls-and-bins language, we can say that prior to throwing ball i at random into one of the bins at vertex u_i, all balls coming from the other neighbors of u_i have been thrown. We will think of i as a red ball and of the other edges at u_i as white balls. Once the red ball is thrown in, say, bin $\ell \in [\Delta]$, a winner is selected uniformly at random among all (i.e., red *and* white) balls in bin ℓ. All other balls, the losers, are discarded. Notice that the probability of discarding the red ball is itself a random variable which depends on the particular placement of the white balls prior to throwing the red ball. (Hence we will study the conditional probability that the red ball loses the lottery, given a placement of white balls.)

Given any placement of white balls at u_i, we construct a vector of probabilities C_i as follows. Let $a_{\ell,i}$ denote the number of white balls in bin $\ell \in [\Delta]$ of vertex u_i, and let $p_{\ell,i} = a_{\ell,i}/(1 + a_{\ell,i})$ denote the probability that the red ball loses the lottery given that it was thrown in bin ℓ. (Equivalently, $p_{\ell,i}$ is the probability that edge i loses given that it got tentative color ℓ.) For each neighbor u_i of our bottom vertex v_b, we construct the corresponding vector $C_i = (p_{1,i}, p_{2,i}, \ldots, p_{\Delta,i})$. We then construct a $\Delta \times k$ matrix M_I whose ith column is the vector C_i. The next lemma explains why this matrix is relevant to us. Henceforth, let $p(m,\ell) \doteq m(m-1)\cdots(m-\ell+1)$.

LEMMA 5.4.

$$\Pr\left(\bigwedge_{i\in I} X_i = 1\right) = \frac{perm(M_I)}{p(\Delta,k)}.$$

Proof. The random proposal of v_b restricted to I is equivalent to choosing a one-to-one function $\pi : I \to [\Delta]$ uniformly at random among the set $\mathcal{P}$ of all such functions. Recall that the entry $M_{i,j}$ of M_I is the probability $p_{i,j}$ that edge w_j loses given that it is given color i. Hence

$$\Pr(\wedge_{i\in I} X_i = 1) = \sum_{\pi\in\mathcal{P}} \Pr(\wedge_{i\in I} X_i = 1 \mid \pi \text{ is selected}) \Pr(\pi \text{ is selected})$$

$$= \sum_{\pi \in \mathcal{P}} \left(M_{\pi(1),1} M_{\pi(2),2} \cdots M_{\pi(k),k} \right) \prod_{i=0}^{k-1} \frac{1}{\Delta - i}$$
$$= \frac{perm(M_I)}{p(\Delta, k)}. \qquad \square$$

We now want to find a good upper bound for $perm(M_I)$. The following lemma gives a simple upper bound that is sufficient for our purposes.

LEMMA 5.5. *Let M be a matrix with c columns and r rows ($c \leq r$) and non-negative entries. Let S_i denote the sum of the entries of the ith column of M. Then* $\text{perm}(M) \leq \prod_{i \in [c]} S_i$.

Proof. Let $\mathcal{P} = \{\pi \mid \pi : [c] \to [r],\ \pi \text{ is one-to-one}\}$. Then

$$\begin{aligned} perm(M) &= \sum_{\pi \in \mathcal{P}} M_{\pi(1),1} M_{\pi(2),2} \cdots M_{\pi(c),c} \\ &\leq (M_{1,1} + \cdots + M_{r,1})(M_{1,2} + \cdots + M_{r,2}) \cdots (M_{1,c} + \cdots + M_{r,c}) \\ &= \prod_{i \in [c]} S_i. \qquad \square \end{aligned}$$

The next lemma relates the value of S_i to that of $1/e \geq \Pr(i \text{ loses})$, $i \in \delta(v_b)$. It is an application of the general definition of λ-correlation. Before the proof of the lemma, we establish the following result.

PROPOSITION 5.6. *If $0 \leq p \leq 1$, $q = 1 - p$, and m is a positive integer, then*

$$\sum_{r=1}^{m} \binom{m}{r} p^r q^{m-r} \frac{r}{r+1} = 1 - \frac{(1 - q^{m+1})}{p(m+1)}.$$

Proof. Let

$$\begin{aligned} f(p) &= \sum_{r=1}^{m} \binom{m}{r} p^r q^{m-r} \frac{k}{k+1} \\ &= 1 - q^m - \sum_{r=1}^{m} \binom{m}{r} p^r q^{m-r} \frac{1}{k+1}. \end{aligned}$$

Integrating both sides of the binomial expansion

$$(x+q)^m = \sum_{r=0}^{m} \binom{m}{r} x^r q^{m-r}$$

between 0 and p, we get

$$\frac{1 - q^{m+1}}{m+1} = p\,(1 - f(p)),$$

from which the proposition follows. $\square$

We now return to our scenario where v_b is the last bottom vertex to pick tentative colors for its edges. Recall that we are focusing on a set $I = \{w_1, w_2, \ldots, w_k\}$ of edges incident on v_b and that we want a good upper bound on $\Pr(\bigwedge_{i \in I} X_i = 1)$; we had also assumed that $I = [k]$ without loss of generality. Combining Lemmas 5.4 and 5.5, we get

$$\Pr\left(\bigwedge_{i \in I} X_i = 1 \right) \leq \frac{perm(M_I)}{p(\Delta, k)} \leq \frac{\Pi_{i=1}^{k} S_i}{p(\Delta, k)},$$

where for each $i \in [d_{v_b}]$, S_i is defined to be the sum of the entries in C_i. Therefore, a good upper bound on S_i for each i will hopefully translate into a good upper bound for $\Pr(\bigwedge_{i\in I} X_i = 1)$. The next lemma says that $S_i \le \Delta(1+\epsilon_1)/e$ w.h.p. for any fixed $\epsilon_1 > 0$ and for each i. Thus a good choice for $\mathcal{A}$ is

$$\mathcal{A} : \text{“}S_i \le \Delta(1+\epsilon_1)/e \text{ for each } i \in [d_{v_b}]\text{”}$$

where ϵ_1 will be fixed later. The next lemma is an application of the general definition of λ-correlation.

LEMMA 5.7. *Let $i = (v_b, u_i)$ be any edge in $[d_{v_b}]$, and let S_i be the sum of the entries of C_i. Then $E[S_i] \le \Delta/e = \mu$ and*

$$\forall \epsilon_1 > 0, \ \Pr(S_i > (1+\epsilon_1)\mu) \le F(\mu, \epsilon_1).$$

Proof. Let Z_ℓ be the random variable denoting the number of white balls in bin ℓ of u_i, and let $Y_\ell = Z_\ell/(Z_\ell + 1)$ be the random variable denoting the probability that the red ball loses the lottery given that it lands in bin ℓ. Then $S_i = Y \doteq \sum_\ell Y_\ell$. Note that the Y_ℓ's are bounded random variables with values in $[0,1]$. We will show that $E[Y] \le \Delta/e$ and that the Y_ℓ's are 1-correlated (under the general definition of λ–correlation), which will give our claim.

We may assume that the total number d of white balls equals $\Delta - 1$ (i.e., that the degree of u_i is Δ): $\Pr(Y > (1+\epsilon_1)\Delta/e)$ is maximized at $d = \Delta - 1$ since d varies from 1 to $\Delta - 1$. (To see this, assume $d = \Delta - 1 - \ell < \Delta - 1$. Add ℓ *yellow* balls to the white balls and run two experiments. In one experiment, throw the white and red balls and compute the probability that the red ball loses the lottery. In the other experiment, throw white, yellow, and red balls and again compute the probability that the red ball loses. In both experiments, let us look at the bin where the red ball fell. The probability that the red ball loses is $b/(b+1)$ for the first experiment and $(b+y)/(b+y+1)$ for the second, where b and y are, respectively, the number of white and yellow balls in the bin. Since $y \ge 0$, $b/(b+1) \le (b+y)/(b+y+1)$. If $Y_i(d)$ indicates the variable Y_i when u_i has degree d, then $Y_i(d) \le Y_i(\Delta)$ for all $i \in [\Delta]$ and $d \in [\Delta]$.)

First, we will show that for all i, $E[Y_i] \le 1/e$, and then we will show that for any set of ℓ indices $J \subseteq [\Delta]$ and strictly positive integers s_i,

$$E\left[\prod_{i\in J} Y_i^{s_i}\right] \le \frac{1}{e^\ell}. \tag{5}$$

Given this we can apply Corollary 3.3 by introducing n independent twin 0–1 random variables $\hat{Y}_i$ such that $E[\hat{Y}_i] = \Pr(\hat{Y}_i = 1) = 1/e$. Since the $\hat{Y}_i$'s are binary, inequality (5) is the same as

$$E\left[\prod_{i\in J} Y_i^{s_i}\right] \le \prod_{i\in J} E[\hat{Y}_i] = \prod_{i\in J} E[\hat{Y}_i^{s_i}],$$

which is to say that the Y_i's are 1-correlated. Noting that $0 \le Y_i \le 1$, it suffices to show that

$$E\left[\prod_{i\in J} Y_i\right] \le \frac{1}{e^\ell}. \tag{6}$$

Without loss of generality, we can assume $J = [\ell]$. We will prove inequality (6) by induction on $\ell \geq 1$; when $\ell = 1$,

$$\begin{aligned} E[Y_1] &= \sum_{r=0}^{\Delta-1} \binom{\Delta-1}{r} \left(\frac{1}{\Delta}\right)^r \left(1-\frac{1}{\Delta}\right)^{\Delta-1-r} \frac{r}{r+1} \\ &= \left(1-\frac{1}{\Delta}\right)^{\Delta} \leq \frac{1}{e}, \end{aligned}$$

where the second equality follows from Proposition 5.6. Notice that for all $j \in [\Delta]$, $E[Y_j] = E[Y_1] \leq 1/e$. When $\ell > 1$, the law of conditional probabilities gives

$$E\left[\prod_{i\in[\ell]} Y_i\right] = E[Y_1 Y_2 \cdots Y_{\ell-1}\, E[Y_\ell \mid Y_1 Y_2 \cdots Y_{\ell-1}]] \tag{7}$$

Suppose we show that for all *nonzero* $c_i \in [0,1]$ with $i \in [\ell-1]$,

$$E\left[Y_\ell \mid \bigwedge_{i=1}^{\ell-1} Y_i = c_i\right] \leq \frac{1}{e}; \tag{8}$$

then since the product $Y_1 Y_2 \cdots Y_{\ell-1}$ in equation (7) is zero when any c_i is zero, we see by induction on ℓ that

$$\begin{aligned} E\left[\prod_{i=1}^{\ell} Y_i\right] &= E[Y_1 Y_2 \cdots Y_{\ell-1}\, E[Y_\ell \mid Y_1 Y_2 \cdots Y_{\ell-1}]] \\ &\leq \frac{1}{e} E\left[\prod_{i=1}^{\ell-1} Y_i\right] \\ &\leq \frac{1}{e^\ell}. \end{aligned}$$

Hence the claim follows if we can show that inequality (8) holds.

If a_i denotes the number of white balls that fell into bin i, then $c_i = a_i/(a_i+1)$. Let $a = \sum_{i=1}^{\ell-1} a_i \geq \ell - 1$, $p = 1/(\Delta - \ell + 1)$, and $q = 1-p$. Then

$$\begin{aligned} E\left[Y_\ell \middle| \bigwedge_{i=1}^{\ell-1} Y_i = c_i\right] &= E\left[Y_\ell \middle| \bigwedge_{i=1}^{\ell-1} Z_i = a_i\right] \\ &= \sum_{r=1}^{\Delta-1-a} t(r,a), \end{aligned}$$

where

$$t(r,a) \doteq \binom{\Delta-1-a}{r} p^r q^{\Delta-1-a-r} \frac{r}{r+1}.$$

It is easy to check that $t(r,a) \geq t(r,a+1)$. As a consequence, the maximum value of $E[Y_\ell \mid \bigwedge_{i=1}^{\ell-1} Y_i = c_i]$ is attained at $a = \ell - 1$, in which case we have

$$\sum_{r=1}^{\Delta-1-a} t(r,a) = \sum_{r=1}^{\Delta-\ell} t(r,\ell-1)$$

$$= \sum_{r=1}^{\Delta-\ell} \left(\begin{array}{c} \Delta-\ell \\ r \end{array} \right) p^r q^{\Delta-\ell-r} \frac{r}{r+1}$$
$$= q^{\Delta-\ell+1} \leq \frac{1}{e}$$

by Proposition 5.6. □

We remark that a short proof of Lemma 5.7 can be derived using the elegant work of [7].

Define $\epsilon_1 \doteq \epsilon/10$. Thus defining the event $\mathcal{A}$ as "$S_i \leq \Delta(1+\epsilon_1)/e$ for each $i \in [d_{v_b}]$", Lemma 5.7 gives the bound

$$\Pr(\mathcal{A}^c) \leq \Delta F\left(\frac{\Delta}{e}, \epsilon_1\right). \tag{9}$$

Now given that $\mathcal{A}$ holds, Lemma 5.5 shows that $perm(M_I) \leq (\Delta(1+\epsilon_1)/e)^k$ and thus from Lemma 5.4,

$$\Pr\left(\bigwedge_{i\in I} X_i = 1 | \mathcal{A}\right) \leq \frac{\left(\frac{\Delta(1+\epsilon_1)}{e}\right)^k}{p(\Delta,k)}. \tag{10}$$

We now turn to defining k suitably to get a good tail bound. Invoking Theorem 5.3 for

$$X = \sum_{i\in[d_{v_b}]} X_i$$

in conjunction with (10), we see that if $a = \Delta(1+\epsilon)/e$, then

$$\begin{aligned}
\Pr(X > a|\mathcal{A}) &\leq \frac{E[q_i(X_1,\ldots,X_{d_{v_b}})|\mathcal{A}]}{\binom{a}{k}} \\
&\leq \frac{\binom{d_{v_b}}{k}\left(\frac{\Delta(1+\epsilon_1)}{e}\right)^k}{p(\Delta,k)\binom{a}{k}} \\
&\leq \frac{\binom{\Delta}{k}\left(\frac{\Delta(1+\epsilon_1)}{e}\right)^k}{p(\Delta,k)\binom{a}{k}} \\
&= \frac{\left(\frac{\Delta(1+\epsilon_1)}{e}\right)^k}{p\left(\frac{\Delta(1+\epsilon)}{e},k\right)}. \qquad (11)
\end{aligned}$$

To lower bound $p(\Delta(1+\epsilon)/e, k)$, we need the following result.

LEMMA 5.8. *For positive integers t and ℓ, $t^\ell/p(t,\ell) \leq e^{\ell^2/t}$ if $\ell \leq t/2$.*

Proof. We first note that $\ln(1-x) \geq -2x$ for $0 \leq x \leq 1/2$. This is true since if we define $f(x) \doteq \ln(1-x) + 2x$, then $f(0) = 0$ and $f'(x) = (1-2x)/(1-x)$, which is nonnegative for $0 \leq x \leq 1/2$. Now

$$\begin{aligned}
\frac{p(t,\ell)}{t^\ell} &= \prod_{i=1}^{\ell-1} \frac{(t-i)}{t} \\
&= \exp\left(\sum_{i=1}^{\ell-1} \ln\left(1 - \frac{i}{t}\right)\right) \\
&\geq \exp\left(-\sum_{i=1}^{\ell-1} \frac{2i}{t}\right) \qquad \left(\text{since } \ell \leq \frac{t}{2}\right) \\
&= \exp\left(-\frac{(\ell-1)\ell}{t}\right) \\
&\geq e^{-\ell^2/t}. \qquad \square
\end{aligned}$$

We now set $k = \lfloor \Delta\epsilon/3e \rfloor$. Using Lemma 5.8 and the facts $\epsilon_1 = \epsilon/10$, $1 + \epsilon \geq e^{\epsilon/2}$, and $1 + \epsilon_1 \leq e^{\epsilon_1}$, we see from (11) that

$$\Pr(X \geq \Delta(1+\epsilon)/e | \mathcal{A}) \leq e^{-\Omega(\Delta\epsilon^2)}. \tag{12}$$

Applying bounds (9) and (12) to (4), we finally arrive at

$$\Pr(X \geq \Delta(1+\epsilon)/e) \leq e^{-\Omega(\Delta\epsilon^2)}. \tag{13}$$

We can now see why the parameter THRESHOLD must be $\Omega(\log^{1+\delta} n)$: the failure probability (13) goes to zero superpolynomially fast if $\Delta = \Omega(\log^{1+\delta} n)$ for any fixed $\delta > 0$. Using (13), we conclude our analysis with the following result.

THEOREM 5.9. *The new degree of the graph after one iteration of Part* I *of the bipartite algorithm is at most* $(1+\epsilon)\Delta/e$ *w.h.p. for any fixed* $\epsilon > 0$.

6. Extensions and applications of the algorithm. Recently, Panconesi and Dubhashi have improved our bounds by presenting a randomized distributed edge-coloring algorithm that runs in polylogarithmic time and uses at most $\Delta(1 + o(1)) + O(\log n)$ colors w.h.p. [6]. However, we feel that this work has independent interest owing to the tools developed to analyze the algorithm. We now describe some recent applications of this work.

Our results on λ-correlation have been used to prove the performance of a randomized rounding technique for multicommodity flow (Young [23]) and to provide an elementary method to bound the upper tail of the number of prime factors of random integers (Srinivasan [21]). As mentioned in section 5.3, the work of [20], which expands the applicability of CH-type bounds to more nonindependent scenarios, was inspired in part by this work. Our results on upper tail bounds for sums of bounded λ-correlated random variables have been generalized in [20].

Our algorithm has also been used and extended in the context of emulating PRAM algorithms using a limited number of processors [5].

Acknowledgments. Our sincere thanks go to David Shmoys for his continued guidance and support. We are grateful to Éva Tardos for an important idea about analyzing the algorithm and to Suresh Chari, Devdatt Dubhashi, David Pearson, Desh Ranjan, Pankaj Rohatgi, and Stephen Vavasis for useful discussions. We also thank the referees for their several valuable comments.

REFERENCES

[1] N. Alon, J. Spencer, and P. Erdős, *The Probabilistic Method*, Wiley–Interscience Series, John Wiley, New York, 1992.

[2] B. Berger and J. Rompel, *Simulating* $(\log^c n)$*-wise independence in NC*, J. Assoc. Comput. Mach., 38 (1991), pp. 1026–1046.

[3] B. Bollobás, *Graph Theory*, Springer-Verlag, New York, 1979.

[4] H. Chernoff, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statist., 23 (1952), pp. 493–509.

[5] X. Deng and P. Dymond, *Minimizing communication phases in optimal parallel algorithms*, Technical Report 94-04, Department of Computer Science, York University, North York, ON, Canada, 1994.

[6] D. Dubhashi and A. Panconesi, *Near-optimal distributed edge coloring*, in Proc. 3rd Annual European Symposium on Algorithms (ESA '95), Lecture Notes in Comput. Sci. 979, Springer-Verlag, Berlin, 1995, pp. 448–459.

[7] D. Dubhashi and D. Ranjan, *Balls and bins: A study in correlations*, Technical Report RS-96-25, Basic Research in Computer Science (BRICS), University of Århus, Århus, Denmark, submitted.

[8] W. Hoeffding, *Probability inequalities for sums of bounded random variables*, Amer. Statist. Assoc. J., 58 (1963), pp. 13–30.

[9] R. Jain, K. Somalwar, J. Werth, and J. C. Browne, *Scheduling parallel I/O operations in multiple bus systems*, J. Parallel Distrib. Comput., 16 (1992), pp. 352–362.

[10] H. J. Karloff and D. B. Shmoys, *Efficient parallel algorithms for edge coloring problems*, J. Algorithms, 8 (1987), pp. 39–52.

[11] G. F. Lev, N. Pippenger, and L. G. Valiant, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. Comput., 30 (1981), pp. 93–100.

[12] N. Linial, *Locality in distributed graph algorithms*, SIAM J. Comput., 21 (1992), pp. 193–201.

[13] M. Luby, *Removing randomness in parallel computation without a processor penalty*, J. Comput. System Sci., 47 (1993), pp. 250–286.

[14] C. McDiarmid, *On the method of bounded differences*, in Surveys in Combinatorics, London Math. Soc. Lecture Notes Ser. 141, Cambridge University Press, Cambridge, UK, 1989, pp. 148–188.

[15] R. Motwani, J. Naor, and M. Naor, *The probabilistic method yields deterministic parallel algorithms*, J. Comput. System Sci., 49 (1994), pp. 478–516.

[16] A. Panconesi and A. Srinivasan, *Fast randomized algorithms for distributed edge coloring*, in Proc. ACM Symposium on Principles of Distributed Computing, ACM, New York, 1992, pp. 251–262.

[17] A. Panconesi and A. Srinivasan, *Improved distributed algorithms for coloring and network decomposition problems*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1992, pp. 581–592.

[18] P. Raghavan, *Lecture notes on randomized algorithms*, Technical Report RC 15340 (#68237), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1990.

[19] R. Raman, *The power of collision: Randomized parallel algorithms for chaining and integer sorting*, in Proc. 10th Annual Conference on the Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Comput. Sci. 472, Springer-Verlag, Berlin, 1990, pp. 161–175; also available as Technical Report 336, Department of Computer Science, University of Rochester, Rochester, NY, 1990 (revised 1991).

[20] J. P. Schmidt, A. Siegel, and A. Srinivasan, *Chernoff–Hoeffding bounds for applications with limited independence*, SIAM J. Discrete Math., 8 (1995), pp. 223–250.

[21] A. Srinivasan, *On the distribution of the number of prime factors of integers*, manuscript, 1993.

[22] V. G. Vizing, *On an estimate of the chromatic class of a p-graph*, Diskret. Anal., 3 (1964), pp. 25–30 (in Russian).

[23] N. Young, *Randomized rounding without solving the linear program*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1995, pp. 170–178.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 369–400, April 1997

004

RANDOM DEBATERS AND THE HARDNESS OF APPROXIMATING STOCHASTIC FUNCTIONS*

ANNE CONDON†, JOAN FEIGENBAUM‡, CARSTEN LUND§, AND PETER SHOR¶

Abstract. A *probabilistically checkable debate system* (PCDS) for a language L consists of a probabilistic polynomial-time verifier V and a debate between Player 1, who claims that the input x is in L, and Player 0, who claims that the input x is not in L. It is known that there is a PCDS for L in which V flips $O(\log n)$ coins and reads $O(1)$ bits of the debate if and only if L is in PSPACE [A. Condon, J. Feigenbaum, C. Lund, and P. Shor, *Chicago J. Theoret. Comput. Sci.*, 1995, No. 4]. In this paper, we restrict attention to RPCDSs, which are PCDSs in which Player 0 follows a very simple strategy: On each turn, Player 0 chooses uniformly at random from the set of legal moves. We prove the following result.

THEOREM. *L has an RPCDS in which the verifier flips $O(\log n)$ coins and reads $O(1)$ bits of the debate if and only if L is in* PSPACE.

This new characterization of PSPACE is used to show that certain *stochastic* PSPACE-hard functions are as hard to approximate closely as they are to compute exactly. Examples of such functions include optimization versions of Dynamic Graph Reliability, Stochastic Satisfiability, Mah-Jongg, Stochastic Generalized Geography, and other "games against nature" of the type introduced in [C. Papadimitriou, *J. Comput. System Sci.*, 31 (1985), pp. 288–301].

Key words. approximation algorithms, complexity theory, probabilistic games, proof systems, PSPACE

AMS subject classification. 68Q15

PII. S0097539793260738

1. Introduction.

Recently, there has been great progress in understanding the precision with which one can approximate solutions to NP-hard problems efficiently. Feige et al. [13], Arora et al. [2, 3], and others proved strong negative results for several fundamental problems such as Clique and Satisfiability. This progress has led to renewed study of approximation algorithms for PSPACE-hard problems.

Not surprisingly, PSPACE-hard problems also display a wide variation in the precision with which they can be approximated efficiently. The following results in the literature show that, with respect to a particular performance guarantee, some PSPACE-hard problems have efficient approximation algorithms, others have such algorithms if and only if NP = P, and still others have such algorithms if and only if PSPACE = P. One interesting class of results concern problems on hierarchically defined structures, such as graphs. Problems on these structures, and on a related class

* Received by the editors December 1, 1993; accepted for publication (in revised form) May 12, 1995. These results first appeared in "Random debaters and the hardness of approximating stochastic functions (extended abstract)", Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, 1993 [11]. They were presented in preliminary form at the 9th Annual IEEE Conference on Structure in Complexity Theory, Amsterdam, The Netherlands, June 1994.

http://www.siam.org/journals/sicomp/26-2/26073.html

† Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 57306 (condon@cs.wisc.edu). The research of this author was supported in part by NSF grants CCR-9100886 and CCR-9257241.

‡ AT&T Laboratories, Room 2C473, 600 Mountain Avenue, Murray Hill, NJ 07974-0636 (jf@research.att.com).

§ AT&T Laboratories, Room 2C324, 600 Mountain Avenue, Murray Hill, NJ 07974-0636 (lund@research.att.com).

¶ AT&T Laboratories, Room 2D149, 600 Mountain Avenue, Murray Hill, NJ 07974-0636 (shor@research.att.com).

of periodic structures, arise in many VLSI and scheduling applications (see, for example, [18] for references to these and other applications). Because such representations can implicitly describe a structure of exponential size, using just polynomial space, the associated problems are often PSPACE-hard. As early as 1981, Orlin [20] claimed a negative result on approximating a PSPACE-hard periodic version of the Knapsack problem, namely that a fully polynomial approximation scheme exists for this problem only if NP = P. Orlin did not address whether in fact it might be PSPACE-hard to approximate this function. On the positive side, Marathe et al. [18] recently developed constant-factor approximation algorithms for PSPACE-hard problems, including the Max Cut and Vertex Cover problems for certain restricted classes of hierarchically represented graphs. In related work, Marathe et al. [19] applied results of Arora et al. [2] to show that these same problems have polynomial-time approximation schemes if and only if NP = P.

In [10], we considered optimization versions of several other problems in PSPACE, including Quantified Satisfiability, Generalized Geography, and Finite Automata Intersection. Building on the techniques of [2, 3, 13], we showed that it is in fact PSPACE-hard to approximate these problems closely (where "closely" depends on the problem). Using direct reduction arguments, Hunt et al. [14] showed that some generalized quantified satisfiability problems (for example, satisfiability of quantified formulas in which "clauses" are not restricted to be disjunctions of literals) are PSPACE-hard to approximate.

An important class of PSPACE-hard problems not previously addressed in this literature is a class of *stochastic* problems that involve decision-making under uncertainty, as in the games against nature of Papadimitriou [21]. In this paper, we develop a new technique for showing that these stochastic PSPACE-hard problems are hard to approximate. Informally, these are problems in which the instances involve probabilities in some essential way. The probabilities may describe failures of arcs in a digraph or moves of one of the players in a game. Examples of functions that we prove are hard to approximate include optimization versions of Stochastic Satisfiability (SSAT) [21], Stochastic Generalized Geography (SGGEOG), Dynamic Graph Reliability (DGR) [21], and Mah-Jongg. We describe two of these problems in more detail below. Precise definitions of all of the functions of interest can be found in section 3.

Our technique for proving that these problems are hard to approximate is based on a new characterization of PSPACE in terms of debates between one powerful and one random player that are checked by a resource-limited verifier. Just as the "games against nature" model is a powerful tool in obtaining hardness results for stochastic problems such as those mentioned above, our new model of PSPACE proves to be a very useful and natural tool in obtaining nonapproximability results for these problems.

We also use the identity IP = PSPACE [16, 23] to derive nonapproximability results for other stochastic PSPACE-hard functions. Examples of the functions that yield to this technique include optimization versions of Dynamic Markov Process (DMP) [21] and Stochastic Coloring, as well as a different optimization version of Stochastic Satisfiability.

We now give two examples that illustrate the kind of problems to which our new technique applies. The first is a variant of Graph Reliability, a #P-complete problem studied by Valiant [25]: Given a directed, acyclic graph G, source and sink vertices s and t, and a failure probability $p(v, w)$ for each arc (v, w), what is the

probability that there is a path from s to t consisting exclusively of arcs that have not failed? Papadimitriou [21] defines Dynamic Graph Reliability as follows: The goal of a strategy is still to traverse the digraph from s to t. Now, however, for each vertex x and arc (v, w), there is a failure probability $p((v, w), x)$; the interpretation is that, if the current vertex is x, the probability that the arc (v, w) will fail before the next move is $p((v, w), x)$. DGR consists of those digraphs for which there exists a strategy for getting from s to t with probability at least $1/2$. A natural optimization problem is MAX-PROB DGR: Given a graph, vertices s and t, and a set $\{p((v, w), x)\}$ of failure probabilities, what is the probability of reaching t from s under an optimal strategy? We show in section 3.3 below that there is a constant $c > 0$ such that it is PSPACE-hard to approximate MAX-PROB DGR within ratio 2^{-n^c}. This implies, for example, that if there is a polynomial-time algorithm that, on input x, outputs a number in the range $[2^{-n^c}\text{MAX-PROB DGR}(x), 2^{n^c}\text{MAX-PROB DGR}(x)]$, then PSPACE = P.

Our second example is a solitaire version of Mah-Jongg that is widely available as a computer game. Mah-Jongg tiles are divided into sets of four matching tiles; we assume that there are an arbitrarily large number of tiles. Initially, the tiles are organized in a preset arrangement of rows, and the rows may be stacked on top of each other. As a result, some tiles are hidden under other tiles. A tile that is not hidden and is at the end of a row is said to be available. In a legal move, any pair of available matching tiles may be removed. The player wins the game if all tiles are removed by a sequence of legal moves. An instance of the Mah-Jongg game describes the arrangement of the rows and, in addition, the tiles that are not hidden. We let MAH-JONGG(x) be the maximum probability of winning the Mah-Jongg game with initial arrangement x of the tiles, assuming that the hidden tiles are randomly and uniformly permuted. We show that approximating the function MAH-JONGG within ratio n^{-c} is PSPACE-hard, where $c < 1$ is some constant. Thus, if there is a polynomial-time algorithm that, on input x, outputs a number in the range $[n^{-c}\text{MAH-JONGG}(x), n^c\text{MAH-JONGG}(x)]$, then PSPACE = P.

Our new characterization of PSPACE, which is used in proving nonapproximability results for these problems, builds on techniques developed in our previous work on *probabilistically checkable debate systems* (PCDSs) [10], which in turn builds on techniques of Arora et al. [2], Lund et al. [16], and Shamir [23]. In a PCDS for L, there are two computationally powerful players, 1 and 0, and a probabilistic polynomial-time verifier V. Players 1 and 0 play a game in which they alternate writing strings on a debate tape π. Player 1's goal is to convince V that an input $x \in L$, and Player 0's goal is to convince V that $x \notin L$. When the debate is over, V looks at x and π and decides whether $x \in L$ (Player 1 wins the debate) or $x \notin L$ (Player 0 wins the debate). Suppose V flips $O(r(n))$ coins and reads $O(q(n))$ bits of π. If, under the best strategies of Players 1 and 0, V's decision is correct with high probability, then we say that L is in $\text{PCD}(r(n), q(n))$. We showed in [10] that $\text{PCD}(\log n, 1) = \text{PSPACE}$. That is, any language in PSPACE can be recognized by a PCDS in which the verifier uses $O(\log n)$ coin flips and queries only a constant number of bits of the debate.

We now restrict attention to PCDSs in which Player 0 follows a very simple strategy—that of tossing coins. Specifically, whenever Player 0 writes a string of length $f(n)$ on the debate tape π, the string is chosen uniformly at random from $\{0, 1\}^{f(n)}$. We call such a debate system an RPCDS and denote by $\text{RPCD}(r(n), q(n))$ the class of languages recognized by RPCDSs in which the verifier flips $O(r(n))$ coins and reads $O(q(n))$ bits of π. We note that an Arthur–Merlin game [4] is an RPCDS

in which the verifier is deterministic and $q(n)$ is an arbitrary polynomial; that is, V reads the entire debate between Arthur (Player 0) and Merlin (Player 1) before deciding whether the input is in the language. Thus the class of languages accepted by Arthur–Merlin games is by definition RPCD$(0, \text{poly}(n))$ and is commonly denoted by IP. It is known that RPCD$(0, \text{poly}(n))$ = PSPACE [16, 23]. In this paper, we prove the following result.

THEOREM. RPCD$(\log n, 1)$ = PSPACE.

This theorem shows that a verifier that tosses $O(\log n)$ coins does not have to read the entire debate between Arthur and Merlin. In fact, only a constant number of bits of the debate are needed. Our result is another in a sequence of results on polynomial-time interactive complexity classes, starting with the result that IP = PSPACE, that show that "universal quantification" can be replaced by "random quantification with bounded error" without changing the complexity class.

In the rest of this section, we first define precisely the PCDS and RPCDS models. We then describe previous work on related complexity classes.

1.1. Preliminaries. In this section, we define both the PCDS model of [10] and the new RPDCS model. We conclude with some definitions relating to the approximability of PSPACE-hard functions.

A *probabilistically checkable debate system*, or PCDS, consists of a *verifier* V and a *debate format* D. The verifier is a probablistic polynomial-time Turing machine that takes as input a pair x, π, where $\pi \in \{0,1\}^*$, and outputs 1 or 0. We interpret these outputs to mean "Player 1 won the debate" and "Player 0 won the debate," respectively.

A debate format is a pair of polynomial-time computable functions $f(n), g(n)$. Informally, for a fixed n, a debate between two players, 0 and 1, consistent with format $f(n), g(n)$, contains $g(n)$ rounds. At round $i \geq 1$, Player i mod 2 chooses a string of length $f(n)$.

For each x of length n, corresponding to the debate format D is a *debate tree*. This is a complete binary tree of depth $f(n)g(n)$ such that, from any vertex, one edge is labeled 0 and the other is labeled 1. A *debate* is any binary string of length $f(n)g(n)$. Thus there is a one-to-one correspondence between debates and the paths in the debate tree. Moreover, a debate is the concatenation of $g(n)$ substrings of length $f(n)$. Each substring is called a *round* of the debate, and each debate of this debate tree has $g(n)$ rounds.

Again for a fixed x of length n, a *debate subtree* is a subtree of the debate tree of depth $f(n)g(n)$ such that each vertex at level i (the root is at level 0) has 1 child if i div $f(n)$ is even, and it has two children if i div $f(n)$ is odd. Informally, the debate subtree corresponds to a list of "responses" of Player 1 against all possible "arguments" of Player 0 in the debate, or, more succinctly, a "strategy" of Player 1.

A language L has a PCDS *with error probability* ϵ if there is a pair $(D = (f(n), g(n)), V)$ with the following properties.

1. For all x in L, there is a debate subtree on which, for all debates π labeling a path of this subtree, V outputs 1 with probability 1 on input x, π. In this case, we say that x is accepted by (D, V).

2. For all x not in L, on all debate subtrees, there exists a debate π labeling some path of the subtree such that V outputs 1 with probability at most ϵ on input x, π. In this case, we say that x is rejected by (D, V).

This definition allows "one-sided error," analogous to the type of errors that are allowed in the complexity class co-RP. (See, for example, Johnson [15] for a definition.)

The main result of [10] also holds for a "zero-sided error" definition, with three possible outputs, 1, 0, and Λ, for "Player 1 won," "Player 0 won," and "I don't know who won," respectively. In this case, the verifier must never declare the losing player to be a winner, but it may, in both the case that $x \in L$ and the case that $x \notin L$, say that it doesn't know who won.

We say that the verifier makes $q(n)$ queries if the number of bits of π read by the verifier is at most $q(n)$ when the input x is of size n. The verifier V in a PCDS is required to be *nonadaptive*, by which we mean that the bits of π read by V depend solely on the input and the coin flips. If L has a PCDS with error probability $1/3$ in which V flips $O(r(n))$ coins and reads $O(q(n))$ bits of π, we say that $L \in \mathrm{PCD}(r(n), q(n))$. The classical result of Chandra et al. [9] that PSPACE is equal to Alternating Polynomial Time can be restated as $\mathrm{PCD}(0, \mathrm{poly}(n)) = \mathrm{PSPACE}$. In [10], we showed that PSPACE is also equal to $\mathrm{PCD}(\log(n), 1)$.

We now focus on PCDSs in which Player 0 follows a very simple strategy—that of tossing coins. Informally, an RPCDS with debate format $D = (f(n), g(n))$ is a PCDS in which, at each even-numbered round, Player 0 moves by choosing a string in $\{0,1\}^{f(n)}$ uniformly at random and writing it on the debate tape. Formally, for a given a debate subtree (i.e., strategy of Player 1), we define the *overall probability* that V outputs 1 to be the average over all debates π in the subtree of the probability that V outputs 1 on debate π. A language L has an RPCDS with error probability ϵ if the following hold.

$1'$. For all x in L, there is a debate subtree for which the overall probability that V outputs 1 is 1. Again, we say that x is accepted by (D, V).

$2'$. For all x not in L, on all debate subtrees, the overall probability that V outputs 1 is at most ϵ. In this case, we say that x is rejected by (D, V).

Note that item $1'$ is equivalent to item 1 above. If L has an RPCDS with error probability $1/3$ in which V flips $O(r(n))$ coins and reads $O(q(n))$ bits of π, we say that $L \in \mathrm{RPCD}(r(n), q(n))$.

To conclude this section, we review some definitions relating to approximability of PSPACE-hard functions. Let f be any real-valued function with domain $D \subseteq \{0,1\}^*$. Let A be an algorithm that, on input $x \in \{0,1\}^*$, produces an output $A(x)$. We say that A *approximates* f *within ratio* $\epsilon(n)$, $0 < \epsilon(n) < 1$, if for all $x \in D$, $\epsilon(|x|) \leq A(x)/f(x) \leq 1/\epsilon(|x|)$. If algorithm A computes the function g, we also say that g approximates f within ratio ϵ.

We say that a function g is PSPACE-*hard* if $\mathrm{PSPACE} \subseteq \mathrm{P}^g$, i.e., if every language in PSPACE is polynomial-time reducible to g. By "approximating f within ratio $\epsilon(n)$ is PSPACE-hard," we mean that, if g approximates f within ratio $\epsilon(n)$, then g is PSPACE-hard.

1.2. Related work. The theory of probabilistically checkable debate systems developed here and in [10] plays the role for PSPACE that the theory of probabilistically checkable proof systems (PCPSs) plays for NP. A PCPS is simply a PCDS with just one player, Player 1, in which case the "debate" corresponds to a "proof." (See Arora et al. [2, 3] or Sudan [24] for an overview of PCPSs.) If the verifier is deterministic and reads all bits of the proof, the model is equivalent in power to a nondeterministic Turing machine, where the proof corresponds to the nondeterministic moves. Thus, by definition, $\mathrm{NP} = \mathrm{PCP}(0, \mathrm{poly}(n))$. Arora et al. [2] showed that $\mathrm{PCP}(\log n, 1) = \mathrm{NP}$. Their result shows that there is a dramatic tradeoff between the number of random bits available to the verifier of such a proof and the number of bits of the proof that the verifier has to read. The techniques used to prove this result are

used heavily in our results on PCDSs and RPCDSs. In the next two paragraphs, we discuss these results and the relationships between the proofs.

In [10], we developed the PCDS in order to extend the work of Arora et al. to PSPACE. The PCDS model is related to the alternating Turing machine model of Chandra et al. [9], just as the PCP model is related to the nondeterministic Turing machine model. Chandra et al. showed that PSPACE is precisely the set of languages recognized by two-player, perfect-information games, in which the referee is a deterministic polynomial-time machine that examines the entire game before deciding who wins. The alternating Turing machine model formalizes their notion of such a game. The PCDS model generalizes this game model of Chandra et al. by allowing the referee to flip coins; this generality allows one to study the tradeoff between the number of coins used and the number of bits of the game, or debate, that are examined by the referee. In current notation, the result of Chandra et al. is that PSPACE $= \mathrm{PCD}(0, \mathrm{poly}(n))$. In [10], we showed that $\mathrm{PCD}(\log n, 1) = \mathrm{PSPACE}$. Thus we get the same tradeoff between random bits and queries as was previously shown for proof systems. The proof that $\mathrm{PSPACE} \subseteq \mathrm{PCD}(\log n, 1)$ is done in two parts. One part shows that PSPACE is contained in the class of languages accepted by PCDSs in which the verifier reads only a constant number of *rounds* of the debate. The second part then shows how the constant number of *rounds* in this result can be replaced by a constant number of *bits*. This second part is proved by extending the work of Arora et al. on PCPSs.

The result IP = PSPACE [16, 23] shows that Chandra et al.'s characterization of PSPACE is true even if one of the two players uses the unsophisticated strategy of simply selecting a move at random. That is, polynomial-time alternating Turing machines and polynomial-round Arthur–Merlin games accept the same class of languages. The main result of this paper shows that this assumption that one player plays randomly does not destroy the tradeoff between the referee's use of randomness and the number of bits of the game that the referee reads. Namely, we prove that $\mathrm{RPCD}(\log n, 1) = \mathrm{PSPACE}$. Again, the proof that $\mathrm{PSPACE} \subseteq \mathrm{RPCD}(\log n, 1)$ is done in two parts. The first shows that PSPACE is contained in the class of languages accepted by PCDS's in which the verifier reads only a constant number of *rounds* of Player 1 and a constant number of *bits* of Player 0. The second part, just as in [10], shows how the constant number of rounds in this result can be replaced by a constant number of bits.

All of these results on PCPSs, PCDSs, and RPCDSs can be used in different ways to prove nonapproximability results for hard problems. The result of Arora et al. that $\mathrm{NP} = \mathrm{PCP}(\log n, 1)$ has been applied to prove that several NP-hard problems are hard to approximate closely. These problems include optimization versions of Satisfiability, Independent Set [2, 3], Clique [13], and Colorability [17]. For example, the MAX SAT function maps a Boolean formula in 3-conjunctive normal form to the maximum number of clauses of that formula that are simultaneously satisfied by some assignment to the variables. Bellare et al. [5] showed that there is no polynomial-time algorithm that can approximate MAX SAT within ratio 112/113, unless NP = P. More recently, Bellare and Sudan [6] improved 112/113 to 64/65, but their assumption is weaker than NP = P.

Similarly, the result that $\mathrm{PCD}(\log n, 1) = \mathrm{PSPACE}$ yields nonapproximability results for optimization versions of PSPACE-hard problems, including Quantified Satisfiability, Generalized Geography, and Finite Automata Intersection [10]. For example, the optimization version of Quantified Satisfiability is defined as follows.

Suppose that the variables of the formula are assigned values, in order of quantification, by two players, 0 and 1. Players 0 and 1 assign values to the universally and existentially quantified variables, respectively. If Player 1 can guarantee that k clauses of the formula will be satisfied, regardless of what Player 0 chooses, we say that k clauses of the formula are simultaneously satisfiable. The function MAX QSAT maps a quantified formula to its maximum number of simultaneously satisfiable clauses. In [10], we show that approximating MAX QSAT within some constant factor $c < 1$ is PSPACE-hard.

The tools developed in [10] are useful in proving nonapproximability results for PSPACE-hard problems that can be cast as two-person games between two powerful players. However, these tools do not seem to lead to similar proofs for the stochastic PSPACE-hard problems that are considered in this paper. Our new results on RPCDSs are used to obtain such proofs in section 3.

There has been other very recent work, both on approximation algorithms and on nonapproximability results for PSPACE-hard problems. Using direct reductions from variations of the Quantified Satisfiability problem, Hunt et al. [14] and Marathe et al. [18] showed that several PSPACE-hard problems are hard to approximate, unless PSPACE = P. These include algebraic problems and graph problems on hierarchically defined graphs. Marathe et al. [19] proved that several graph problems such as vertex cover and independent set, when restricted to planar, hierarchically defined graphs, are PSPACE-hard and yet *do* have polynomial-time approximation schemes. They also developed approximation algorithms for restricted optimization problems on periodically defined graphs. Such problems were proved to be PSPACE-hard by Orlin [20].

The rest of this paper is organized as follows. Our main result that RPCD($\log n, 1$) = PSPACE is proven in section 2. In section 3, we prove several nonapproximability results for stochastic functions, using this characterization of PSPACE. Finally, in section 4, we prove additional nonapproximability results, using the result that IP = PSPACE.

2. Language-recognition power. In this section, we prove our main result, that RPCD($\log n, 1$) = PSPACE. To prove the (harder) direction that PSPACE $\subseteq$ RPCD($\log n, 1$), we build on several techniques of Lund et al. [16], Shamir [23], Arora et al. [3, 2], and Condon et al. [10]. We first describe these results, and, in Lemma 2.3, we put them together to prove PSPACE $\subseteq$ RPCD($\log n, 1$).

The first result we need, Lemma 2.1, shows that in order to prove that PSPACE $\subseteq$ RPCD($\log n, 1$), it is sufficient to show that any language in PSPACE can be recognized by a RPCDS in which the verifier uses $O(\log n)$ random bits, reads a constant number of *rounds* of Player 1, and reads a constant number of bits of Player 0.

LEMMA 2.1. *Suppose L is accepted by an RPCDS (D, V) in which the verifier uses $O(\log n)$ random bits, reads a constant number of rounds of Player* 1*, and reads a constant number of bits of Player* 0*. Then L is accepted by an RPCDS (D', V') in which the verifier uses $O(\log n)$ random bits, reads a constant number of bits of Player* 1*, and reads a constant number of bits of Player* 0*.*

Proof (sketch). We showed in [10, Theorem 3.2] that any language recognized by a PCDS in which V flips $O(\log n)$ coins and reads $O(1)$ *rounds* of the debate is also recognized by a PCDS in which V flips $O(\log n)$ coins and reads $O(1)$ *bits* of the debate. The lemma follows by a straightforward modification of the proof of [10, Theorem 3.2].

Briefly, if D has N rounds on a given input, then D' has $N+1$ rounds. Let the players of (D, V) be 0 and 1 and the players of (D', V') be $0'$ and $1'$. Roughly, the idea is that, in rounds 1 through N, Player $1'$ plays as Player 1 does in debate D, except that each move is encoded using a special encoding function, known as the low-degree polynomial code [2]. The string written by $1'$ in round $N+1$ contains a proof, π_R, for each random string R of V. The proof π_R shows that V outputs 1 given random string R and the decoded debate of rounds $1, \ldots, N$. Moreover, V' can check this proof while examining only a constant number of bits.

The verifier V' chooses a random seed R and computes the indices $i_1, i_2, \ldots, i_q$ of rounds that V queries using the random seed R. Using a protocol of Arora et al. [2, 3], V' need only examine a constant number of bits of each round $i_1, i_2, \ldots, i_q$ and a constant number of bits of round $N+1$ in order to verify that V outputs 1 on random string R and the decoded debate of rounds $1, \ldots, N$.

The correctness of this protocol follows from Arora et al. [2, 3]. For details, see [10]. □

The next lemma is implicit in the proof that IP = PSPACE [16, 23].

LEMMA 2.2. *Let L be a language in* PSPACE *and $x = x_1 x_2 \ldots x_n$ be an input. Then there is a sequence of multivariate polynomials*

$$g_1(y_{1,1}, \ldots, y_{1,n_1}) \text{ (where } n_1 = n), \; g_2(y_{2,1}, \ldots, y_{2,n_2}), \ldots, g_m(y_{m,1}, \ldots, y_{m,n_m})$$

with the following properties.

1. *If $x \in L$, then $g_1(x_1, \ldots, x_n) = 1$, and, if $x \notin L$, then $g_1(x_1, \ldots, x_n) = 0$. (Thus membership in L can be reduced to a g_1-question.)*

2. *There exist polynomial-time computable functions $h_{1,i}$, $h_{2,i}$, and f_i such that $g_i(y) = f_i(g_{i+1}(h_{1,i}(y)), g_{i+1}(h_{2,i}(y)))$ for all y. (That is, one g_i-question reduces to two g_{i+1}-questions.)*

3. *Finally, g_m is a polynomial-time computable function, and the degree of all the polynomials is bounded by $d = \text{poly}(n)$.*

In [16, 23], each g_i is a polynomial that can be explicitly written as a formula of polynomial length in terms of sums and products but may result in a formula of exponential length when these sums and products are expanded.

Finally, we describe a technique, called the *polynomial verification technique*, that was proposed by Babai as a generalization of a technique first used by Lund et al. [16]. Roughly, this method "reduces" a *set* of questions of the form "Is the value of multivariate polynomial g at point a equal to v?" to *one* such question.

To describe this technique precisely, we need the following definitions. Given a (multivariate) polynomial g over a finite field F, a *g-question* is a pair (a, v), where a is an assignment of values in F to the indeterminates in g and v is a value in F. Let $\{(a_1, v_1), (a_2, v_2), \ldots, (a_l, v_l)\}$ be a set of g-questions. Let $L(t)$ be the interpolated polynomial of degree $l-1$ such that $L(j) = a_j$ for every $j = 1, 2, \ldots, l$. (In "$L(j)$," the symbol "j" is used as an abbreviation for "the jth element of the finite field F.") Let p be the (univariate) polynomial $g(L(t))$. Note that $p(j) = g(a_j)$ for $j = 1, 2, \ldots, l$.

The polynomial verification technique receives as input the set of g-questions and a polynomial p'. The technique outputs one g-question $(L(r), p'(r))$, where r is chosen uniformly at random from F. Suppose that $p'(j) = v_j$ for all j. Then the output has the following property. If all input g-questions are *good*, that is, $g(a_j) = v_j, 1 \le j \le l$, and $p' = p$, the output g-question is also good. Otherwise, with probability at least $1 - d(l-1)/|F|$, the output g-question is not good. Correctness in the latter case follows from the fact that two distinct polynomials of degree $d(l-1)$ can agree on

at most $d(l-1)$ points. Note that, since $p'(j) = v_j$ and $p(j) = g(a_j)$ for all j, then $p' \neq p$ because at least one question is not good. Thus the probability that a random point on p' agrees with p is low (at most $d(l-1)/|F|$). This completes the description of the polynomial verification technique.

To motivate our proof that PSPACE $\subseteq$ RPCD($\log n$,1), it is useful to review the proof that PSPACE $\subseteq$ IP. In that proof, an RPCDS for a language L in PSPACE is obtained by combining the polynomial verification technique and Lemma 2.2. Roughly, in each odd-numbered round $2k-1$ of the debate, Player 1 writes a g_k-question, where g_k is defined as in Lemma 2.2. In the first round, the g_1-question should be $((x_1, \ldots, x_n), 1)$, which is equivalent to claiming that $x \in L$. In round $2k+1$, where $k \geq 1$, the g_{k+1}-question written by Player 1 should be obtained from the g_k-question written by Player 1 in round $2k-1$ by first reducing the g_k-question to two g_{k+1}-questions, as in part 2 of Lemma 2.2, and then by randomly reducing these two questions to one g_{k+1}-question, using the polynomial verification technique. The random number r needed in this reduction is provided by Player 0 in round $2k$ and the polynomial p' is provided by Player 1 in round $2k-1$ (in addition to the g_k-question). To verify that x is indeed in L, the verifier does three things. First, the verifier checks that the g_1-question written by Player 1 is $((x_1, \ldots, x_n), 1)$. Second, the verifier checks that, for each $k \geq 1$, the g_{k+1}-question in round $2k+1$ is correctly computed from the g_k-question in round $2k-1$, according to the above description. Finally, the verifier checks that the g_m-question written in the last round of the debate is good; by property 3 of Lemma 2.2, this can be done in polynomial time. If all three checks are passed, the verifier accepts; otherwise, the verifier rejects. For details, see [16, 23].

Unfortunately, this protocol requires that the verifier read all rounds of the debate system. In Lemma 2.3, building on the ideas of the above protocol, we describe a new protocol in which the verifier need only read a constant number of rounds of Player 1 and a constant number of bits of Player 0. The key idea of the new protocol is to add much "redundancy" in the rounds of Player 1 in order to enable the verifier to check correctness while looking at only a constant number of rounds. Roughly, this is achieved by requiring that Player 1 write in each odd-numbered round not just a single g_k-question but also all the questions written in previous rounds.

In each odd-numbered round k and for each $i = 1, 2, \ldots, m$, Player 1 actually writes a (possibly empty) set of g_i-questions, all from some finite field F. In the first round, the only question that Player 1 writes should be $((x_1, \ldots, x_n), 1)$, which is a g_1-question, equivalent to claiming that $x \in L$. Player 1 claims that all g_i-questions written are good. In order to enable the verifier to verify this, Player 1 furthermore writes a univariate polynomial p_i for each i such that the set of g_i-questions is not empty in round k. Player 1 claims that, for every i, the polynomial p_i is the polynomial $g_i(L_i(t))$, where L_i is the polynomial interpolating the domains of the g_i-questions, as in the polynomial verification technique.

Player 0's random move at round $k+1$ supplies the random point $r \in F$ on the curve L to be used in the polynomial verification technique for each i. By first using the polynomial verification technique and then reducing the resulting g_i-question to two g_{i+1}-questions, as in Lemma 2.2, any polynomial number of g_i-questions can be probabilistically reduced to two g_{i+1}-questions in one round. The list of g_{i+1}-questions at round $k+2$ is the list of g_{i+1}-questions at round k, plus these additional two questions. Since at most two new g_i-questions are introduced in each odd round, the total number of g_i-questions in any round is polynomial. The main technical

contribution of our proof shows that the resulting "redundancy" is sufficient to enable the verifier to check the debate while examining $O(1)$ rounds.

Also, in round $4m+1$, Player 1 writes a sequence of strings $(r_2, r_4, \ldots, r_{4m})$. Player 1 claims that these are the random moves of Player 0 in the even-numbered rounds. This enables the verifier to read any of the random strings of Player 0 by examining only one round (of Player 1). The verifier can verify with high probability that Player 1 writes the correct strings by reading just a constant number of bits of the rounds of Player 0.

LEMMA 2.3. PSPACE $\subseteq$ RPCD$(\log n, 1)$.

Proof. Let L be a language in PSPACE. Lemma 2.1 shows that it is sufficient to construct an RPCDS for L in which the verifier uses $O(\log n)$ random bits, reads a constant number of rounds of Player 1, and reads a constant number of bits of Player 0.

The RPCDS is constructed as follows. The debate system has $4m+1$ rounds. In each odd-numbered round, for every $i = 1, 2, \ldots, m$, Player 1 writes a set of g_i-questions

$$\{(a_{i,1}, v_{i,1}), (a_{i,2}, v_{i,2}), \ldots, (a_{i,l_i}, v_{i,l_i})\},$$

where $a_{i,j} \in F^{n_i}$ and $v_{i,j} \in F$ for some finite field F whose size will be determined later. Here $l_1 = 1$; that is, there is only one g_1-question $(a_{1,1}, v_{1,1})$. Also, for $i > 1$, $l_i = 0$ if the round number $k \leq 2i-3$ and otherwise l_i increases by 2 in each subsequent odd-numbered round. That is, if $k > 2i-3$, the number of g_i-questions in odd-numbered round k is 2 more than the number of g_i-questions in round $k-2$. Furthermore, for each i such that $l_i > 0$, Player 1 writes a univariate polynomial $p_i(t)$ of degree at most $d(l_i - 1)$. Finally, in round $4m+1$, Player 1 also writes a sequence of strings $(r_2, r_4, \ldots, r_{4m})$. This completes the description of the debate.

Before describing the verifier, we need one definition. Let $\{(a'_{i,j}, v'_{i,j}),\ 1 \leq i \leq m,\ 1 \leq j \leq l'_i\}$ be the g_i-questions and $\{p'_i\}$ be the polynomials in odd-numbered round $k-2$, and let $\{(a_{i,j}, v_{i,j}), 1 \leq i \leq m, 1 \leq j \leq l_i\}$ be the g_i-questions and $\{p_i\}$ be the polynomials in round k, where $1 < k \leq 4m+1$. Let L'_i be the polynomial interpolating the points $\{a'_{i,j}\}$ in round $k-2$, as in the polynomial verification technique. For odd $k > 1$, we say that round k is *locally consistent with respect to* r if the following holds. First, for all i and j, $1 \leq i \leq m$, $1 \leq j \leq l_i$, $p_i(j) = v_{i,j}$. Also, for all i, $1 \leq i \leq m$, if $l'_{i-1} = 0$ then $l_i = 0$, and if $l'_{i-1} > 0$ then

$$(a_{i,1}, \ldots, a_{i,l_i}) = (a'_{i,1}, \ldots, a'_{i,l'_i}, h_{1,i-1}(L'_{i-1}(r)), h_{2,i-1}(L'_{i-1}(r))),$$

and

$$(v_{i,1}, \ldots, v_{i,l_i}) = (v'_{i,1}, \ldots, v'_{i,l'_i}, w_{i,1}, w_{i,2}),$$

where $p'_{i-1}(r) = f_{i-1}(w_{i,1}, w_{i,2})$. Note that $(L'_{i-1}(r), p'_{i-1}(r))$ is the single g_{i-1}-question obtained by applying the polynomial verification technique to the set of g_{i-1}-questions in round $k-1$. Applying property 2 of Lemma 2.2, one can check whether this single g_{i-1}-question is good by checking that $p'_{i-1}(r) = f_{i-1}(w_{i,1}, w_{i,2})$ and that the two g_i-questions

$$(h_{1,i-1}(L'_{i-1}(r)), w_{i,1}) \quad \text{and} \quad (h_{2,i-1}(L'_{i-1}(r)), w_{i,2})$$

are good. Thus round k is locally consistent with respect to r if, for each $i, 1 \leq i \leq m$, the list of g_i-questions at round k consists of the list of g_i-questions at round $k-2$,

plus two additional questions that can used to verify that the g_{i-1}-questions at round $k-2$ are good.

We now describe the protocol of the verifier V. V first reads the final round $4m+1$. V checks that, for all pairs $(a_{m,j}, v_{m,j})$ in round $4m+1$, $g_m(a_{m,j}) = v_{m,j}$ and also that $(a_{1,1}, v_{1,1}) = ((x_1, \ldots, x_n), 1)$. If not, V rejects. Otherwise, V reads a random odd-numbered round $k > 1$ and checks that (i) round k is *consistent* with the final round $4m+1$—that is, every pair $(a_{i,j}, v_{i,j})$ written in round k is also written in round $4m+1$—and that (ii) round k is locally consistent with respect to the string r_{k-1} written by Player 1 in round $4m+1$. Finally, V reads a random bit of round $k-1$ of Player 0 and checks that it is equal to the corresponding bit of the string r_{k-1} written by Player 1 in round $4m+1$. If all of these checks are satisfied, V accepts; otherwise, V rejects.

It is straightforward to show that if $x \in L$, then Player 1 has a strategy that causes V to accept with probability 1. Hence suppose that $x \notin L$. For a given run of the RPCDS, if, in the final round of the debate, $(a_{1,1}, v_{1,1}) \neq ((x_1, \ldots, x_n), 1)$, then V rejects; hence suppose that, in the final round, $(a_{1,1}, v_{1,1}) = ((x_1, \ldots, x_n), 1)$. Let S be the set of odd-numbered rounds $k > 1$ that are locally consistent with respect to the string r_{k-1} written by Player 1 in the final round, are consistent with the final round, and have $\Delta(r_{k-1}, s_{k-1}) < 1/3$, where s_{k-1} is the move of Player 0 in round $k-1$ and $\Delta(r, s)$ is the fraction of bits that differ in r and s. We say that such a run is *S-consistent*. We will show that, if Player 1 is S-consistent for some S with $|S| \geq m$, then, with very high probability, Player 1's move in round $4m+1$ contains a pair $(a_{m,j}, v_{m,j})$ for which $g_m(a_{m,j}) \neq v_{m,j}$. This follows from the next claim and the fact that the run is S-consistent.

CLAIM. *Let S contain the elements $i_1 < i_2 < \cdots < i_m$. If Player 1 is S-consistent and $g_1(x_1, \ldots, x_n) = 0$, then, for every $k = 1, 2, \ldots, m$, with probability at least $1 - 4mkd/|F|^{1/18}$ (computed over Player 0's coin tosses), there exists some j such that $g_k(a_{k,j}) \neq v_{k,j}$, where $(a_{k,j}, v_{k,j})$ are played in the i_kth round.*

Proof. The claim is proven by induction on k. For $k = 1$, the claim holds: Because Player 1 is S-consistent, the pair $((x_1, \ldots, x_n), 1)$ is written by Player 1, whereas by assumption $g_1(x_1, \ldots, x_n) = 0$.

Thus assume that, for some pair $(a_{k-1,j}, v_{k-1,j})$ in round i_{k-1}, $g_{k-1}(a_{k-1,j}) \neq v_{k-1,j}$. This happens with probability at least $1 - 4m(k-1)d/|F|^{1/18}$ by the inductive hypothesis. Because the run is S-consistent, this pair is also written in round $i_k - 2$. This implies that the polynomial p'_{k-1} played in round $i_k - 2$ is not the polynomial $g_{k-1}(L'_{k-1}(t))$. Thus if $r = r_{i_k - 1}$ is not a root of the polynomial $p'_{k-1}(t) - g_{k-1}(L'_{k-1}(t))$, then $p'_{k-1}(r) \neq g_{k-1}(L'_{k-1}(r))$, implying that either

$$g_k(h_{1,k-1}(L'_{k-1}(r))) \neq w_{k,1} \quad \text{or} \quad g_k(h_{2,k-1}(L'_{k-1}(r))) \neq w_{k,2}.$$

Otherwise, $p'_{k-1}(r) \neq f_{k-1}(w_{k,1}, w_{k,2})$, which contradicts the fact that round i_k is locally consistent with respect to r.

The claim now follows because the polynomial $p'_{k-1}(t) - g_{k-1}(L'_{k-1}(t))$ has at most $4md$ roots (since l'_{k-1} is always at most $4m$), and the number of s's such that $\Delta(s, r) < 1/3$ for some root r is less than $|F|^{17/18}$ (using Chernov bounds). □

The above claim shows that, if Player 1 plays S-consistently for any fixed S with $|S| \geq m$, then with probability at most $\text{poly}(n)/|F|^{1/18}$, the verifier V accepts. Thus to bound the probability that V accepts, on a run in which Player 1 plays S-consistent rounds for some S, where $|S| \geq m$, sum over all possible S to obtain a bound of $2^{2m}\,\text{poly}(n)/|F|^{1/18}$. The error is thus less than $1/3$ for sufficiently large F.

If Player 1 does not play S-consistently for any S of size greater than m, the verifier rejects with probability at least 1/6. This is because, with probability at least 1/2, V checks a round k that is not S-consistent, and then, with probability at least 1/3, V detects that this round is not S-consistent. □

Our main result on the complexity class RPCD($\log n, 1$) now follows easily.

THEOREM 2.4. RPCD($\log n, 1$) = PSPACE.

Proof. Lemma 2.3 proves one direction, that PSPACE $\subseteq$ RPCD($\log n, 1$). In order to prove the other direction, first note that RPCD(poly(n), poly(n)) $\subseteq$ RPCD(0, poly(n)) because, in the last round of the debate, Player 0 can supply the verifier of an RPCDS with random bits; hence the verifier needs no random bits as long as it can read all bits in the debate. Combining this with the result that RPCD(0, poly(n)) = IP = PSPACE [16, 23] gives Theorem 2.4. □

3. Nonapproximability results based on RPCD($\log n, 1$). In this section, we prove that several PSPACE-hard problems are hard to approximate. The PSPACE-complete language SSAT (or "Stochastic Satisfiability"), introduced by Papadimitriou [21], plays an important role in the proofs in this section. In section 3.1, we define the language SSAT and, using our result that PSPACE = RPCD($\log n, 1$), show that the corresponding optimization problem is hard to approximate. In the following three sections, we prove that optimization versions of Stochastic Generalized Geography, Dynamic Graph Reliability, and Mah-Jongg are all hard to approximate.

3.1. Stochastic Satisfiability (SSAT). An SSAT instance is a Boolean formula ϕ over the set of variables $\{x_1, \ldots, x_n\}$ in conjunctive normal form (CNF) with three literals per clause. The instance is in the language if there is a choice of Boolean value for x_1 such that for a random choice (with true and false each chosen with probability 1/2) of x_2, there is a choice for x_3, etc., so the probability that ϕ is satisfied is greater than 1/2. Think of an instance as a game between an existential player and a random player. For each odd value of i, the existential player chooses an optimal Boolean value for x_i, where "optimal" means "maximizes the number of clauses of ϕ that are satisfied." For each even value of i, the random player flips a fair coin to get a Boolean value for x_i. The odd-numbered variables are called existential variables, and the even-numbered variables are called random variables. The players choose boolean values in order, by increasing value of i. We define the function MAX-CLAUSE SSAT, whose value on a given instance ϕ is the expected number of clauses that are satisfied if the existential player follows an optimal strategy.

THEOREM 3.1. *There is a constant* $0 < c < 1$ *such that approximating* MAX-CLAUSE SSAT *within ratio* c *is* PSPACE*-hard.*

Proof. Let L be a language in PSPACE. From section 2, there is an RPCDS (D, V) for L, where V is polynomial-time bounded and uses $r(n) = O(\log n)$ random bits and $O(1)$ queries. Let $D = (f(n), g(n))$. Without loss of generality, we can assume that $f(n)$ is even for all n. We reduce the problem of deciding whether a string x is accepted by (D, V) to the problem of approximating the expected number of simultaneously satisfiable assignments of a quantified 3CNF formula within a constant factor.

To do this, it is sufficient to construct a formula from x and (D, V) such that if $x \in L$, then all clauses are simultaneously satisfiable, but, if $x \notin L$, then the expected number of simultaneously satisfiable clauses is a constant fraction < 1 of the total number of clauses. Let $|x| = n$. The formula has $f(n)g(n)$ variables, corresponding to the bits of a debate between Players 0 and 1, ordered as they appear in the debate. By adding extra variables (which will not appear in the clauses), we can ensure that the

variables corresponding to rounds of Player 1 are existential (odd-numbered) variables and those corresponding to rounds of Player 0 are random (even-numbered) variables.

For each sequence of random bits R of length $r(n)$, there is a subformula with $s = O(1)$ clauses. The subformula is satisfied by a truth assignment to the variables if and only if V outputs 1, when the query bits are as in the truth assignment. Such a subformula can be constructed using those variables corresponding to the bits of a debate that are queried on random sequence R. A constant number of additional existential variables are needed so that the subformula is in 3CNF form; these should be ordered after all variables corresponding to bits of the debate.

If x is accepted by (D, V), then there is a debate subtree for which the overall probability that V outputs 1 is 1. This implies that the expected fraction of simultaneously satisfiable clauses of the formula is 1 if the existential player assigns values to the existential variables according to this debate subtree. If x is not accepted by (D, V), then, for any debate subtree, the overall probability that V outputs 1 is at most 1/3. Thus, no matter how the existential variables are chosen, the expected fraction of subformulas satisfied is at most 1/3. Since each subformula contains $O(1)$ clauses, it follows that the expected fraction of clauses that can be simultaneously satisfied is a constant fraction < 1. □

3.2. Stochastic Generalized Geography (SGGEOG). In this section, we derive a nonapproximability result for a stochastic version of Generalized Geography. Generalized Geography, as defined by Schaefer [22], is a game played on a directed graph G with a distinguished vertex s. A marker is initially placed on s, and two players, 1 and 0, alternately move the marker along arcs of the graph, with the constraints that Player 1 moves first and that each arc can be used at most once. The first player unable to move loses. The language GGEOG is the set of pairs (G, s) on which Player 1 has a winning strategy.

In previous work [10], we defined the function MAX GGEOG that maps a pair (G, s) to the largest integer k such that Player 1 can force the game to be played for k moves. (Note that Player 1's objective is to keep the game going as long as possible, whether or not Player 1 ultimately wins.) Here we consider a variation, Stochastic Generalized Geography, in which Player 0 plays randomly. At each even-numbered move of the game, Player 0 simply chooses an arc uniformly at random among all of the unused arcs out of the vertex at which the marker currently sits. The objective of Player 1 is to maximize the length of the game. The function MAX SGGEOG maps a pair (G, s) to the expected length of the game that is achieved when Player 1 follows an optimal strategy.

THEOREM 3.2. *For any constant $0 < c < 1/2$, it is* PSPACE*-hard to approximate* MAX SGGEOG *within ratio n^{-c}, where n is the number of vertices of the graph.*

Proof. We present an approximability-preserving reduction from MAX-CLAUSE SSAT. A key part of our construction is a gadget (directed graph) with the following properties, where $\epsilon > 0$ is some constant. Let ϕ be an instance of SSAT with n variables and m clauses. The gadget constructed from ϕ has a source vertex and a destination vertex. If $\phi \in$ SSAT, then Player 1 has a strategy that guarantees that the destination is reached from the source, with the last move being by Player 0, if the geography game is played from the source vertex, starting with Player 0. However, if $\phi \notin$ SSAT, then on each strategy of Player 1, one of two facts hold: (i) the destination is reached from the source, with probability at most $1-\epsilon$, with the last move being by Player 0, or (ii) the destination is reached from the source, with probability at most $(1-\epsilon)^n$, with the last move being by Player 1.

Let the size of this gadget be bounded by $p(|\phi|)$ for some polynomial p of degree > 1. Given this gadget, the reduction builds an instance $((V, A), s)$ of MAX SGGEOG as follows. The directed graph (V, A) contains n independent copies of the gadget, say $g_1, \ldots, g_n$. The distinguished vertex s is an additional vertex, with a single arc to the source vertex of g_1. In addition, A contains a directed edge from the destination vertex of of gadget g_i to the source vertex of gadget g_{i+1}, for $1 \leq i < n$. Finally, (V, A) contains a "tail" starting at the destination vertex of g_n. The length of the tail is $(np(|\phi|))^K$, where K is some constant that we will determine later. That is, the tail consists of $(np(|\phi|))^K$ ordered vertices, with an edge between the ith and the $(i+1)$st and one additional edge between the destination vertex of the nth gadget g_n and the first vertex of the tail.

Then if $\phi \in$ SSAT, Player 1 has a strategy that guarantees that the tail is reached from s. Namely, Player 1 first moves to the source vertex of g_1, and then uses the strategy that guarantees that the destination vertex of g_1 is reached on a move of Player 0. From there, Player 1 follows the single arc to the source vertex of g_2, and so on, until the destination vertex of g_n is finally reached. At that point, the tail is traversed, and so the length of the game is at least $(np(|\phi|))^K$, that is, the length of the tail. However, if $\phi \notin$ SSAT, then the properties of the gadgets above guarantee that on all strategies of Player 1, the probability that the tail is reached is at most $(1-\epsilon)^n$. Hence the expected length of the game is at most $np(|\phi|)$ (that is, the size of the graph (V, A), excluding the tail) $+ (1-\epsilon)^n (np(|\phi|))^K$. Thus for sufficiently large ϕ, a multiplicative factor of approximately $(np(|\phi|))^{K-1}/2$ separates the expected length of the game in the cases $\phi \in$ SSAT, and $\phi \notin$ SSAT, respectively. The size (i.e., the number of nodes) of the new instance is $N = np(|\phi|) + (np(|\phi|))^K$. Thus an approximation within a factor of $N^{-(K-2)/2K} = N^{-(1/2-1/K)}$ would distinguish the two cases. This immediately yields the theorem.

It remains to explain how the gadget is constructed. Assume without loss of generality that n (the number of variables of ϕ) is odd and that MAX-CLAUSE SSAT$(\phi) \geq 1$. In the gadget (V_g, A_g), the vertex set V_g consists of V_1, the "variable-assignment vertices," V_2, the "clause vertices," V_3, the "staircase vertices," and in addition, a source vertex s_g and a destination vertex d_g.

$$V_1 = \left(\bigcup_{i=1}^{n} \{u_i, q_i, \overline{q_i}, w_i, \overline{w_i}, z_i, z_i'\} \right) \cup \{u_{n+1}\}.$$

V_2 consists of $\{y_1, \ldots, y_m\}$. V_3 contains $2n$ sets of vertices, each of size $3t$, where t will be chosen later; we denote them by

$$\{w_{i,1}, w_{i,1}', w_{i,1}'', \ldots, w_{i,t}, w_{i,t}', w_{i,t}''\} \quad \text{and} \quad \{\overline{w_{i,1}}, \overline{w_{i,1}'}, \overline{w_{i,1}''}, \ldots, \overline{w_{i,t}}, \overline{w_{i,t}'}, \overline{w_{i,t}''}\},$$

for $1 \leq i \leq n$.

The arc set A_g also consists of four parts, A_1, A_2, A_3, and A_4, that serve the following functions. A_1 connects the vertices of V_1 into a sequence of hexagons: for each odd value of i, $1 \leq i \leq n$, the digraph contains the arcs (u_i, q_i), $(u_i, \overline{q_i})$, (q_i, w_i), $(\overline{q_i}, \overline{w_i})$, (w_i, z_i), $(\overline{w_i}, z_i)$, (z_i, z_i'), (z_i', u_{i+1}); for each even value of i, $1 \leq i \leq n-1$, it contains the arcs (u_i, w_i), $(u_i, \overline{w_i})$, (w_i, q_i), $(\overline{w_i}, \overline{q_i})$, (q_i, z_i), $(\overline{q_i}, z_i)$, (z_i, z_i'), (z_i', u_{i+1}). A_2 connects the vertex u_{n+1} to the clause vertices and the clause vertices back to the hexagons as follows. For $1 \leq j \leq m$, there is an arc (u_{n+1}, y_j). For each literal in the jth clause of ϕ, there is an arc (y_j, v), where v is the "w-vertex" corresponding to this literal. For example, if the 17th clause of ϕ is $x_2 \vee \overline{x_7} \vee x_{12}$, the arcs (y_{17}, w_2), $(y_{17}, \overline{w_7})$,

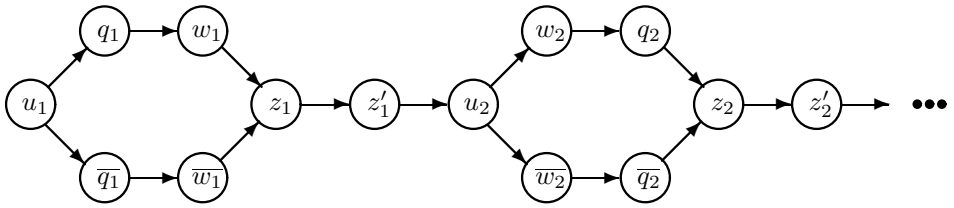


FIG. 1. *The start of the variable assignment (hexagon) section.*

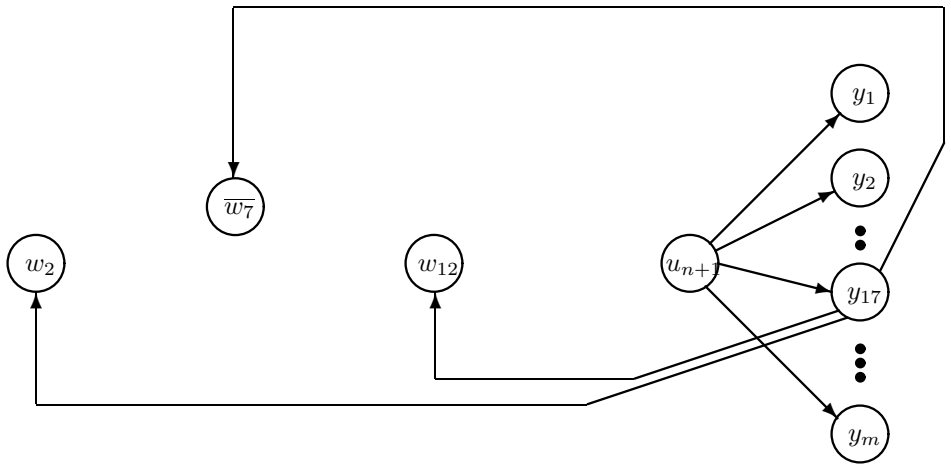


FIG. 2. *The arcs* (u_{n+}, y_j) *and those from* y_{17} *back to the hexagons, where* $c_{17} = x_2 \vee \overline{x_7} \vee x_{12}$.

and (y_{17}, w_{12}) are present. A_3 creates $2n$ "staircases" hanging off the vertices w_i and $\overline{w_i}$: for $1 \le i \le n$, the arcs $(w_i, w_{i,1})$, $(w_{i,1}, w'_{i,1})$, and $(w_{i,1}, w''_{i,1})$ are present, as are the arcs $(w''_{i,j}, w_{i,j+1})$, $(w_{i,j+1}, w'_{i,j+1})$, and $(w_{i,j+1}, w''_{i,j+1})$, for $1 \le j \le t-1$; analogous arcs form staircases emanating from the $\overline{w_i}$'s. Finally, A_4 contains the arc (s_g, u_1), which connects the source vertex s_g to the first variable-assignment vertex u_1, and also arcs $(w''_{i,t}, d_g)$ and $(\overline{w''_{i,t}}, d_g)$, $1 \le i \le n$, which connect the end of each staircase to the destination vertex d_g. Figures 1–3 give examples of hexagons, clause-arcs, and staircases.

Within a gadget (V_g, A_g), the Stochastic Generalized Geography game plays out as follows. We assume that Player 0 initially moves from the source vertex s_g along the single arc to vertex u_1. Player 1 (the existential player) assigns (optimally) either true or false to x_1. If Player 1 assigns true, the next two moves of the game are (u_1, q_1), played by 1, and (q_1, w_1), played by 0. Note that 0 has no choice but to move the marker along (q_1, w_1). Then Player 1 plays (w_1, z_1), 0 plays (z_1, z'_1), and 1 plays (z'_1, u_2). We will consider later the case where Player 1 chooses the arc $(w_1, w_{1,1})$. If Player 1 assigns false to x_1, then the analogous moves are made using the vertices $\overline{w_1}$, $\overline{q_1}$, etc. After (z'_1, u_2) has been traversed, Player 0 (the random player) assigns (with equal probability) true or false to x_2. If Player 0 assigns true, the next move of the game is (u_2, w_2), played by 0. Play continues along the arcs (w_2, q_2), (q_2, z_2), (z_2, z'_2), (z'_2, u_3), played by 1, 0, 1, 0. As before, if Player 0 assigns false to x_2, analogous moves are made using the vertices $\overline{w_2}$, $\overline{q_2}$, etc. From u_3, play continues in this fashion until Player 1 moves along either (z'_n, u_{n+1}) or $(\overline{z'_n}, u_{n+1})$, depending on the Boolean value chosen for x_n. This ends the "variable assignment" phase of the game.

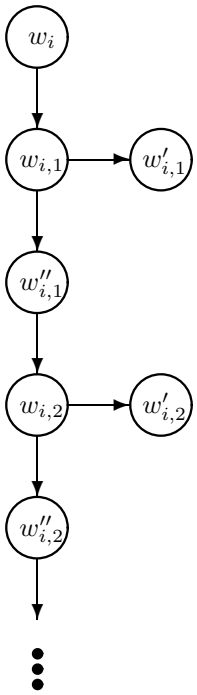


FIG. 3. *The start of the staircase emanating from* w_i.

Player 0 now chooses uniformly at random among the arcs (u_{n+1}, y_1), (u_{n+1}, y_2), $\ldots, (u_{n+1}, y_m)$. Suppose Player 0 chooses (u_{n+1}, y_j). If the jth clause of ϕ is satisfied by the assignment chosen in the first phase of the game, then Player 1 chooses an arc (y_j, w_i) (resp. $(y_j, \overline{w_i})$) corresponding to a true literal x_i (resp. $\overline{x_i}$) that appears in the jth clause. The game then takes $2t$ more steps because it follows the "spine" of the staircase emanating from w_i (resp. $\overline{w_i}$); Player 0 will have no choice but to take a step down the spine, and Player 1 will always choose to go one step further down the spine instead of off onto a "stair" because the game ends as soon as a stair is chosen. The end of the spine is reached on a move of Player 1, and finally Player 0 follows the single arc to the destination d_g. If the jth clause is not satisfied, then Player 1 must choose an arc (y_j, w_i) (resp. $(y_j, \overline{w_i})$) corresponding to a false literal x_i (resp. $\overline{x_i}$). For concreteness, suppose that Player 1 chooses the arc (y_j, w_i). Both of the arcs from w_i are unused in this case; in the previous case, when x_i was a true literal, only the arc $(w_i, w_{i,1})$ was unused. Player 0 chooses each unused arc with probability 1/2. If Player 0 chooses $(w_i, w_{i,1})$, the game continues for $2t$ moves as before to destination d_g; otherwise, the game ends in $O(1)$ moves and d_g is not reached.

The probability of reaching d_g is thus equal to 1 if $\phi \in$ SSAT since in this case Player 1 can guarantee that all clauses are satisfied by the variable assignment. However, if $\phi \notin$ SSAT, then the probability that a given clause is satisfied by the variable assignment chosen in the first phase of the gadget game is at most k/m, where $k =$ MAX-CLAUSE SSAT(ϕ). If in the second phase of the game, Player 0 chooses a clause that is not satisfied, then with probability 1/2, the destination is not reached. Hence the probability of reaching d_g is at most $k/m + (1 - k/m)/2 \leq 1 - \epsilon$, for some $\epsilon > 0$, by Theorem 3.1. Thus, if $\phi \notin$ SSAT, then on a strategy of Player 1 that never goes off into a staircase, fact (i) of the first paragraph of the proof holds. It remains to

consider those strategies of Player 1 that go off into a staircase instead of continuing down the string of hexagons. If Player 1 chooses, say, the arc $(w_i, w_{i,1})$ on the first move from w_i, then Player 0 chooses between the stair arc $(w_{i,1}, w'_{i,1})$ and the spine arc $(w_{i,1}, w''_{i,1})$. With probability 1/2, the stair arc would be chosen, and the game would end after one more step. If the spine arc were chosen, the same choice would confront Player 0 after one more step. Thus the probability of reaching the end of the spine is 2^{-t}, where t is the number of stairs. We choose t so that $2^{-t} < (1-\epsilon)^n$ to ensure that in this case fact (ii) holds. □

We note that the result of Theorem 3.2 could also be proved using a reduction from the MAX-PROB SSAT function, defined in section 4.

3.3. Dynamic Graph Reliability (DGR). Graph Reliability is a #P-complete problem studied by Valiant [25]: Given a directed, acyclic graph G, source and sink vertices s and t, and a failure probability $p(v, w)$ for each arc (v, w), what is the probability that there is a path from s to t consisting exclusively of arcs that have not failed? Papadimitriou [21] defines Dynamic Graph Reliability as follows: The goal of a strategy is still to traverse the digraph from s to t. Now, however, for each vertex x and each arc (v, w), there is a failure probability $p((v, w), x)$; the interpretation is that if the current vertex is x, the probability that the arc (v, w) will fail before the next move is $p((v, w), x)$. The language DGR consists of those digraphs for which there exists a strategy for getting from s to t with probability at least 1/2. A natural optimization problem is MAX-PROB DGR: Given a graph, vertices s and t, and a set $\{p((v, w), x)\}$ of failure probabilities, what is the probability of reaching t from s under an optimal strategy?

To obtain a nonapproximability result for MAX-PROB DGR, we need the following variant of Theorem 3.1.

THEOREM 3.3. *Consider the restriction of* SSAT *to instances in which the random variables appear only nonnegated and there is at most one random variable per clause. There are constants* $0 < c_1 < c_2 < 1$ *such that, given such a restricted instance with* m *clauses, it is* PSPACE*-hard to decide whether on average at least* $c_2 m$ *clauses are satisfiable or whether at most* $c_1 m$ *clauses are satisfiable.*

Proof. From the proof of Theorem 3.1, we can conclude that, given a formula ϕ, it is PSPACE-hard to distinguish between the cases that the expected number of simultaneously satisfiable clauses is at most $c_1 m$ or at least $c_2 m$, for some constants c_1 and c_2 such that $0 < c_1 < c_2 < 1$. It also follows from this proof, together with the proof of Theorem 2.4, that this is true for instances with only one random variable per clause. To see this, note that in the proof of Theorem 2.4, the verifier queries only one of Player 0's bits; thus each clause of any subformula in the construction of Theorem 3.1 has only one random variable.

We now show how to modify that construction so that no random variable is negated in the resulting formula. To do this, we use the identity

$$\bar{x}yz = \bar{x} + yz + xy\bar{z} + x\bar{y}z + x\bar{y}\bar{z} - 3,$$

where concatenation means "or" and "+" and "−" mean addition and subtraction over $\mathbf{Z}$. Suppose ϕ contains n variables and m clauses, of which r contain a negated random variable. We replace any clause $\bar{x}yz$ containing a negated random variable x and literals y and z by the four clauses yz, $xy\bar{z}$, $x\bar{y}z$, and $x\bar{y}\bar{z}$. We now have a formula ϕ' containing n variables and $m + 3r$ clauses. We claim that

$$\text{MAX-CLAUSE SSAT}(\phi') = \text{MAX-CLAUSE SSAT}(\phi) + 5r/2.$$

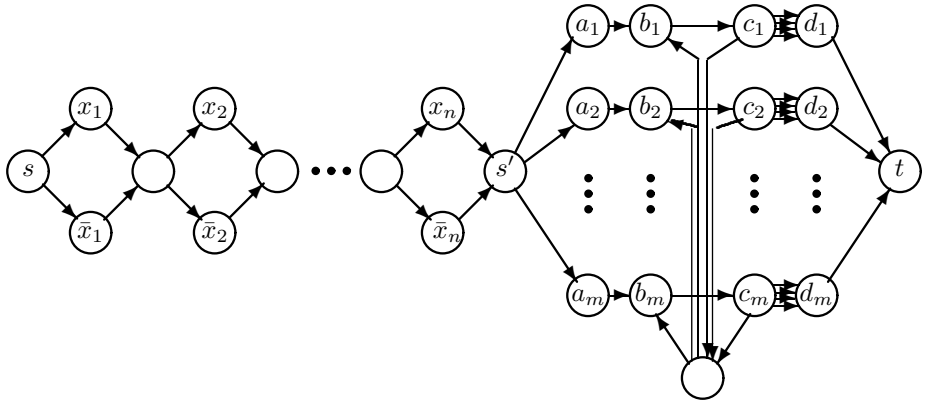


FIG. 4. *The graph constructed in reducing* MAX-CLAUSE SSAT *to* DGR. *Note that all edges are directed from left to right except those incident to r, which go from right to left.*

This formula is derived from the above identity; the term $\bar{x}$ is satisfied exactly half the time, and after subtracting 3, we find that we expect 5/2 additional clauses to be satisfied in ϕ' for each of the r substitutions that we made. □

THEOREM 3.4. *There is a constant $c > 0$ such that approximating* MAX-PROB DGR *within ratio 2^{-n^c} is* PSPACE*-hard.*

Proof. We show how to reduce the problem of approximating the MAX-CLAUSE SSAT function on an instance of the form given in Theorem 3.3 to the problem of approximating the Dynamic Graph Reliability function. The target DGR instance is composed of two parts, a variable-setting component and a set of clause-testing components. The construction is illustrated in Figure 4. The variable-setting component consists of the n "diamonds" in this figure, and the ith clause-testing component lies between vertices a_i and d_i. There are four edges between each pair c_i and d_i. Three of these correspond to the three literals in the clause, and the fourth is a "bypass edge."

Each path from s through the variable-setting component corresponds to an assignment of the variables in order. If x_i is an existential variable, both of the edges into vertices x_i and $\bar{x}_i$ are available, so the strategy determines which vertex is in the path. If the vertex x_i is chosen, all of the edges in the clause-testing components corresponding to $\bar{x}_i$ fail with probability 1, and similarly, if $\bar{x}_i$ is chosen, all of the edges corresponding to x_i in the clause-testing components fail. If x_i is a random variable, upon reaching the vertex immediately before x_i, with probability 1/2 the edge to x_i fails. If it does, the path must go through $\bar{x}_i$, in which case all of the edges corresponding to x_i in the clause-testing components fail. Otherwise, the path may go through $\bar{x}_i$ or x_i. Since $\bar{x}_i$ never appears in any clauses, it can only help to go through the x_i node, so we may assume that the strategy is consistent with this.

From vertex s', the path leaves the variable-setting component and enters one of m clause-testing components. When s' is reached, each edge $(b_i, c_i), 1 \le i \le m$, fails with probability $1 - 1/m$. If all edges (b_i, c_i) fail, clearly t cannot be reached from s'. We next show that if if two or more of these edges survive, t is always reachable, and if exactly one of these edges survives, t is reachable only if the corresponding clause is satisfied. Thus t is reachable with probability approximately $1 - 2/e + k/(me)$, where k is the number of satisfied clauses and e is 2.71828.... From Theorem 3.3, it now follows that for some constants $c_1 < c_2$, it is PSPACE-hard to distinguish between

the cases MAX-PROB DGR(x) $< c_1$ and MAX-PROB DGR(x) $> c_2$.

Recall that three of the four edges between nodes c_i and d_i correspond to the three literals in the clause, and each will be present when s' is reached if the corresponding literal is true. If a_i is visited, then the bypass edge for clause i fails. If exactly one edge (b_i, c_i) survives, clearly t can be reached only if the corresponding clause is satisfied, in which case the path contains $s', a_i, b_i, c_i, d_i, t$. Finally, if two edges (b_i, c_i) and (b_j, c_j) survive, then the path $s', a_i, b_i, c_i, r, b_j, c_j, d_j, t$ reaches t. Note that in this case the bypass edge between c_j and d_j is available because node a_j is never visited.

Finally, we strengthen our result by making several copies of this construction and repeating it in series and in parallel. To show this, we use a result of Ajtai and Ben-Or [1] (see Theorem 3.14 in [8]): For every probabilistic circuit C of size s that accepts a language L with error probability $\epsilon < 1/2$, there is a probabilistic circuit C' of size $s\ \mathrm{poly}(N)$ that accepts L with error probability $< 2^{-N}$. In fact, the circuit C' has the following structure:

$$\bigvee^{2\log N}\left(\bigwedge^{2N^2\log N}\left(\bigvee^{N^3}\left(\bigwedge^{N} C\right)\right)\right).$$

To apply this result to our construction, we replace $\wedge$ by repeating the construction in series (i.e., taking two copies and connecting the sink in the first copy with the source in the second) and replace $\vee$ by repeating the construction in parallel (i.e., taking two copies and connecting s' in the first copy with the source in the second and the sink in the first copy with the sink in the second). The number of edges of the resulting construction is then $O(sN^{6+\epsilon})$ for any $\epsilon > 0$, where s is the size of the construction above. From this, the theorem follows, where c is any constant less than $1/6$ (assuming that the instance is encoded such that the length is $O(sN^{6+\epsilon}\log(sN))$, which can easily be done). □

3.4. A solitaire game using Mah-Jongg tiles (MAH-JONGG). Solitaire Mah-Jongg, widely available as a computer game, is played roughly as follows. The game uses Mah-Jongg tiles, which are divided into sets of two or four matching tiles. We generalize the standard game simply by assuming that there are an arbitrarily large number of tiles. Initially, the tiles are organized in a preset arrangement of rows, and the rows may be stacked on top of each other. As a result, some tiles are hidden under other tiles; it is assumed that each possible arrangement of the hidden tiles is equally likely. A tile that is not hidden and is at the end of a row is said to be available. In a legal move, any pair of available matching tiles may be removed, resulting in a new configuration of the tiles in which up to two previously hidden tiles are uncovered. The player wins the game if all tiles are removed by a sequence of legal moves. MAH-JONGG(x) is defined to be the maximum probability of winning the generalized Mah-Jongg game from initial arrangement x of the tiles, assuming that the hidden tiles are randomly and uniformly permuted.

To define the game precisely, we define a set of Mah-Jongg tiles to be $\mathcal{T} = \cup_i \mathcal{T}_i$, where $\mathcal{T}_1, \ldots, \mathcal{T}_t$ are disjoint sets of tiles, each set being of size 2 or 4. We say tiles T_1 and T_2 *match* if and only if for some i, $T_1, T_2 \in \mathcal{T}_i$. A *configuration* C is a set of positions (i, j, k), where each of i, j, and k is a nonnegative integer, satisfying the following constraints.

1. If $(i, j, k) \in C$ and $(i, j', k) \in C$, where $j < j'$, then for every j'' in the range $[j, j']$, $(i, j'', k) \in C$.
2. If $(i, j, k) \in C$, where $k > 0$, then $(i, j, k-1) \in C$.

Intuitively, this captures the fact that tiles are arranged in three dimensions. Tiles can be stacked on up of each other; all tiles with common k are at the same height. Tiles at the same height with common i index form a row. The first condition ensures that there cannot be "gaps" in a row; the second ensures that a tile at height $k > 0$ must have a tile underneath it.)

With respect to a given configuration, a position (i, j, k) is *hidden* if $(i, j, k + 1)$ is also in the configuration. An *arrangement* consists of a set of Mah-Jongg tiles $\mathcal{T}$, a configuration C of size $|\mathcal{T}|$, and a 1–1 function from the positions of C that are not hidden into $\mathcal{T}$. If this function maps position (i, j, k) to tile T, we say that T is *in position* (i, j, k). With respect to a given arrangement, we say that a position (i, j, k) is *available* if it is not hidden and either position $(i, j - 1, k)$ or $(i, j + 1, k)$ is not in the configuration. An arrangement is *empty* if $\mathcal{T}$ is empty.

Let $x = (\mathcal{T}, C, f)$ be an arrangement. Each pair $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$ of available positions at which there are matching tiles T_1 and T_2 defines a *match* of x. We say that arrangement x' is obtainable from x via match $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$ if $x' = (\mathcal{T}', C', f')$, where $\mathcal{T}' = \mathcal{T} - \{T_1, T_2\}$, $C' = C - \{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$, and finally, f' and f agree on the set of positions which are not hidden in both C and C'. (Note that f' is not defined on positions (i_1, j_1, k_1) and (i_2, j_2, k_2), and f' is defined on position $(i_1, j_1, k_1 - 1)$ if $k_1 > 0$ and on position $(i_2, j_2, k_2 - 1)$ if $k_2 > 0$ since these are no longer hidden.) Corresponding to each match $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$ of x is a *move*, which results in an arrangement x', chosen uniformly and randomly from the set of arrangements obtainable from x via match $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$. A sequence of moves from arrangement x leads to a win if it results in the empty arrangement. A *strategy* on x associates a match (if any) with each possible arrangement obtainable from x via any sequence of moves.

An instance of the Mah-Jongg game is an arrangement x. MAH-JONGG(x) is the maximum probability of a win from x, where the maximum is taken over all strategies on x.

In Theorem 3.6, we show that it is PSPACE-hard to approximate the MAH-JONGG function within ratio n^{-c} for some constant $c < 1$. Our result holds for instances x in which tiles are never stacked more than two deep (see Figure 5). We use the following lemma in our reduction.

LEMMA 3.5. *Let $q(n)$ be any polynomial, and let $\epsilon > 0$ be any constant. Let ϕ' be an instance of* SSAT *with no negated random variables. Then there is an instance* SSAT *ϕ with no negated random variables that can be efficiently constructed from ϕ', with the following properties.*

(a) *If the expected fraction of simultaneously satisfiable clauses of ϕ' is at most $c_1 - \epsilon(\geq 0)$, then on any strategy for assigning values to the existential variables of ϕ, with probability at least $1 - 1/q(|\phi'|)$, the fraction of satisfied clauses of ϕ is at most c_1.*

(b) *If the expected fraction of simultaneously satisfiable clauses of ϕ' is at least $c_2 + \epsilon(\leq 1)$, then there is a strategy for assigning values to the existential variables of ϕ such that with probability at least $1 - 1/q(|\phi'|)$, the fraction of satisfied clauses of ϕ is at least c_2.*

Proof. ϕ is simply composed of many independent copies of ϕ', as follows. If

$$\phi' = \exists x_1 \forall x_2 \ldots \exists x_{n'} f(x_1, \ldots, x_{n'}),$$

then for some $p = p(|\phi|)$ to be chosen later,

$$\phi = \exists x_{11} \forall x_{12} \ldots \exists x_{1n'} \ldots \exists x_{p1} \forall x_{p2} \ldots \exists x_{pn'} f(x_{11}, \ldots, x_{1n'}) \wedge \cdots \wedge f(x_{p1}, \ldots, x_{pn'}).$$

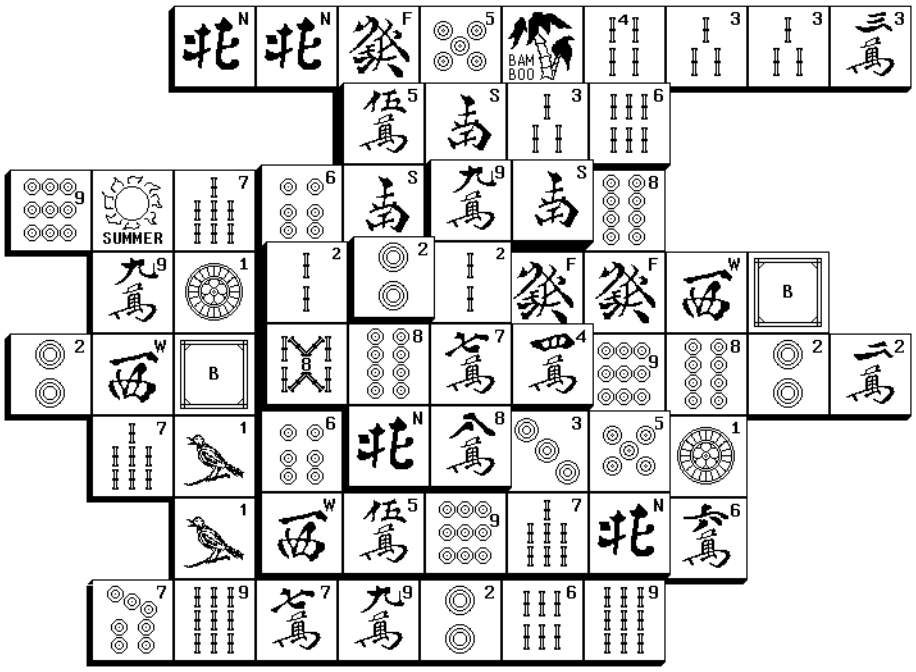


FIG. 5. *The arrangement of tiles part way through a game of Xmahjongg. Possible moves include removing the six of dots in the third row and the six of dots in the sixth row, removing the pair of twos of bamboo in the fourth row, and removing any two of the three available north tiles (in the first, sixth, and seventh rows). Xmahjongg is copyright* 1989 *by Jeff S. Young, and the tile designs are copyright* 1988 *by Mark A. Holm (used with permission).*

Fix any strategy for assigning values to the existential variables of ϕ. Let random variable $X_i, 1 \leq i \leq p$, be the fraction of clauses that are satisfied in the subformula $f(x_{i1}, \ldots, x_{in'})$. Let random variable Y be the fraction of satisfied clauses of ϕ. Then $Y = (1/p)\sum_{i=1}^{p} X_i$. If $\mathrm{Var}(X_i)$ is the variance of X_i, then the variance of Y is

$$\mathrm{Var}(Y) = \sum_{i=1}^{p} \mathrm{Var}(X_i/p) \leq 1/p.$$

To prove part (a), suppose that ϕ' is such that the expected fraction of simultaneously satisfiable clauses is at most $c_1 - \epsilon$. Then the expected value of X_i, and hence of Y, is at most $c_1 - \epsilon$ for all i. Thus, from Chebyshev's inequality, $\mathrm{Prob}[Y \geq c_1] \leq 1/(\epsilon^2 p)$. If $p(|\phi'|) \geq (1/\epsilon^2)q(|\phi'|)$, then with probability at least $1 - 1/q(|\phi|)$, $Y \leq c_1$. The proof of part (b) is similar. □

THEOREM 3.6. *There is a constant* $0 < c < 1$ *such that approximating* MAH-JONGG *within ratio* n^{-c} *is* PSPACE*-hard.*

Proof. To prove this result, we show how to reduce the problem of approximating the MAX-CLAUSE SSAT function on an instance ϕ' of one of the two types given in Theorem 3.3 to the problem of approximating the MAH-JONGG function on some instance. We first convert ϕ' into formula ϕ of Lemma 3.5 and then apply the following reduction to ϕ. We assume without loss of generality that the number of literals in each of the m clauses of ϕ is exactly three and that the number of variables, n, is even. Given such an instance ϕ, the main structures of the corresponding Mah-Jongg

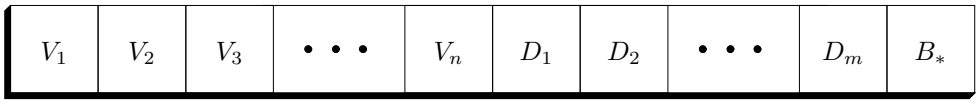


FIG. 6. *The regulating row.*

game are roughly as follows.

Corresponding to each variable is a set of *variable-setting rows*. Removing a particular tile from one of these rows corresponds to "setting" the variable. Corresponding to each clause is a set of *clause rows*. Most of the hidden tiles are in the clause rows. The *treasure chest* consists of a single row. On the left end of this row is a special *treasure-key* tile; the game can be won only by removing this tile. The matching treasure-key tile can only be accessed by uncovering a "good" subset of *guard* tiles, which are initially hidden. Thus the goal of a good strategy is to maximize the number of hidden tiles uncovered so as to maximize the chances of finding a good subset of guard tiles and hence of removing the treasure-key tile. This strategy corresponds to "setting" the existential variables so as to maximize the number of hidden tiles in the clause rows that are uncovered. (We will see that the random variables are "set" randomly.)

A *regulating row* of tiles is used to ensure that variables are set in the correct order and that no tiles can be removed from a clause row until all variables are set. Furthermore, the regulating row, together with the variable-setting rows, ensures that the only tiles in the clause rows that can legally be removed are those tiles in rows of clauses that have been satisfied during the variable-setting phase. Thus the probability of winning the game depends on the number of clauses that are satisfied by the variable assignment chosen in the variable-setting phase.

We now describe the tiles and their arrangement in detail. For each existential variable x_i, there are four tiles labeled V_i (one of which is in the treasure chest), and for each random variable x_i, there are two tiles labeled V_i. For each clause c_j, there are four tiles C_j and two tiles D_j. There are also b pairs of *blocking* tiles; again, b will be specified later. The kth pair of blocking tiles is labeled B_k. One of each pair is in the treasure chest.

Figure 6 describes the regulating row. Here and in what follows, we represent a blocking tile by B_*; the subscript is assumed to be different in each occurrence of B_*, and the position of a particular subscript is arbitrary. If the treasure chest is not open, in order to remove all tiles in regulating row, the V_i must first be matched in order, followed by the D_i in order.

We next describe the variable-setting structures. In addition to the tiles defined above, the variable-setting structures contain for each variable x_i a pair of tiles labeled $M_{i,k}$, where k ranges between 1 and the number of clauses in which either x_i or $\bar{x}_i$ occurs.

If x_i is existential, the ith variable-setting structure is described in Figure 7. If at some point in the game the treasure chest is not open and V_i is the leftmost tile in the regulating row, then this tile must be matched to the V_i in either the first or the second row of this structure. If it is matched to the V_i in the first row, this is equivalent to setting x_i to true. By removing the M-tiles in this row, the player can make available the tiles C_j in this structure for which clause c_j contains literal x_i. Similarly, setting x_i to false makes available the tiles C_j for which c_j contains $\bar{x}_i$.

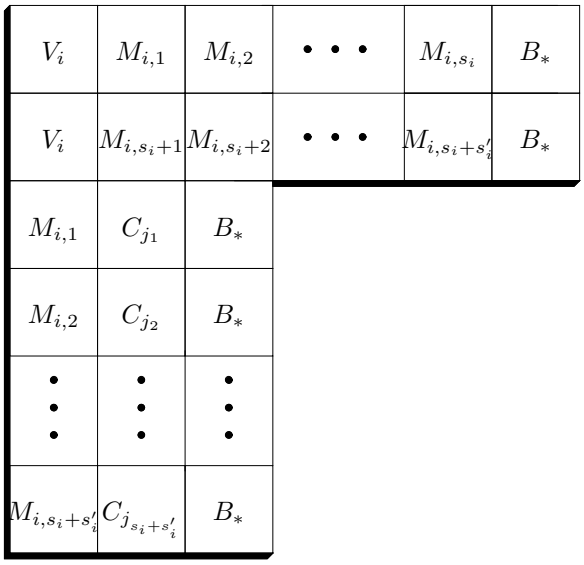


FIG. 7. *The variable-setting structure for existential variable i, where $j_1, j_2, \ldots, j_{s_i}$ are the clauses containing x_i and $j_{s_i+1}, \ldots, j_{s_i+s_i'}$ are the clauses containing $\bar{x}_i$.*

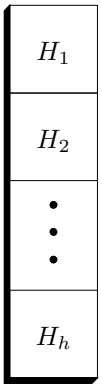


FIG. 8. *The remaining tiles.*

Since the fourth V_i tile is in the treasure chest, the V_i that is not matched cannot be removed until the treasure chest is opened.

If the variable x_i is a random variable, the variable-setting structure is quite similar, except that *hidden* tiles are used to ensure that the setting of x_i is randomly chosen. Each pair of hidden tiles is labeled H_k or H_k', for k in the range 1 to h (where h will be specified later). One tile from each pair is hidden. Each remaining tile H_k' is in the treasure chest and each remaining tile H_k forms a singleton row (see Figure 8). Then if x_i is random, the ith variable-setting structure is described in Figure 9. If the hidden tile under V_i is H_*, then it can be matched to a tile in a singleton row. We will see that this happens with probability approximately 1/2. As before, if x_i is contained in clause c_j, then tile C_j in this structure can be made available. Recall that $\bar{x}_i$ never appears if x_i is a random variable.

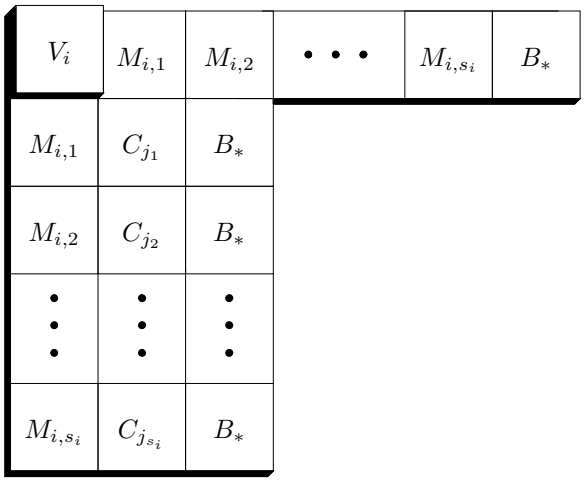


FIG. 9. *The variable-setting structure for random variable i, where $j_1, j_2, \ldots, j_{s_i}$ are the clauses in which x_i appears.*

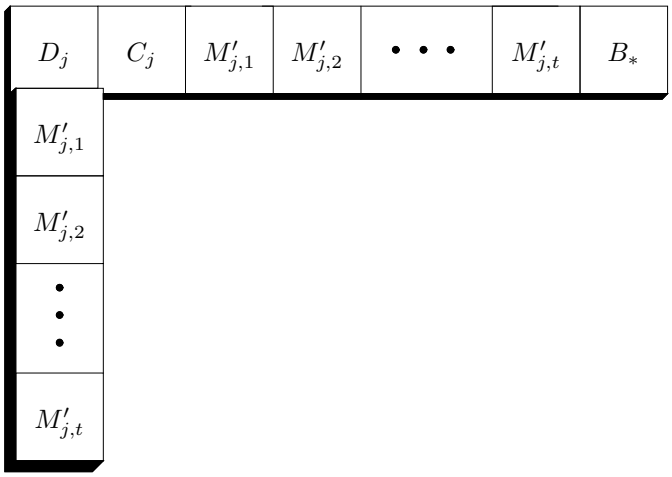


FIG. 10. *The clause row for the ith clause.*

We next describe the clause rows (see Figure 10). Again, we need additional tiles. For each clause j, there is a pair of tiles labeled $M'_{j,k}$, where k ranges between 1 and t (where t will be chosen later). There is a tile hidden under each M'-tile. This arrangement ensures that, if the jth clause is true, then all t of the hidden tiles under all $M'_{j,k}$ can be removed.

Before describing the treasure chest and the arrangement of guard tiles, we state some properties of the game structures constructed so far. In what follows, by "with high probability," we mean with probability at least $1 - 1/p(|\phi'|)$, for some large polynomial p. First, if the number $t = t(|\phi'|)$ of tiles hidden in each clause component is sufficiently large, we can ensure that, with high probability, only tiles of type H_k or H'_k are uncovered in the variable-setting phase. Second, assuming that the random variables are true and false with probability 1/2, it follows from Lemma 3.5 that,

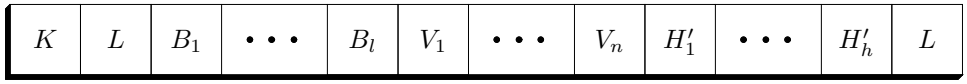


FIG. 11. *The treasure chest, where* $l = 4m + (3n/2) + 1 + (3v - 1)/2$.

if $q(|\phi'|)$ is sufficiently large, then with high probability, the fraction of hidden tiles uncovered in the clause components is either $\geq c_1$ or $\leq c_2$, depending on the type of ϕ'. The difficulty here is that after several hidden tiles have been revealed, the probability of a random variable being true or false is no longer $1/2$ but depends on the relative numbers of H_i and H'_i already revealed. However, if t is sufficiently large, say $t \geq r^k$, the chance of any particular variable being true will change by a factor of at most $1 \pm k!/r^{k-1}$. Thus the probability of any particular assignment of trues and falses to the random variables will be changed by a factor of at most $(1 \pm k!/r^{k-1})^r \approx 1 \pm k!/r^{k-2}$, which won't affect the probabilities enough to change the result. (This is the reason for the tiles D_i: they ensure that the hidden tiles associated with true clauses cannot be revealed, thus changing the probability of a variable being true, until after the variables have been set.)

The treasure chest is described in Figure 11. On the left end is the treasure-key K. Rows called *guard rows* control access to the single key K that matches the treasure-key in the following way. There are v guard rows (where v is a power of 3 and will be chosen later). Associated with the kth guard row are w pairs of guard tiles $G_{kj}, 1 \leq j \leq w$, one of which is initially hidden. The other guard tile from each pair is in the guard row. By finding all guard tiles for one guard row, the treasure-key can be accessed and the treasure chest opened. We describe the structure that enforces this later; we first show how such a structure, with appropriate choice of v and w, can guarantee that the reduction is "approximation preserving."

First, suppose that the expected fraction of simultaneously satisfiable clauses of ϕ' is at most $c_1 - \epsilon$ so that by Lemma 3.5, on any strategy for assigning values to the existential variables of ϕ, with probability at least $1 - 1/q(|\phi'|)$, the fraction of satisfied clauses of ϕ is at most c_1. Then, with high probability (at least $1 - 1/p(|\phi'|)$), on any play of the Mah-Jongg game, at most a fraction c_1 of the hidden tiles in the clause components are uncovered. To remove all guard tiles from a given guard row, all w guard tiles must be uncovered. If at most a fraction c_1 of the hidden tiles in the clause components are uncovered, the probability of finding these tiles in the clause components is at most c_1^w. Hence the probability that the treasure chest is not opened is at least the probability that no guard tiles are found during the variable-setting phase, times the probability that at most a fraction c_1 of the hidden tiles in the clause components are uncovered, times the probability that for all guard rows, it is not the case that all guard-keys in that guard row are uncovered. This is at least $(1 - 1/p(|\phi'|)^2(1 - c_1^w)^v)$.

In the case that the expected fraction of simultaneously satisfiable clauses of ϕ' is at least $c_2 + \epsilon$, a slightly different argument shows that the probability that the treasure chest is opened is at least $(1 - 1/p(|\phi'|)^2(1 - (1 - c_2^w)^v)$. We need to choose w and v so that $(1 - c_1^w)^v$ is large and $(1 - c_2^w)^v$ is small. We let $w = \log_{1/c_1} |\phi'|$ and let $v = |\phi'|$. Then $(1 - c_1^w)^v = (1 - 1/|\phi'|)^{|\phi'|} \approx 1/e$ and $(1 - c_2^w)^v \leq n^{-c}$, for some $c > 0$. This completes the proof that the reduction is approximation preserving.

It remains to describe the structure that ensures that the treasure-key can only be accessed by finding all guard tiles for one guard row. One way to ensure this in

G_{11}	• • •	G_{1w}	K_{01}	B_*
G_{21}	• • •	G_{2w}	K_{01}	B_*
G_{31}	• • •	G_{3w}	K_{01}	B_*
G_{41}	• • •	G_{4w}	K_{02}	B_*
G_{51}	• • •	G_{5w}	K_{02}	B_*
G_{61}	• • •	G_{6w}	K_{02}	B_*
⋮	⋮	⋮	⋮	⋮
$G_{v-2,1}$	• • •	$G_{v-2,w}$	K_{0i}	B_*
$G_{v-1,1}$	• • •	$G_{v-1,w}$	K_{0i}	B_*
G_{v1}	• • •	G_{vw}	K_{0i}	B_*

FIG. 12. *The guard rows.*

our construction would be to put a key K matching the treasure-key in each guard row, which could be accessed once all the guard tiles of the guard row are removed. However, this requires an unbounded number of treasure-keys. The following scheme achieves the same goal while ensuring that the number of tiles of any one type is at most 4 (see Figures 12 and 13).

The guard rows are arranged in groups of three; each row in the ith group has a tile K_{0i} to the right of all of its guard tiles; this is followed by a blocking tile B_*. For each group i, a fourth tile K_{0i} appears as the leftmost tile in a row with two other tiles. Again, these "first-level" rows are grouped in threes. Each row in the i'th group has a tile $K_{1i'}$ as its middle tile and a blocking B_*-tile as its rightmost tile. Again for each i', the fourth $K_{1i'}$ tile appears as the leftmost tile in a "second-level" row with two other tiles, and these rows are grouped in threes. Each row in the i''th group has a tile $K_{2i''}$ as its middle tile and a blocking B_*-tile as its rightmost tile. Further levels of rows are constructed in this way; the number of rows at each level of this structure decreases by a factor of three until there is only one row left. This row again has three tiles, but the middle one matches the treasure-key K. By opening any one of

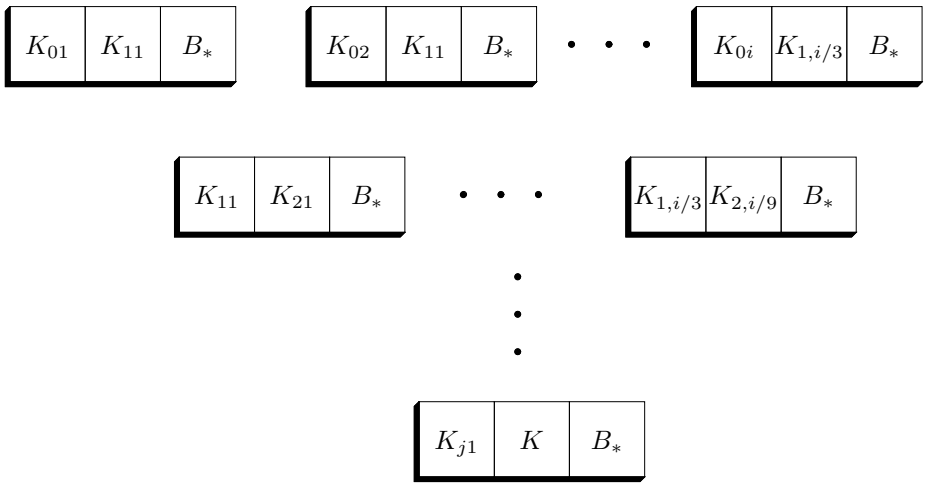


FIG. 13. *Levels of rows which guard access to the treasure chest. If all guard tiles from one guard row are removed, the key K_{0j} in that row can be used to remove keys at successive levels of this structure until the treasure-key K is removed at the last level.*

the guard rows, a row at each level can successively be opened until the treasure-key is finally accessed.

To account for the number of pairs of blocking tiles needed, note that $(3v-1)/2$ are needed for the guard rows, as well as $4m+3n/2$ for the clause and variable-setting rows and 1 for the regulating row. Therefore, the total number of blocking tiles needed is $4m+3n/2+1+(3v-1)/2$. Also, once t, v, and w are selected to ensure correctness of the reduction, h, the number of pairs of initially hidden tiles of the form H_k or H'_k, for some k, can be determined as follows. The total number of hidden tiles is $2h+vw$, that is, one tile from each pair of the form H_k or H'_k, and one of each of the vw pairs of guard tiles. Also, the number of locations in which tiles are hidden initially is $r+mt$, that is, one in each variable-setting structure for the r random variables and t in each of the m clause structures. Hence h is chosen so that $2h+vw = r+mt$. □

4. Nonapproximability results using IP. The proofs of all of the nonapproximability results in the previous section were based on the characterization RPCD$(\log n, 1)$ = PSPACE. In this section, we prove nonapproximability results for different PSPACE-hard functions based on the characterization IP = PSPACE obtained in [16, 22].

The first problem we consider here is also based on the language SSAT. Once again, an instance of SSAT can be thought of as a game between an existential player and a random player, but this time the objective of the existential player is to maximize the probability that ϕ is satisfied. The value of MAX-PROB SSAT on a given instance is the probability that ϕ is satisfied if the existential player follows an optimal strategy. The language SSAT, as defined in [21], consists of all instances ϕ for which MAX-PROB SSAT$(\phi) > 1/2$.

The reductions in this section are based on the following fact, which is a direct consequence of the proof that IP = PSPACE.

FACT 4.1. *For any language* L *in* PSPACE *and any $\epsilon < 1$, there is a polynomial-*

time reduction f from L *to* SSAT *such that*

$$x \in \mathrm{L} \Rightarrow \text{MAX-PROB SSAT}(f(x)) = 1 \quad \textit{and}$$
$$x \notin \mathrm{L} \Rightarrow \text{MAX-PROB SSAT}(f(x)) < 1/2^{n^{\epsilon}},$$

where n is the number of variables in $f(x)$.

The next theorem is a direct consequence of Fact 4.1.

THEOREM 4.2. *There is a constant $c > 0$ such that approximating* MAX-PROB SSAT *within ratio 2^{-n^c} is* PSPACE*-hard.*

Papadimitriou [21] defines the language Dynamic Markov Process (DMP). An instance is a set S of states and an $n \times n$ stochastic matrix P, where $n = |S|$. Associated with each state s_i is a set D_i of decisions, and each $d \in D_i$ is assigned a cost $c(d)$ and a matrix R_d. Each row of R_d must sum to 0, and each entry of $P + R_d$ must be nonnegative. The result of making decision d when the process is in state s_i is that a cost of $c(d)$ is incurred, and the probability of moving to state s_j is the (i, j)th entry of $P + R_d$. A strategy determines which decisions are made over time; an optimal strategy is one that minimizes the expected cost of getting from state s_1 to state s_n. The language DMP consists of tuples $(S, P, \{D_i\}, c, \{R_d\}, B)$ for which there is a strategy with expected cost at most B. A natural optimization problem is MIN DMP, the function that maps $(S, P, \{D_i\}, c, \{R_d\})$ to the expected cost of an optimal strategy. Papadimitriou [21] proves that DMP is PSPACE-complete by providing a reduction from SSAT. In fact, the reduction has the property that if SSAT instance ϕ is mapped to DMP instance $(S, P, \{D_i\}, c, \{R_d\}, B)$ and MAX-PROB SSAT(ϕ) $= p$, then MIN DMP$(S, P, \{D_i\}, c, \{R_d\}) = p/4$. The next result thus follows from Theorem 4.2.

THEOREM 4.3. *There is a constant $c > 0$ such that approximating* MIN DMP *within ratio 2^{-n^c} is* PSPACE*-hard.*

The complexity of coloring games was studied by Bodlaender [7], motivated by scheduling problems. An instance of a coloring game consists of a graph $G = (V, E)$, an ownership function o that specifies which of two players, 0 and 1, owns each vertex, a linear ordering f on the vertices, and a finite set C of colors. This instance specifies a game in which the players color the vertices in the order specified by the linear ordering. When vertex i is colored, its owner chooses a color from the set of *legal* colors, i.e., those in set C that are *not* colors of the colored neighbors of i. The game ends either when all vertices are colored or when a player cannot color the next vertex in the linear ordering f because there are no legal colors. Player 1 wins if and only if all vertices are colored at the end of the game. The length of the game is the number of colored vertices at the end of the game.

We consider a stochastic coloring game (SCG) in which one player, say 0, randomly chooses a color from the set of legal colors at each stage. Two corresponding optimization problems are to maximize the following functions: MAX-PROB SCG(G, o, f, C), which is the maximum probability that Player 1 wins the game (G, o, f, C), and MAX-LENGTH SCG(G, o, f, C), which is the maximum expected length of the game. (Both maxima are taken over all strategies of Player 1.)

We show that MAX-PROB SCG is PSPACE-hard to approximate within ratio 2^{-n^c}, for some constant $c > 0$, and that MAX-LENGTH SCG is PSPACE-hard to approximate within ratio $n^{-c'}$, for some constant $c' > 0$.

THEOREM 4.4. *There is a constant $c > 0$ such that approximating* MAX-PROB SCG *within ratio 2^{-n^c} is* PSPACE*-hard.*

Proof. We describe a reduction from MAX-PROB SSAT to MAX-PROB SCG that adapts the original construction [7]. By Fact 4.1, we can restrict our attention

to instances ϕ such that either MAX-PROB SSAT(ϕ) = 1 or MAX-PROB SSAT(ϕ) $< 1/2^{n^\epsilon}$, for some $\epsilon > 0$. We construct an instance $G = (V, E)$ of MAX-PROB SCG as follows:

$$V = \{\mathit{true}, \mathit{false}, X\} \cup \{x_i, \bar{x}_i \mid 1 \le i \le n\}$$

$$\cup \{c_{j,k} \mid 1 \le j \le m, 1 \le k \le n\} \cup \{d\}.$$

The linear ordering f of the vertices is as follows:

$$\mathit{true}, \mathit{false}, X, x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n\bar{x}_n, c_{1,1}, c_{1,2}, \ldots, c_{m,n}, d.$$

The ownership function is specified as follows. Player 1 owns the vertices *true*, *false*, and X and also the vertices x_i and $\bar{x}_i$, where x_i is existentially quantified in the formula ϕ. Player 0 owns the remaining vertices. The set of colors C is {*true*, *false*, X}.

It remains to specify the set of edges.

$$E = \{\{\mathit{true}, \mathit{false}\}, \{\mathit{true}, X\}, \{\mathit{false}, X\}\} \tag{1}$$

$$\cup \{\{x_i, \bar{x}_i\}, \{X, x_i\}, \{X, \bar{x}_i\} \mid 1 \le i \le n\} \tag{2}$$

$$\cup \{\{l_i, c_{j,k}\}, \{\mathit{false}, c_{j,k}\} \mid l_i \text{ is a literal in } c_j, 1 \le i \le n, 1 \le j \le m, 1 \le k \le n\} \tag{3}$$

$$\cup \{\{c_{j,k}, d\} \mid 1 \le j \le m, 1 \le k \le n\} \cup \{\{d, \mathit{false}\}, \{d, X\}\}. \tag{4}$$

The set of edges (1) ensures that vertices *true*, *false*, and X are colored with three distinct colors. Without loss of generality, suppose that they are colored *true*, *false*, and X, respectively.

The set of edges (2) ensures that for each pair $x_i, \bar{x}_i$, one is colored *true* and the other *false*. Thus each coloring of these vertices corresponds to a truth assignment of the variables $x_1, \ldots, x_n$.

The set of edges (3) ensures that if clause c_j is true with respect to the truth assignment corresponding to the coloring of vertices $x_1, \ldots, x_n$, then each vertex $c_{j,k}$ is colored X. If c_j is false, then each vertex $c_{j,k}$ is independently colored X or true, each with probability 1/2.

The set of edges (4) ensures that d can be colored only if all the $c_{j,k}$ are colored X.

We claim that deciding whether MAX-PROB SAT(ϕ) is equal to 1 or is at most $1/2^{n^\epsilon}$ can be reduced in polynomial time to approximating MAX-PROB SCG(G, o, f, C) within ratio 2^{-n^c}, where c is a positive constant that depends on ϵ. To see this, note that if MAX-PROB SSAT(ϕ) = 1, then there is a way to choose a truth assignment to the existentially quantified variables that ensures that all clauses of ϕ are true. Hence by definition of the ownership function, Player 1 has a strategy for coloring vertices such that for every choice of colors of Player 0, the corresponding truth assignment to $x_1, \ldots, x_n$ satisfies all clauses. Hence all vertices $c_{j,k}$ are colored X, and so vertex d can be colored. Thus Player 1 has a strategy that wins with probability 1.

On the other hand, if MAX-PROB SSAT(ϕ) $< 1/2^{n^\epsilon}$, then on any strategy of Player 1, d can be colored with only exponentially small probability. This is because, on all strategies of Player 1, with probability at least $1 - 1/2^{n^\epsilon}$, the truth assignment corresponding to the variable coloring fails to satisfy $f(x)$. Suppose that clause c_j is false. Then with probability at least $1 - 1/2^n$, one of the vertices $c_{j,k}, 1 \le k \le n$, is colored *true*. As a result, d can be colored with probability at most $1 - (1 - 1/2^n)(1 - 1/2^{n^\epsilon})$. □

The proof of Theorem 4.4 can easily be modified to show the following.

THEOREM 4.5. *There is a constant $c' > 0$ such that approximating* MAX-LENGTH SCG *within ratio $n^{-c'}$ is* PSPACE*-hard.*

Proof. Simply modify the construction of Theorem 4.4 so that the vertex set is V', consisting of the vertices of V plus $|V|n^{2c'}$ additional vertices $d_1, \ldots, d_{|V|n^{2c'}}$. The ownership of these vertices can be specified arbitrarily, and they are ordered after the vertices in V. No additional edges are needed.

The expected length of the game is now either $|V| - 1 + |V|n^{2c'}$ or $|V| - 1 + (1 - (1 - 1/2^n)(1 - 1/2^{n^\epsilon}))(|V|n^{2c'})$, depending on the two possibilities for the probability that ϕ is satisfied. □

5. Open problems. There are many other two-player games that can be made into stochastic functions by letting Player 0 play randomly. Examples include the following.

1. In the Node Kayles game of Schaefer [22], the input is a graph. A move consists of putting a marker on an unoccupied vertex that is not adjacent to any occupied vertex. The first player unable to move loses. We can define Stochastic Node Kayles (SNK) in the same way, except that Player 0, rather than choosing optimally among all unoccupied vertices that are not adjacent to occupied vertices, instead chooses uniformly at random from the same set. Player 1's objective is to keep the game going as long as possible. MAX SNK is the expected length of the game under an optimal strategy of Player 1.

2. In the Generalized Hex game of Even and Tarjan [12], the input is a graph with two distinguished nodes n_1 and n_2. A move for Player 1 (0) consists of putting a white (black) marker on a vertex; the player is free to choose any unoccupied vertex except n_1 or n_2. After all vertices have been chosen, Player 1 wins if and only if there is a path from n_1 to n_2 along only white-occupied vertices. Stochastic Generalized Hex (SGH) is, as usual, the same game in which Player 0 places a black marker on a random unoccupied vertex rather than an optimal unoccupied vertex. A natural stochastic function is MAX-PROB SGH, which maps a graph to the probability, under an optimal strategy of Player 1, that there will be a white path from n_1 to n_2.

Superficially, these games differ from games like Generalized Geography and Stochastic Coloring in that there is no "locality" requirement on the moves of the random player. In Stochastic Generalized Geography, the random player must choose from among the arcs out of the current vertex, and in SCG the random player must color the vertex specified by the ownership function. The reductions that we use to prove MAX SGGEOG, MAX-PROB SCG, and MAX-LENGTH SCG hard to approximate make essential use of this locality. In Node Kayles, say, the random player may choose to mark a vertex that is very far away from the vertex just marked by the existential player.

We have not proven nonapproximability results for any functions without locality. Are they easy to approximate within some constant ratio? Are they in fact easy to compute exactly? It would be interesting to settle the difficulty of approximating

these functions and, more generally, to characterize precisely those PSPACE-complete languages that give rise to stochastic functions that are hard to approximate.

Acknowledgments. We thank the anonymous referees for their very helpful comments, in particular, for pointing out that our bounds on the nonapproximability of SSGEOG and MAH-JONGG could be improved.

REFERENCES

[1] M. Ajtai and M. Ben-Or, *A theorem on probabilistic constant depth circuits*, in Proc. 16th Annual ACM Symposium on Theory of Computing, ACM, New York, 1984, pp. 471–474.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation problems*, in Proc. 33rd Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 14–23.

[3] S. Arora and M. Safra, *Probabilistic checking of proofs*, in Proc. 33rd Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 2–13; final version, J. Assoc. Comput. Mach., to appear.

[4] L. Babai and S. Moran, *Arthur–Merlin games: A randomized proof system and a hierarchy of complexity classes*, J. Comput. System Sci., 36 (1988), pp. 254–276.

[5] M. Bellare, S. Goldwasser, C. Lund, and A. Russell, *Efficient probabilistic checkable proofs and applications to approximation*, in Proc. 25th Symposium on Theory of Computing, ACM, New York, 1993, pp. 286–293.

[6] M. Bellare and M. Sudan, *Improved non-approximability results*, in Proc. 26th Symposium on Theory of Computing, ACM, New York, 1994, pp. 184–193.

[7] H. L. Bodlaender, *On the complexity of some coloring games*, Internat. J. Found. Comput. Sci., 2 (1991), pp. 133–147.

[8] R. B. Boppana and M. Sipser, *The complexity of finite functions*, in Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, J. van Leeuwen, ed., MIT Press/Elsevier, Cambridge, MA, New York, 1990, pp. 757–800.

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

[10] A. Condon, J. Feigenbaum, C. Lund, and P. Shor, *Probabilistically checkable debate systems and nonapproximability of PSPACE-hard functions*, Chicago J. Theoret. Comput. Sci., 1995, No. 4; extended abstract, *Probabilistically checkable debate systems and approximation algorithms for PSPACE-hard functions*, in Proc. 25th Symposium on Theory of Computing, ACM, New York, 1993, pp. 305–314.

[11] A. Condon, J. Feigenbaum, C. Lund, and P. Shor, *Random debaters and the hardness of approximating stochastic functions (extended abstract)*, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, 1993.

[12] S. Even and R. Tarjan, *A combinatorial problem which is complete in polynomial space*, J. Assoc. Comput. Mach., 23 (1976), pp. 710–719.

[13] U. Feige, S. Goldwasser, L. Lovász, M. Safra, and M. Szegedy, *Interactive proofs and the hardness of approximating cliques*, J. Assoc. Comput. Mach., 43 (1996), pp. 268–292.

[14] H. Hunt III, M. Marathe, and R. Stearns, *Generalized CNF satisfiability problems and non-efficient approximability*, in Proc. 9th Conference on Structure in Complexity Theory, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 356–366.

[15] D. S. Johnson, *A catalog of complexity classes*, in Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, J. van Leeuwen, ed., MIT Press/Elsevier, Cambridge, MA, New York, 1990, pp. 67–162.

[16] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic methods for interactive proof systems*, J. Assoc. Comput. Mach., 39 (1992), pp. 859–868.

[17] C. Lund and M. Yannakakis, *On the hardness of approximating minimization problems*, J. Assoc. Comput. Mach., 41 (1994), pp. 960–981.

[18] M. Marathe, H. Hunt III, and S. Ravi, *The complexity of approximating PSPACE-complete problems for hierarchical specifications*, Nordic J. Comput., 1 (1994), pp. 275–316.

[19] M. Marathe, H. Hunt III, R. Stearns, and V. Radhakrishnan, *Hierarchical specifications and polynomial-time approximation schemes for PSPACE-complete problems*, in Proc. 26th Symposium on Theory of Computing, ACM, New York, 1994, pp. 468–477; final version, *Approximation algorithms for PSPACE-hard hierarchically and periodically specified problems*, SIAM J. Comput., to appear.

[20] J. Orlin, *The complexity of dynamic languages and dynamic optimization problems*, Proc. 13th Symposium on Theory of Computing, ACM, New York, 1981, pp. 218–227.
[21] C. Papadimitriou, *Games against nature*, J. Comput. System Sci., 31 (1985), pp. 288–301.
[22] T. J. Schaefer, *On the complexity of some two-person perfect-information games*, J. Comput. System Sci., 16 (1978), pp. 185–225.
[23] A. Shamir, *IP = PSPACE*, J. Assoc. Comput. Mach., 39 (1992), pp. 869–877.
[24] M. Sudan, *Efficient checking of polynomials and proofs and the hardness of approximation problems*, Ph.D. thesis, Computer Science Division, University of California at Berkeley, Berkeley, CA, 1992.
[25] L. Valiant, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410–421.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 401–409, April 1997

005

A STRIP-PACKING ALGORITHM WITH ABSOLUTE PERFORMANCE BOUND 2*

A. STEINBERG†

Abstract. This paper proposes a new approximation algorithm M for packing rectangles into a strip with unit width and unbounded height so as to minimize the total height of the packing. It is shown that for any list L of rectangles, $M(L) \leq 2 \cdot \mathrm{OPT}(L)$, where $M(L)$ is the strip height actually used by the algorithm M when applied to L and $\mathrm{OPT}(L)$ is the minimum possible height within which the rectangles in L can be packed.

Key words. two-dimensional bin packing, strip packing, packing rectangles, absolute performance bound

AMS subject classifications. 68Q25, 90C27

PII. S0097539793255801

1. Introduction.

We consider the following two-dimensional packing problem first proposed in [1]: Given a vertical strip of width 1, bounded below but not above, and a list L of rectangular pieces $R_1, R_2, \ldots, R_\ell$, pack the pieces into the strip so that the height to which the strip is filled is as small as possible. The pieces are not allowed to overlap. We also assume that each piece R_i in L is defined by its width a_i and height b_i and must be packed in such a way that the edges corresponding to width (i.e., edges of length a_i) are parallel to the horizontal bottom edge of the strip (see Fig. 1). This type of packing is called orthogonal oriented.

Since the problem of finding an optimal packing is NP-hard [1], polynomial-time approximation algorithms which generate near-optimal packings are studied, in particular, from a performance-guarantee point of view [1], [2], [3], [4], [5], [6], [7].

For an arbitrary list $L = \{R_1, R_2, \ldots, R_\ell\}$ of rectangular pieces all having width no more than 1, let $\mathrm{OPT}(L)$ denote the minimum possible strip height within which the pieces $R_1, R_2, \ldots, R_\ell$ can be packed and let $A(L)$ denote the height actually used by a particular packing algorithm A when applied to L. If α is a constant such that for every L,

$$A(L) \leq \alpha \cdot \mathrm{OPT}(L), \tag{1.1}$$

then α is called an absolute performance bound for A. Various algorithms satisfying inequalities of such a type were described in [1], [3], [5] ($\alpha = 3$) and [3] ($\alpha = 2.7$). In [6], an algorithm was found for which (1) holds with $\alpha = 2.5$. In this paper, we present a new algorithm with an absolute performance bound of $\alpha = 2$ which packs the pieces into the strip in an almost linear (up to a logarithmic factor) time.

In fact, this algorithm, denoted by M, is intended for packing pieces $R_1, R_2, \ldots, R_\ell$ from L into a bounded rectangle Q. (We always assume that all rectangles are oriented in such a way that their edges corresponding to width are horizontal).

*Received by the editors September 20, 1993; accepted for publication (in revised form) May 15, 1995. This research was supported by the Center for Absorption in Science, Israel Ministry of Absorption, and by the Israel Ministry of Science and Technology.
http://www.siam.org/journals/sicomp/26-2/25580.html
†Department of Mathematics, Technion, Haifa 32000, Israel (orad@netvision.net.il).

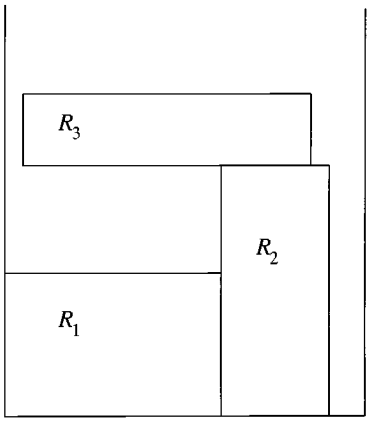


FIG. 1. *An orthogonal oriented strip packing of rectangular pieces* R_1 $(a_1 = 0.6, b_1 = 0.4)$, R_2 $(a_2 = 0.3, b_2 = 0.7)$, *and* R_3 $(a_1 = 0.8, b_1 = 0.2)$.

Denote the width and the height of Q by u and v, respectively, and let

$$a_L = \max_{1\le i\le \ell} a_i, \qquad b_L = \max_{1\le i\le \ell} b_i,$$
$$s_i = a_i b_i \quad (1 \le i \le \ell),$$
$$S_L = \sum_{i=1}^{\ell} s_i.$$

Our main result is the following.

THEOREM 1.1. *If the following inequalities hold,*

$$a_L \le u, \qquad b_L \le v, \qquad 2S_L \le uv - (2a_L - u)_+(2b_L - v)_+ \tag{1.2}$$

then it is possible to pack the rectangles $R_1, R_2, \ldots, R_\ell$ *into the rectangle* Q *by the algorithm* M. (*As usual,* $x_+ = \max(x, 0)$.)

Let us denote the problem of packing all the pieces of L into Q by (Q, L). We will say that (Q, L) is a tractable problem if L is nonempty and (2) is fulfilled.

Our plan is to show how every tractable problem can be reduced to tractable problems with a smaller list of pieces.

2. Reduction procedures. In this section, we define seven reduction procedures P_μ, $-3 \le \mu \le 3$. (For $\mu \ne 0$, the procedure $\mathrm{P}_{-\mu}$ is obtained from the procedure P_μ by interchanging horizontal and vertical directions.) Each P_μ can be applied only to tractable problems satisfying some additional condition C_μ. We combine the description of the procedures P_μ with the proof of the following lemma.

LEMMA 2.1. *Let a tractable problem* (Q, L) *satisfy* C_μ. *If* $|\mu| \le 2$, *then* P_μ *either solves* (Q, L) *or packs a proper part of* L *into* Q, *and for the remaining part* L' *of* L, *it constructs a tractable problem* (Q', L') *with* Q' *lying within the unoccupied area of* Q. *If* $|\mu| = 3$, *then* P_μ *divides* L *into two proper sublists* L' *and* L'' *and divides* Q *into two nonoverlapping rectangles* Q' *and* Q'' *in such a way that the problems* (Q', L') *and* (Q'', L'') *are tractable.*

Of course, we also need the following result.

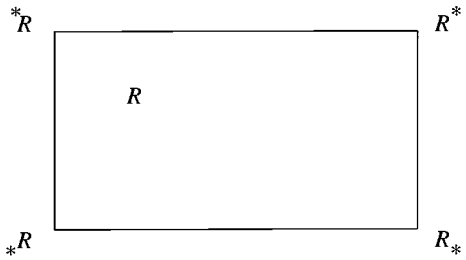


FIG. 2. *Vertices R_*, R^*, ${}^*R$, and ${}_*R$ of a rectangle R.*

LEMMA 2.2. *Every tractable problem satisfies at least one of the conditions* C_μ, $|\mu| \le 3$.

Notation. We will mark the vertices of any rectangle R by R_*, R^*, ${}^*R$, and ${}_*R$ as shown in Fig. 2.

Let (Q, L) be a tractable problem. As before, u and v denote the width and the height of Q. In order to define Procedures P_1 and P_3, it is convenient to assume that the list $L = \{R_1, R_2, \dots, R_\ell\}$ is ordered by decreasing width: $a_1 \ge a_2 \ge \cdots \ge a_\ell$.

PROCEDURE P_1. This procedure can be applied to the problem (Q, L) if the following condition holds:

$$(\mathrm{C}_1) \qquad a_L \ge \frac{1}{2}u.$$

Let m be the maximal index ($1 \le m \le \ell$) such that $a_m \ge \frac{1}{2}u$. Place the pieces $R_1, R_2, \dots, R_m$ so that

$${}_*[R_1] = {}_*Q, \qquad {}_*[R_i] = {}^*[R_{i-1}] \quad (2 \le i \le m)$$

(see Fig. 3). The third inequality in (2) implies that $S_L \le \frac{1}{2}uv$. Hence $\sum_{i=1}^m b_i \le v$ and the pieces $R_1, R_2, \dots, R_m$ lie inside Q. In particular, Procedure P_1 solves the problem (Q, L) if $m = \ell$.

Suppose that $m < \ell$. Denote $v - \sum_{i=1}^m b_i$ by v'. It is clear that $v' > 0$. Let $R_{t_1}, R_{t_2}, \dots, R_{t_{\ell-m}}$ be the list $R_{m+1}, R_{m+2}, \dots, R_\ell$ ordered by decreasing height:

$$b_{t_1} \ge b_{t_2} \ge \cdots \ge b_{t_{\ell-m}}.$$

If $b_{t_1} \le v'$, we form a problem (Q', L'), where

$$L' = \{R_{m+1}, R_{m+2}, \dots, R_\ell\}$$

and the rectangle Q' is defined by

$${}_*[Q'] = {}^*[R_m], \qquad [Q']^* = Q^*.$$

Since

$$2S_{L'} = 2S_L - 2\sum_{i=1}^m s_i \le uv - 2 \cdot \frac{1}{2}u(v - v') = uv'$$

and $a_{L'} < \frac{1}{2}$, we conclude that the new problem (Q', L') is tractable.

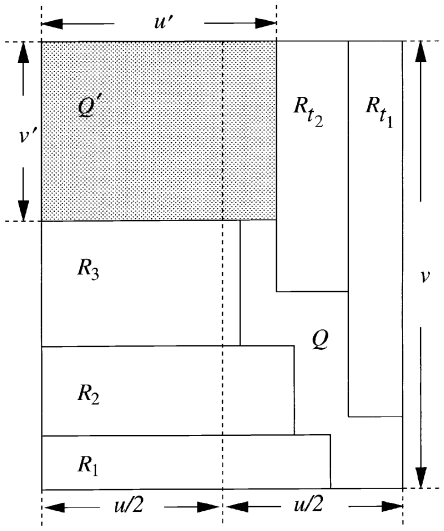


FIG. 3. *Illustration for Procedure* P_1 $(m = 3, n = 2)$.

In the case where $b_{t_1} > v'$, denote by n the maximal index $(1 \le n \le \ell - m)$ for which $b_{t_n} > v'$ and place $R_{t_1}, R_{t_2}, \ldots, R_{t_n}$ in the following way:

$$[R_{t_1}]^* = Q^*, \qquad [R_{t_j}]^* = {}^*[R_{t_{j-1}}] \quad (2 \le j \le n)$$

(see Fig. 3). We claim that the pieces R_i and R_{t_j} do not overlap for $1 \le i \le m$ and $1 \le j \le n$. Indeed if R_i and R_{t_j} overlap, then the pieces $R_1, R_2, \ldots, R_m$ and R_{t_1}, $R_{t_2}, \ldots, R_{t_n}$ cover the rectangles Q_1 and Q_2 defined by

$${}_*[Q_1] = {}_*Q, \qquad [Q_1]^* = [R_i]^* = {}_*[Q_2], \qquad [Q_2]^* = Q^*.$$

Therefore the area of $Q_1 \cup Q_2$ is less than S_L, i.e.,

$$a_i(v - d) + (u - a_i)d < S_L,$$

where $d = v - b_1 - b_2 - \cdots - b_i$. This inequality can be written in the following form:

$$uv - (2a_i - u)(2d - v) < 2S_L.$$

Since $0 \le 2a_i - u \le 2a_L - u = (2a_L - u)_+$ and $d \le b_{t_j} \le b_L$, we obtain

$$uv - (2a_L - u)_+(2b_L - v)_+ < 2S_L,$$

contradicting the fact that (Q, L) is a tractable problem. Note that the inequalities $b_{t_1} \ge b_{t_2} \ge \cdots \ge b_{t_n} > v'$ imply that all of the pieces $R_{t_1}, R_{t_2}, \ldots, R_{t_n}$ lie inside Q. Hence Procedure P_1 solves (Q, L) if $n = \ell - m$.

In the case where $n < \ell - m$, we form a new problem (Q', L'), where

$$L' = \{R_{t_{n+1}}, R_{t_{n+2}}, \ldots, R_{t_{\ell-m}}\}$$

and Q' is the rectangle satisfying

$${}_*[Q'] = {}^*[R_m], \qquad [Q']^* = {}^*[R_{t_n}].$$

Finally, the problem (Q', L') is tractable because $a_{L'} \le a_{m+1} < \frac{1}{2}u \le a_m \le u', b_{L'} = b_{t_{n+1}} \le v'$, and

$$\begin{aligned} 2S_{L'} &= 2S_L - 2\sum_{i=1}^{m} s_i - 2\sum_{j=1}^{n} s_{t_j} \\ &\le uv - 2\cdot\frac{1}{2}u(v-v') - 2(u-u')v' = u'v' - (u-u')v' \\ &\le u'v' - (2a_{L'} - u')_+ (2b_{L'} - v')_+ . \end{aligned}$$

Here u' and v' are the width and the height of Q'.

PROCEDURE P_3. This procedure can be applied to the problem (Q, L) if the following condition holds:

$$(C_3) \qquad a_L \le \frac{1}{2}u, \qquad b_L \le \frac{1}{2}v, \qquad \ell > 1,$$

and

$$S_L - \frac{1}{4}uv \le \sum_{i=1}^{m} s_i \le \frac{3}{8}uv, \qquad a_{m+1} \le \frac{1}{4}u$$

for some index m, $1 \le m < \ell$.

Set

$$Z = \sum_{i=1}^{m} s_i, \qquad u' = \max\left(\frac{1}{2}u, \frac{2Z}{v}\right), \qquad u'' = \min\left(\frac{1}{2}u, u - \frac{2Z}{v}\right).$$

Bearing in mind that $u' + u'' = u$, we cut Q into two rectangles Q' and Q'' with widths u' and u'' and the same height v and we form two problems (Q', L') and (Q'', L'') with $L' = \{R_1, R_2, \dots, R_m\}$ and $L'' = \{R_{m+1}, R_{m+2}, \dots, R_\ell\}$. Since $u' \ge \frac{1}{2}u$, $u'' \ge \frac{1}{4}u$, and $S_L - \frac{1}{2}u''v \le Z \le \frac{1}{2}u'v$, these problems are tractable.

PROCEDURE P_2. This procedure can be applied to the problem (Q, L) if the following condition holds:

$$(C_2) \qquad a_L \le \frac{1}{2}u, \qquad b_L \le \frac{1}{2}v,$$

and there exist two different indices i and $k (1 \le i, k \le \ell)$ such that

$$a_i \ge \frac{1}{4}u, \qquad a_k \ge \frac{1}{4}u, \qquad b_i \ge \frac{1}{4}v, \qquad b_k \ge \frac{1}{4}v,$$

$$2(S_L - s_i - s_k) \le (u - \max(a_i, a_k))v.$$

Assuming that $a_i \ge a_k$, we place R_i and R_k so that

$${}_*[R_i] = {}_*[Q], \qquad {}_*[R_k] = {}^*[R_i].$$

Evidently, R_i and R_k lie inside Q. If $\ell = 2$, the problem (Q, L) is solved. If $\ell > 2$, we form a new problem (Q', L'), where L' is obtained from L by deleting R_i and R_k and Q' is the rectangle such that

$${}_*[Q'] = [R_i]_*, \qquad [Q']^* = Q^*.$$

It is easy to check that (Q', L') is a tractable problem.

In Procedures P_μ ($\mu = 1, 2, 3$), the horizontal and vertical directions play a nonsymmetrical role. Obviously, we can interchange these directions and obtain "transposed" procedures, denoted by $\mathrm{P}_{-\mu}$. Procedure $\mathrm{P}_{-\mu}$ can be applied to tractable problems satisfying condition $(\mathrm{C}_{-\mu})$, "transposed" to (C_μ).

PROCEDURE P_0. This procedure can be applied to the problem (Q, L) if the following condition holds:

$$(\mathrm{C}_0) \qquad a_L \le \frac{1}{2}u, \qquad b_L \le \frac{1}{2}v, \quad \text{and} \quad S_L - \frac{1}{4}uv \le s_i$$

for some i, $1 \le i \le \ell$.

In this case, we place the piece R_i so that

$$_*[R_i] = {}_*Q.$$

If $\ell = 1$, the problem (Q, L) is solved. If $\ell > 1$, we form a new problem (Q', L'), where the rectangle Q' is defined by

$$_*[Q'] = [R_i]_*, \qquad [Q']^* = Q^*$$

and L' is obtained from L by deleting R_i. A trivial verification shows that (Q', L') is a tractable problem.

The proof of Lemma 1 is complete. Our next step is to prove Lemma 2.

Suppose that a tractable problem (Q, L) does not satisfy any of the conditions (C_μ), $|\mu| \le 3$. Since it does not satisfy (C_1) and (C_{-1}), it follows that

$$a_L \le \frac{1}{2}u, \qquad b_L \le \frac{1}{2}v.$$

We retain the assumption that the list L is ordered by decreasing width:

$$a_1 \ge a_2 \ge \cdots \ge a_\ell.$$

Denote by m the minimal index ($1 \le m \le \ell$) such that

$$\sum_{i=1}^{m} s_i > S_L - \frac{1}{8}uv. \tag{2.1}$$

Obviously, $m > 1$ since otherwise (Q, L) satisfies (C_0). Note that

$$\sum_{i=1}^{m-1} s_i \le S_L - \frac{1}{8}uv \le \frac{1}{2}uv - \frac{1}{8}uv = \frac{3}{8}uv.$$

The assumption $a_m \le \frac{1}{4}u$ yields

$$\sum_{i=1}^{m-1} s_i > S_L - \frac{1}{8}uv - s_m \ge S_L - \frac{1}{8}uv - \frac{1}{4}u \cdot \frac{1}{2}v = S_L - \frac{1}{4}uv$$

and hence implies that (Q, L) satisfies (C_3). Therefore, we proved that (3) holds together with

$$a_m > \frac{1}{4}u.$$

Now order list L by decreasing height:

$$L = \{R_{t_1}, R_{t_2}, \ldots, R_{t_\ell}\}, \qquad b_{t_1} \geq b_{t_2} \geq \cdots \geq b_{t_\ell}.$$

By using the "transposed" argument, we see that

$$\sum_{j=1}^{n} s_{t_j} > S_L - \frac{1}{8}uv, \qquad b_{t_n} > \frac{1}{4}v$$

for some n, $1 \leq n \leq \ell$. Let $I = \{1, 2, \ldots, m\}$ and $J = \{t_1, t_2, \ldots, t_n\}$. We have

$$\sum_{k \in I \cap J} s_k = \sum_{i=1}^{m} s_i + \sum_{j=1}^{n} s_{t_j} - \sum_{k \in I \cup J} s_k > 2\left(S_L - \frac{1}{8}uv\right) - S_L = S_L - \frac{1}{4}uv.$$

The intersection $I \cap J$ cannot contain a single index because (Q, L) does not satisfy (C_0). Therefore, there exist two different indices i and k belonging to $I \cap J$. Since

$$a_i, a_k \geq a_m > \frac{1}{4}u, \qquad b_i, b_k \geq b_{t_n} > \frac{1}{4}v,$$

and (Q, L) does not satisfy conditions (C_2) and (C_{-2}), we have

$$2(S_L - s_i - s_k) > (u - \max(a_i, a_k))v,$$
$$2(S_L - s_i - s_k) > u(v - \max(b_i, b_k)).$$

These inequalities together with the inequality $2S_L \leq uv$ imply

$$s_i + s_k < \frac{v}{2}\max(a_i, a_k), \qquad s_i + s_k < \frac{u}{2}\max(b_i, b_k);$$

hence

$$s_i + s_k < \frac{v}{4}\max(a_i, a_k) + \frac{u}{4}\max(b_i, b_k).$$

On the other hand, we have

$$s_i + s_k = a_i b_i + a_k b_k \geq \max(a_i, a_k) \cdot \min(b_i, b_k) + \min(a_i, a_k) \cdot \max(b_i, b_k) > \frac{v}{4}\max(a_i, a_k) + \frac{u}{4}\max(b_i, b_k).$$

This contradiction completes the proof of Lemma 2.

Lemmas 1 and 2 suggest the following recursive algorithm M. For every tractable problem (Q, L), M uses an appropriate Procedure P_μ. The algorithm then stops if (Q, L) is solved; otherwise, M is applied to one or two tractable problems constructed by P_μ.

The algorithm M can be implemented to run in time $O\left((\ell \cdot \log^2 \ell)/\log\log\ell\right)$, where ℓ is the number of pieces in L. To see this, we equip L with the structure of three linked lists corresponding to decreasing order of width, height, and area, and we keep this structure for all sublists of L created by P_μ $(|\mu| \leq 3)$. Since for $|\mu| \leq 2$, the checking of (C_μ) has a constant cost, and since P_μ packs pieces into Q and deletes them from L in linear time, it suffices to show that the total time $T(\ell)$ used by P_3 and P_{-3} when M is applied to (Q, L) has the $O((\ell \cdot \log^2 \ell)/\log\log\ell)$ bound. We may

assume without loss of generality that M uses P_3 and P_{-3} if and only if it cannot use any other reduction procedure. Note that every tractable problem (Q, L) satisfying (C_3) or (C_{-3}) can be decomposed into two subproblems (Q', L') and (Q'', L'') (as in Lemma 1) in $O(\min\{\ell' \log \ell', \ell'' \log \ell'', \ell\})$ time, where ℓ' and ℓ'' are the number of pieces in L' and L'' $(\ell' + \ell'' = \ell, \ell' > 0, \ell'' > 0)$. Therefore,

$$T(\ell) \le T(\ell') + T(\ell'') + O(\min\{\ell' \log \ell', \ell'' \log \ell'', \ell\}).$$

This allows us to prove the desired bound by induction.

Let us consider M as an algorithm for the strip-packing problem. For this purpose, we cut from the strip the rectangle Q_h that has the bottom of the strip as a lowest edge and a minimal height h, for which (Q_h, L) is a tractable problem. This height h is determined by

$$h = \inf\{v : b_L \le v \quad \text{and} \quad 2S_L \le v - (2a_L - 1)_+(2b_L - v)_+\}.$$

It is easy to check that

$$h = \max\left\{b_L, 2S_L + \left(2 - \frac{1}{a_L}\right)_+ (b_L - S_L)_+\right\}.$$

Thus we proved the following result.

THEOREM 2.3. *Let L be a list of rectangular pieces such that $a_L \le 1$. Then*

$$M(L) \le \max\left\{b_L, 2S_L + \left(2 - \frac{1}{a_L}\right)_+ (b_L - S_L)_+\right\}. \tag{2.2}$$

In particular,

$$M(L) \le S_L + \max\{b_L, S_L\} \tag{2.3}$$

and

$$M(L) \le 2 \cdot \mathrm{OPT}(L).$$

The last inequality follows from (5) because $S_L \le \mathrm{OPT}(L)$ and $b_L \le \mathrm{OPT}(L)$.

The examples given in Fig. 4 show that estimate (5) is the best possible in the following sense: for every $b > 0$ and $S > 0$, the optimal packing height $\mathrm{OPT}(L)$ of a list L with $b_L = b$ and $S_L = S$ can be made as close as desired to $S + \max\{b, S\}$. In particular, as a corollary of Theorem 2, we have that

$$\sup\{\mathrm{OPT}(L) : a_L \le 1, \quad b_L = b, \quad S_L = S\} = S + \max\{b, S\}.$$

This suggests the following problem: for every $0 < a \le 1$, $b > 0$, and $S > 0$, find the supremum of $\mathrm{OPT}(L)$ over all lists L of rectangles for which $a_L = a, b_L = b$, and $S_L = S$. One can check that if $\frac{1}{2} < a \le 1$ or $S \le b$, then this supremum coincides with the right side of (4). In general, however, the problem is still open.

Acknowledgments. I would like to thank A. Lipovetsky for many helpful discussions. I also thank the referees for suggestions on the organization of this paper.

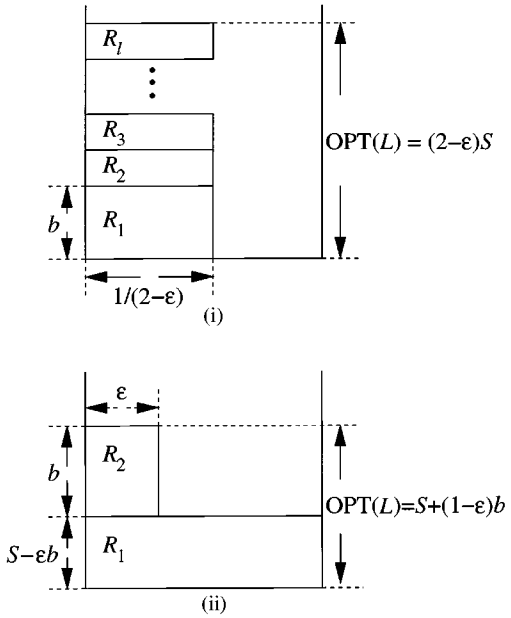


FIG. 4. *Examples of lists L with $b_L = b$, $S_L = S$, and* $\mathrm{OPT}(L) = S + (1-\varepsilon)\max\{b, S\}$ *($\varepsilon > 0$ is sufficiently small).* (i) *The case where $b < S : L = \{R_1, R_2, \ldots, R_\ell\}$, $\ell \geq (2-\varepsilon)S/b$. The pieces $R_2, R_3, \ldots, R_\ell$ have the same height $((2-\varepsilon)S-b)/(\ell-1)$.* (ii) *The case where $b \geq S : L = \{R_1, R_2\}$.*

REFERENCES

[1] B. S. BAKER, E. G. COFFMAN, JR., AND R. L. RIVEST, *Orthogonal packings in two dimensions*, SIAM J. Comput., 9 (1980), pp. 846–855.

[2] B. S. BAKER, D. J. BROWN, JR., AND H. P. KATSEFF, *A 5/4 algorithm for two-dimensional packing*, J. Algorithms, 2 (1981), pp. 348–368.

[3] E. G. COFFMAN, JR., M. R. GAREY, D. S. JOHNSON, AND R. E. TARJAN, *Performance bounds for level-oriented two-dimensional packing algorithms*, SIAM J. Comput., 9 (1980), pp. 808–826.

[4] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *Approximation algorithms for bin packing: An updated survey*, in Algorithm Design for Computer System Design, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer-Verlag, Berlin, New York, 1984, pp. 49–106.

[5] I. GOLAN, *Performance bounds for orthogonal oriented two-dimensional packing algorithms*, SIAM J. Comput., 10 (1981), pp. 571–582.

[6] D. SLEATOR, *A 2.5 times optimal algorithm for packing in two dimensions*, Inform. Process. Lett., 10 (1980), pp. 37–40.

[7] D. COPPERSMITH AND P. RAGHAVAN, *Multidimensional on-line bin packing: Algorithms and worst-case analysis*, Oper. Res. Lett., 8 (1989), pp. 17–20.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 410–417, April 1997

006

APPROXIMATING SHORTEST SUPERSTRINGS*

SHANG-HUA TENG† AND FRANCES F. YAO‡

Abstract. The shortest-superstring problem is to find a shortest possible string that contains every string in a given set as substrings. This problem has applications to data compression and DNA sequencing. Since the problem is NP-hard and MAX SNP-hard, approximation algorithms are of interest. We present a new algorithm which always finds a superstring that is at most 2.89 times as long as the shortest superstring. Our result improves the 3-approximation result of Blum et al.

Key words. approximation algorithms, combinatorial optimization, DNA sequencing, data compression, optimal assignments, the shortest-superstring problem

AMS subject classifications. 05C50, 68R10

PII. S0097539794286125

1. Introduction.

Let $S = \{s_1, \ldots, s_n\}$ be a set of strings over an alphabet Σ. A *superstring* of S is a string α which contains each s_i as a *substring*, i.e., there exist u_i and v_i such that α can be written as $u_i s_i v_i$. The *shortest-superstring problem* (SSP) is to find a minimum-length superstring for any given set S.

One obvious application for the shortest-superstring problem is data compression. Storer and Szymanski [14, 15], for example, considered a fairly general model of data compression which includes the SSP as an important special case. See also Mayne and James [9]. Another application is to DNA sequencing. The SSP is one of the simplest models for the problem of recovering DNA sequencing information from experimental data, and some heuristic algorithms have been in routine use for this task [2, 6, 7, 8, 12, 13].

Since the shortest-superstring problem is known to be NP-hard [4, 5], it is of interest to find approximation algorithms with good performance guarantees. Some heuristics have been considered by Tarhio and Ukkonen [16], Turner [18], and Blum et al. [2]. The simplest one is the algorithm Greedy: repeatedly merge a pair of strings with maximum overlap until only one string remains. It was shown that with respect to the *compression* (or *overlap*) measure, that is, the sum of the overlaps between consecutive strings in the candidate superstring, Greedy achieves at least 1/2 the compression of an optimal superstring [16, 18]. However, although a shortest superstring also achieves maximum compression, in general, a performance guarantee with respect to compression implies no performance guarantee with respect to length (see section 2). Indeed, it seemed quite hard to establish any linear approximation in the length measure. The first breakthrough was made by Blum et al. in [2], where they gave an intricate proof that Greedy achieves an approximation ratio of 4. Furthermore, a modified algorithm, called TGreedy, has an approximation ratio of 3. They also showed that the superstring problem is MAX SNP-hard [11]. In conjunction with the recent result of Arora et al. [1] that MAX SNP-hard problems do not have polynomial-time approximation schemes unless P = NP, it implies that

* Received by the editors August 25, 1994; accepted for publication (in revised form) May 17, 1995.

http://www.siam.org/journals/sicomp/26-6/28612.html

† Department of Mathematics and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (steng@math.mit.edu). The research of this author was supported in part by AFOSR grant F49620-92-J-0125 and DARPA grant N00014-92-J-1799. Part of this work was done while the author was at the Xerox Palo Alto Research Center.

‡ Xerox Corporation, Palo Alto Research Center, Palo Alto, CA 94304 (yao@parc.xerox.com).

a polynomial-time approximation scheme for this problem is unlikely. It was left as an open question in [2] whether there exist polynomial-time algorithms that achieve approximation ratios better than 3.

In this paper, we present a new polynomial-time approximation algorithm for this problem. The algorithm always finds a superstring whose length is at most 2.89 times the length of the shortest superstring. Thus it affirmatively answers the open question raised in [2].

Our construction makes use of some new structural properties of superstrings. These properties are interesting in their own right and may have other applications to problems on string matching. In section 2, we give an overview of our approach. Section 3 reviews some basic concepts of the superstring problem and the underlying graph representation. In section 4, we describe a new superstring algorithm and prove that it achieves an approximation ratio of 2.89.

2. Overview of our approach. We first give an informal description of the ideas behind our approximation algorithm. These ideas will be worked out analytically in sections 3 and 4. Precise definitions of the terminology used here can be found in the next section.

For any superstring α of S, let length_α and overlap_α denote the length and the overlap (compression) achieved by α, respectively. Then $\text{length}_\alpha + \text{overlap}_\alpha = |S|$, the total length of strings in S. Comparing α with the optimal superstring gives

$$\text{length}_\alpha - \text{length}_{\text{opt}} = \text{overlap}_{\text{opt}} - \text{overlap}_\alpha.$$

In general, $\text{overlap}_{\text{opt}}$ may grow quadratically in $\text{length}_{\text{opt}}$ (in the case that the input strings can be packed densely into a superstring). Hence a guarantee that α achieves, say, 50% of optimal compression does not translate into a constant-factor guarantee for its length. Therefore, a key to designing algorithms with linear length approximation is to construct suitable subproblems for which $\text{overlap}_{\text{opt}}$ is only linear in $\text{length}_{\text{opt}}$ (cf. Lemmas 3.5 and 4.4). We now outline our method below.

Since the shortest-superstring (or maximum-compression) problem corresponds to the maximum-Hamiltonian-path problem in a certain weighted digraph, our approximation scheme starts by computing, as in [2] and [18], an optimal assignment (or cycle cover) C in this graph. The remaining task, then, is to find a suitable linear extension of C, i.e., to merge the cycles into one long path P. Our approach for generating P is to first construct a "backbone" of P by taking one vertex from each cycle of C and finding a suitable path P' for this vertex set V'. We then design a procedure that inserts the remaining parts of C into the backbone structure P' without losing much of their original weights.

To achieve an approximation ratio of 3, one can simply construct P' from an optimal cycle cover C' for V' (or by the Greedy algorithm). To obtain further improvement, we take the above process one step further and build P' via a recursive construction, again making use of optimal cycle covers. However, the key here is to design a different procedure for insertion when it comes to 2-cycles in order to get better performance guarantees.

The complete algorithm, which admits a short description, is given in section 4.3. The design and analysis of our algorithm are accompanied by an investigation of the combinatorial structures of the SSP as well as the periodic properties of strings. Further discussions of our method are given in section 5.

3. Definitions and lemmas. As in previous studies, we may assume without loss of generality that S is *substring free*, i.e., no string $s_i \in S$ is a substring of $s_j \in S$

for $j \neq i$. Let $|s|$ denote the length of a string s, and let $|S| = \sum_{s \in S} |s|$. We use $opt(S)$ to denote the length of the shortest superstring of S.

For two strings s and t, let v be the longest string such that $s = uv$ and $t = vw$ for some nonempty strings u and w. We call $|v|$ the *overlap* between s and t, denoted by $ov(s,t)$. String u is called the *prefix* of s with respect to t and is written as $pref(s,t)$. The value $|pref(s,t)|$ is called the (prefix) *distance* from s to t, denoted by $d(s,t)$. The string $uvw = pref(s,t)t$ of length $d(s,t) + |t| = |s| + |t| - ov(st)$ is the shortest superstring of s and t in which s appears before t, and is called the *merge* of s and t. We will abbreviate $pref(s_i, s_j)$, $d(s_i, s_j)$, and $ov(s_i, s_j)$ to simply $pref(i,j)$, $d(i,j)$, and $ov(i,j)$, respectively.

Given a list of strings $s_1, s_2, \ldots, s_r$, we define the superstring $\langle s_1, s_2, \ldots, s_r \rangle$ to be the string $pref(1,2)pref(2,3)\ldots pref(r-1,r)s_r$. Thus it is the shortest superstring in which $s_1, s_2, \ldots, s_r$ appear in order. For any permutation π of $\{1, 2, \ldots, n\}$, let $S_\pi = \langle s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)} \rangle$. It is easy to see that the shortest superstring α for S must equal S_π for some π.

3.1. Distance graph and overlap graph. The shortest-superstring problem can be formulated either as a minimization problem with respect to prefix distance or as a maximization problem with respect to overlap. We will have occasion to refer to both of these measures in our discussions. First, define a graph G_S for $S = \{s_1, \ldots, s_n\}$ to be the complete digraph on the vertex set $\{1, 2, \ldots, n\}$. The *distance graph* (G_S, d) of S is the weighted digraph where the weight of an edge (i,j) is $d(i,j)$ for $i \neq j$. The *overlap graph* (G_S, ov) of S is also a weighted digraph with edge weights $ov(i,j)$ for $i \neq j$. We use $d(E)$ and $ov(E)$ to denote the total weight of an edge set E in (G_S, d) and in (G_S, ov), respectively. Note that if E is a cycle, then the sum $d(E) + ov(E)$ is equal to the total length of the strings in E.

Notice that $\mathrm{HMP} = |S| - opt(S)$, where HMP is the cost of a maximum Hamiltonian path in (G_S, ov). On the other hand, $\mathrm{TSP}(G_S) \leq opt(S)$, where $\mathrm{TSP}(G_S)$ is the cost of a minimum Hamiltonian cycle in (G_S, d). (See also [2, 18].) Since finding optimal Hamiltonian paths is a hard problem, both Turner and Blum et al. considered approximation schemes that are based on solving the related *assignment* problem, as we will discuss below.

Remark. We have defined G_S to be without self-loops, which is different from the definition adopted in [2]. By doing so, we have sacrificed the "Monge condition" which the graph (G_S, ov) would otherwise satisfy. This slows down the running time of our algorithm, but it enables us to design a more effective approximation.

3.2. Optimal assignment. An *assignment* (or *cycle cover*) of a digraph G is a collection of cycles such that every vertex of G is in exactly one cycle. An *optimal* assignment is an assignment whose total weight is minimized (or maximized). Since $d(C) = |S| - ov(C)$ for any assignment C on G_S, a minimum assignment for (G_S, d) corresponds to a maximum assignment for (G_S, ov). We will use the convention that minimum assignments are for (G_S, d) and maximum assignments are for (G_S, ov). The optimal-assignment problem can be solved in $O(n^3)$ time by the Hungarian method (see, e.g., [10] and [17]).

For a cycle c in an assignment C, we refer to $d(c)$ as the *weight* of c. For a vertex (string) $s_i \in c$, we also say that $d(c)$ is the weight of s_i. The total weight of all cycles in assignment C is $d(C)$, and the total overlap is $ov(C)$.

LEMMA 3.1. *Let C be an optimal assignment for S. Then $d(C) \leq \mathrm{TSP}(G_S) \leq opt(S)$, and $ov(C) \geq |S| - opt(S)$.*

LEMMA 3.2. *Let ov_2, ov_3, and ov_4 denote, respectively, the amount of overlap in all 2-cycles, 3-cycles, and cycles of size 4 or larger in an optimal assignment C for S. Let C' be a subset of C obtained by discarding the minimum-weight edge (with respect to ov) from each cycle in C; then $ov(C') \geq ov_2/2 + 2ov_3/3 + 3ov_4/4$.*

It follows that $ov(C') \geq ov(C)/2$. Thus the superstring corresponding to C' achieves at least half the overlap of an optimal superstring. The same can be said about the superstring produced by the Greedy algorithm, although the proof is not as straightforward (see [18]). We state this result for later use.

LEMMA 3.3. *The total overlap achieved by the Greedy algorithm is at least 1/2 of the overlap achieved by an optimal superstring.*

The structure of the optimal assignment for a string set S is closely related to the periodicity structure of the strings. These relations can be exploited in designing approximate-superstring algorithms. We will look at some of these relations.

3.3. Periodicity of strings. We say a string t is *irreducible* if all cyclic shifts of t yield distinct strings. It is easy to show that (see, e.g., [3]) every string s has a unique prefix t such that t is irreducible and $s = t^k$ for some $k \geq 1$. String t is called the *period* of s. Clearly, in a minimum assignment C for S, the string $t = pref(i_1, i_2)pref(i_2, i_3) \dots pref(i_r, i_1)$ associated with a cycle $c = (i_1, \dots, i_r) \in C$ must be an irreducible string. We refer to t (or any of its cyclic shifts) as the period of c, denoted by $period(c)$; thus $|period(c)| = d(c)$. The next lemma follows immediately from the definition of a cycle.

LEMMA 3.4. *Each string s in c is a substring of $(period(c))^k$ for sufficiently large k.*

The following lemma from [2] is central to the analysis of various superstring algorithms based on minimum assignments. (A strengthened version, given in Lemma 4.4, will be used to analyze our algorithm.)

LEMMA 3.5. *Let c_1 and c_2 be two cycles in a minimum-weight assignment C with $s_1 \in c_1$ and $s_2 \in c_2$. Then the overlap between s_1 and s_2 is less than $d(c_1) + d(c_2)$.*

We use the shorthand $\langle s_j, c \rangle$ for the superstring $\langle s_j, s_{j+1}, \dots, s_r, s_1 \dots, s_j \rangle$. That is, $\langle s_j, c \rangle$ is the result of merging the strings in order around cycle c, beginning and ending with s_j. Let C be an assignment for G_S. Suppose we pick an arbitrary string r_i from each cycle c_i to form a representative set R and construct a superstring α for R, say $\alpha = \langle r_1, \dots, r_t \rangle$. Now let $\bar{\alpha}$ be the string derived from α by "extending" each r_i by its period, i.e., by letting $\bar{\alpha} = \langle \langle r_1, c_1 \rangle, \dots, \langle r_t, c_t \rangle \rangle$. Note that $\bar{\alpha}$ is a superstring of S, and we will call $\bar{\alpha}$ the *extended string* of α (with respect to C).

LEMMA 3.6. *Let R be a representative set of an assignment C, and let α be a superstring of R. Then the extended string $\bar{\alpha}$ of α is a superstring of S and $|\bar{\alpha}| = |\alpha| + d(C)$.*

4. A new approximation algorithm. In this section, we present an algorithm for the superstring problem that has an approximation ratio of 2.89. There are several new ingredients in this approximation scheme. The first idea is to select, among all optimal assignments of G_S, a *canonical* one, as we shall define. This enables us to prove a strengthened version of Lemma 3.5. Another crucial step in our algorithm is to find an economical way to merge the 2-cycles in an optimal assignment. In this case, our method gives a better guarantee than retaining only half of the overlap, as stated in Lemmas 3.2 and 3.3.

4.1. Canonical optimal assignments. Let C be an minimum assignment of a string set S. For a string $s \in S$ and a cycle $c \in C$, if there exists a (sufficiently large)

positive integer k such that s is a substring of $(period(c))^k$, then we say that s *fits* c. Clearly, if $s \in c$, then s fits c (Lemma 3.4). We can restate Lemma 3.5 as follows.

LEMMA 4.1. *Let c_1 and c_2 be two cycles in C. If $s \in S$ fits both c_1 and c_2, then $|s| \leq d(c_1) + d(c_2)$.*

We say that s fits c *uniquely* if s fits c and any two occurrences of s as substrings in $(period(c))^k$ must be offset by one or more periods.

LEMMA 4.2. *Let c be a cycle in a minimum assignment of S. If $s \in S$ fits c and $|s| \geq d(c)$, then s fits c uniquely.*

Proof. The result follows immediately from the fact that $period(c)$ is an irreducible string. □

Let c and c' be two cycles of C. Suppose that string s belongs to c and that s also fits c'. Since removing s from c and reassigning s to c' does not increase the overall weight, it still yields a minimum assignment. We call a minimum assignment *canonical* if each string s is assigned to a cycle whose weight is the smallest among all cycles that s fits.

LEMMA 4.3 (canonical assignment). *Given a minimum assignment C for S, we can transform C into a canonical minimum assignment C' for S in $O(n|S|)$ time.*

Proof. Note that s fits c if and only if s is a prefix of $(period(c))^k$ for $k = \lceil |s|/|d(c)| \rceil + 1$. Thus in $O(n|S|)$ time, we can find for each $s \in S$ the set of all cycles that s fits and assign s to the one with the smallest weight. □

Thus a canonical minimum assignment of a string set can be found in polynomial time. For canonical minimum assignments, we can establish an extension of Lemma 3.5 by bounding the sum of a pair of symmetric overlaps $ov(r_1, r_2)$ and $ov(r_2, r_1)$. This lemma is useful for designing linear embeddings of 2-cycles.

LEMMA 4.4. *Let c_1 and c_2 be two cycles in a canonical minimum assignment C with $r_1 \in c_1$ and $r_2 \in c_2$. Then $ov(r_1, r_2) + ov(r_2, r_1) < \max(|r_1|, |r_2|) + \min(d(c_1), d(c_2))$.*

Proof. Denote the sum $ov(r_1, r_2) + ov(r_2, r_1)$ by L. Let h_1 be the amount of overlap between the two copies of r_1 in $\langle r_1, r_2, r_1 \rangle$. Similarly, let h_2 be the amount of overlap between the two copies of r_2 in $\langle r_2, r_1, r_2 \rangle$.

We have the following two cases:

1. Suppose that $h_1 < d(c_1)$ and $h_2 < d(c_2)$. Then we have $L = |r_1| + h_2 < |r_1| + d(c_2)$, and similarly $L = |r_2| + h_1 < |r_2| + d(c_1)$; the lemma is true.

2. Suppose that either $h_1 \geq d(c_1)$ or $h_2 \geq d(c_2)$. Without loss of generality, assume that $h_1 \geq d(c_1)$. By Lemma 4.2, in the superstring $\langle r_1, r_2, r_1 \rangle$, the two occurrences of r_1 must be offset by a multiple of $d(c_1)$; hence $h_1 \leq |r_1| - d(c_1)$. It also follows that r_2, being sandwiched between these r_1's, must fit c_1. Hence $d(c_1) \geq d(c_2)$ since C is a canonical optimal assignment. Also, $|r_2| < d(c_1) + d(c_2)$ by Lemma 3.5. Therefore, $L = |r_2| + h_1 < (d(c_1) + d(c_2)) + (|r_1| - d(c_1)) \leq |r_1| + d(c_2) \leq \max(|r_1|, |r_2|) + \min(d(c_1), d(c_2))$. □

Remark. It is worthwhile to point out that the notion of canonical optimal assignment is essential in the sense that Lemma 4.4 does not hold for all optimal assignments. In fact, the left-hand side of the inequality can be much larger than the right-hand side in general.

4.2. Long paths through 2-cycles. Let $F = \{f_1, \ldots, f_m\}$ and $G = \{g_1, \ldots, g_m\}$ be two lists of m strings. Form m 2-cycles $C_{F,G}$ by matching f_i with g_i, that is, let $C_{F,G} = \{\langle f_i, g_i, f_i \rangle, 1 \leq i \leq m\}$. Let $ov(F, G)$ be the total overlap in the resulting 2-cycles $\langle f_i, g_i, f_i \rangle$. Let η be a superstring of G; without loss of generality, assume that $\eta = \langle g_1, g_2, \ldots, g_m \rangle$. Let $ov(\eta)$ be the total overlap in η.

We now describe an algorithm Insert(F, η) which generates a superstring of $F \cup G$ by inserting F into η. This is the process we alluded to in section 2 for building a path on $F \cup G$ from a backbone structure η on G.

ALGORITHM Insert(F, η)

1. Define

$$Q_O = \{\langle f_i, g_i, g_{i+1}, f_{i+1}\rangle : i \text{ is odd}\},$$
$$Q_E = \{\langle f_i, g_i, g_{i+1}, f_{i+1}\rangle : i \text{ is even}\}.$$

2. Let q_O and q_E be the concatenation of strings from Q_O and Q_E, respectively.
3. Let q be the shorter of q_O and q_E; output q.

Clearly, q is a superstring of $F \cup G$. Moreover, we have the following lemma.

LEMMA 4.5. *The amount of overlap in q constructed above is at least $ov(F, G)/2 + ov(\eta)/2$.*

Proof. It follows from the fact that $Q_O \cup Q_E$ and $C_{F,G} \cup \eta$ correspond to two decompositions of the same edge set in G_S (see Figure 1). □

Given F and G, we will apply the above procedure Insert(F, η) to the string η obtained by first running the Greedy algorithm on G. (It would work equally well if we obtained η via an optimal assignment, as described in Lemma 3.2.) Call the combined procedure Greedy-Insert(F, G). It produces a superstring of $F \cup G$ with guaranteed overlap because of Lemmas 3.3 and 4.5.

LEMMA 4.6 (Greedy-Insert). *The amount of overlap in the superstring generated by* Greedy-Insert(F, G) *is at least $ov(F, G)/2 + \max\{0, (|G| - opt(G))/4\}$.*

4.3. The algorithm. We now present our new approximation algorithm.

ALGORITHM (superstring approximation)
Input: $S = \{s_1, \ldots, s_n\}$.

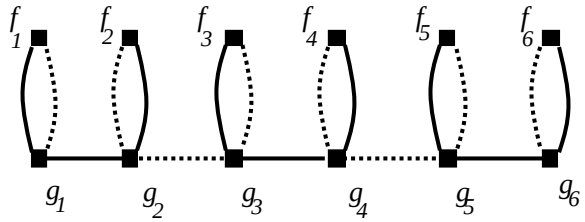
1. Find an optimal assignment C of S, and make C canonical.
2. Take an arbitrary string from each cycle of C to form a set R, and find an optimal assignment CC for R.
3. Let F be the set that contains the shorter string from each 2-cycle in CC, and let G be the set containing the longer string from each 2-cycle in CC; run Greedy-Insert(F, G) to obtain q.
4. Open each non-2-cycle in CC by deleting the edge with the smallest overlap; concatenate the resulting strings together with q to obtain α; let $\bar{\alpha}$ be the extended string of α and return $\bar{\alpha}$.

4.4. Analysis. Clearly, the algorithm runs in polynomial time. To analyze its performance ratio, we first introduce some notation.

• Let d_2, d_3, and d_4 be, respectively, the total weight (in C) of the strings that appear in 2-cycles, 3-cycles, and all i-cycles for $i \geq 4$ in assignment CC. Thus $d_2 + d_3 + d_4 = d(C)$.

• Let ov_2, ov_3, and ov_4 be, respectively, the total overlap present in the 2-cycles, 3-cycles, and all i-cycles for $i \geq 4$ in assignment CC. Thus $ov_2 + ov_3 + ov_4 = ov(CC)$. Also from Lemma 3.1, we have $d(C) \leq opt(S)$ and $|R| - ov(CC) \leq opt(R) \leq opt(S)$.

Recall that G is the set containing the longer string from each 2-cycle in CC. Summing over all 2-cycles (using Lemma 4.4), we obtain the following lemma.



$$Q_O = \langle f_1, g_1, g_2, f_2, f_3, g_3, g_4, f_4, f_5, g_5, g_6, f_6 \rangle,$$
$$Q_E = \langle g_1, f_1, f_2, g_2, g_3, f_3, f_4, g_4, g_5, f_5, f_6, g_6 \rangle.$$

FIG. 1. *The decompositions induced by Q_O and Q_E are indicated by bold and dashed edges, respectively.*

LEMMA 4.7. $ov_2 \leq |G| + d_2/2$.

LEMMA 4.8. $|\alpha| \leq (1 + 8/9)opt(S)$.

Proof. By Lemmas 3.2 and 4.6, the amount of overlap in α, $ov(\alpha)$, is at least max(a,b), where

$$a = ov_2/2 + 2ov_3/3 + 3ov_4/4,$$
$$b = ov_2/2 + (|G| - opt(G))/4 + 2ov_3/3 + 3ov_4/4.$$

Recall that $ov_i \leq 2d_i$ for $i = 2, 3, 4$ by Lemma 3.5. Therefore,

$$\begin{aligned} |\alpha| &\leq |R| - a \\ &\leq opt(S) + ov_2/2 + ov_3/3 + ov_4/4 \\ &\leq opt(S) + d_2 + (2/3)d_3 + (1/2)d_4, \end{aligned} \tag{1}$$

where we have made use of the fact that $|R| \leq opt(S) + ov(CC)$.

Since $|G| \geq ov_2 - d_2/2$ by Lemma 4.7 and since $opt(G) \leq opt(S)$, we have $b \geq ov_2/2 + (ov_2 - d_2/2 - opt(S))/4 + 2ov_3/3 + 3ov_4/4 = 3ov_2/4 + 2ov_3/3 + 3ov_4/4 - d_2/8 - opt(S)/4$. Therefore,

$$\begin{aligned} |\alpha| &\leq |R| - b \\ &\leq 5opt(S)/4 + ov_2/4 + ov_3/3 + ov_4/4 + d_2/8 \\ &= 5opt(S)/4 + ov(CC)/4 + ov_3/12 + d_2/8 \\ &\leq (2 - 1/4)opt(S) + d_3/6 + d_2/8. \end{aligned} \tag{2}$$

We complete the analysis by considering two cases.

Case 1. If $d_2 \leq (2/3)d(C)$, then the right-hand side of (1) is maximized when $d_2 = (2/3)d(C)$ and $d_3 = (1/3)d(C)$, giving

$$\begin{aligned} |\alpha| &\leq opt(S) + (2/3)d(C) + (2/3)(1/3)d(C) \\ &\leq (1 + 8/9)opt(S). \end{aligned}$$

Case 2. If $d_2 > (2/3)d(C)$, then $d_3 \leq d(C)/3$. By inequality (2),

$$\begin{aligned} |\alpha| &= (2 - 1/4)opt(S) + (d_2 + d_3)/8 + d_3/24 \\ &\leq (2 - 1/4)opt(S) + d(C)/8 + d(C)/72 \\ &\leq (1 + 8/9)opt(S). \end{aligned}$$

Now the result follows immediately from Lemma 3.6. □

THEOREM 4.9. *There exists an $O(n^3 + n|S|)$-time algorithm for constructing superstrings that achieves an approximation ratio of* 2.89.

5. Conclusion. The shortest-superstring problem is a fundamental problem about strings with some natural applications. We have established a better approximation bound than what was previously available. Given the possible large sizes of $|S|$ and $opt(S)$, a small percentage improvement in superstring approximation might still be viewed as significant. We do not know any examples for which our algorithm's approximation ratio is worse than 2; thus it is possible that the analysis can be further improved. We close by raising two open problems:

- Give further improvement to the 2.89 upper bound or the $1+\epsilon$ lower bound (as provided by [1]) for superstring approximation.
- Find an algorithm that achieves better than 50% of the optimal compression. Such an algorithm, when used as a subroutine, will immediately lead to improvement in length approximation as well.

REFERENCES

[1] A. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and hardness of approximation problems*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 14–23.

[2] A. BLUM, T. JIANG, M. LI, J. TROMP, AND M. YANNAKAKIS, *Linear approximation of shortest superstrings*, in Proc. 23rd ACM Symposium on the Theory of Computing, ACM, New York, 1991, pp. 328–336.

[3] N. FINE AND H. WILF, *Uniqueness theorems for periodic functions*, Proc. Amer. Math. Soc., 16 (1965), pp. 109–114.

[4] J. GALLANT, D. MAIER, AND J. STORER, *On finding minimal length superstrings*, J. Comput. System Sci., 20 (1980), pp. 50–58.

[5] M. GAREY AND D. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, New York, 1979.

[6] T. R. GINGERAS, J. P. MILAO, P. SCIAKY, AND R. J. ROBERTS, *Computer programs for the assembly of DNA sequences*, Nucleic Acid Res., 7 (1979), pp. 529–545.

[7] A. LESK, ED., *Computational Molecular Biology: Sources and Methods for Sequence Analysis*, Oxford University Press, Oxford, UK, 1988.

[8] M. LI, *Towards a DNA sequencing theory*, in Proc. 31st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 125–134.

[9] A. MAYNE AND E. B. JAMES, *Information compression by factorising common superstrings*, Comput. J., 18 (1975), pp. 157–160.

[10] C. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, Englewood Cliffs, NJ, 1982.

[11] C. PAPADIMITRIOU AND M. YANNAKAKIS, *Optimization, approximation and complexity classes*, in Proc. 20th ACM Symposium on the Theory of Computing, ACM, New York, 1988, pp. 229–234.

[12] H. PELTOLA, H. SODERLUND, J. TARHIO, AND E. UKKONEN, *Algorithms for some string matching problems arising in molecular genetics*, in Information Processing: Proc. IFIP World Computer Congress, Elsevier, New York, 1983, pp. 53–64.

[13] M. B. SHAPIRO, *An algorithm for reconstructing protein and RNA sequences*, J. Assoc. Comput. Mach., 4 (1967), pp. 720–731.

[14] J. STORER AND T. SZYMANSKI, *Data compression via textual substitution*, J. Assoc. Comput. Mach., 29 (1982), pp. 928–951.

[15] J. STORER, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.

[16] J. TARHIO AND E. UKKONEN, *A greedy approximation algorithm for constructing shortest common superstrings*, Theoret. Comput. Sci., 57 (1988), pp. 131–145.

[17] R. E. TARJAN, *Data Structures and Network Algorithms*, SIAM, Philadelphia, 1983.

[18] J. TURNER, *Approximation algorithms for the shortest common superstring problem*, Inform. and Comput., 83 (1989), pp. 1–20.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 418–455, April 1997

007

BOUNDED CONCURRENT TIME-STAMPING*

DANNY DOLEV† AND NIR SHAVIT‡

Abstract. We introduce concurrent time-stamping, a paradigm that allows processes to temporally order concurrent events in an asynchronous shared-memory system. *Concurrent time-stamp systems* are powerful tools for concurrency control, serving as the basis for solutions to coordination problems such as mutual exclusion, ℓ-exclusion, randomized consensus, and multiwriter multireader atomic registers. Unfortunately, all previously known methods for implementing concurrent time-stamp systems have been theoretically unsatisfying since they require unbounded-size time-stamps—in other words, unbounded-size memory.

This work presents the first bounded implementation of a concurrent time-stamp system, providing a modular unbounded-to-bounded transformation of the simple unbounded solutions to problems such as those mentioned above. It allows solutions to two formerly open problems, the bounded-probabilistic-consensus problem of Abrahamson and the *fifo-ℓ*-exclusion problem of Fischer, Lynch, Burns and Borodin, and a more efficient construction of *multireader multiwriter* atomic registers.

Key words. atomic registers, serialization, concurrency, time-stamping, distributed computing, parallel computing

AMS subject classifications. 68Q22, 05C90, 05C99

PII. S0097539790192647

1. Introduction.

A *time-stamp system* is like a ticket machine at an ice cream parlor. People's requests to buy the ice cream are time-stamped based on a numbered ticket (label) taken from the machine. In order to know the order in which requests will be served, a person need only scan through all the numbers and observe the order among them. A *concurrent* time-stamp system (CTSS) is a time-stamp system in which any process can either take a new ticket or scan the existing tickets simultaneously with other processes. A CTSS is required to be *wait-free*, which means that a process is guaranteed to finish any of the two above-mentioned label-taking or scanning tasks in a finite number of steps, even if other processes experience stopping failures. Wait-free algorithms are highly suited for fault-tolerant and real-time applications (see Herlihy [Her91]).

Concurrent time-stamping is the basis for simple solutions to a wide variety of problems in concurrency control. Examples of such algorithms include Lamport's *first-come first-served* mutual exclusion [Lam74], Vitanyi and Awerbuch's construction of a multireader multiwriter (MRMW) atomic register [VA86], Abrahamson's randomized consensus [Abr88], and Fischer, Lynch, Burns, and Borodin's *fifo-ℓ*-exclusion problem [FLBB79, FLBB89] (also see [AD*94]).

* Received by the editors December 10, 1990; accepted for publication (in revised form) May 18, 1995. A preliminary version of this paper appeared in *Proc. 21st Annual ACM Symposium on Theory of Computing*, ACM, New York, 1989, pp. 454–465.

http://www.siam.org/journals/sicomp/26-2/19264.html

† IBM Almaden Research Center, K53/802, 650 Harry Road, San Jose, CA 95120-6099 (dolev@almaden.ibm.com) and Institute of Mathematics and Computer Science, Hebrew University, Givat-Ram, Jerusalem 91906, Israel (dolev@cs.huji.ac.il).

‡ Department of Computer Science, Hebrew University, Givat-Ram, Jerusalem 91906, Israel. Current address: Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel (shanir@cs.tau.ac.il). The research of this author was supported by a Libnitz Foundation Scholarship, the Israeli Communications Ministry Award, NSF contract CCR-8611442, ONR contract N0014-85-K-0168, DARPA contract N00014-83-K-0125, and a special grant from IBM. Parts of this research were also conducted while this author was visiting the Theory of Distributed Systems group at MIT, AT&T Bell Laboratories, and the IBM Almaden Research Center.

Unfortunately, the only formerly known implementation of the CTSS paradigm using read/write registers was a version of Lamport's "bakery algorithm," which uses labels of unbounded size [Lam74]. Researchers were thus led to devise complicated problem-specific solutions to show that the above problems are solvable in a bounded way.[1]

In [IL93], Israeli and Li were the first to isolate the notion of bounded time-stamping (time-stamping using bounded-size memory) as an independent concept, developing an elegant theory of bounded *sequential* time-stamp systems. Sequential time-stamp systems prohibit concurrent operations. This work was continued in several interesting papers on sequential systems with weaker ordering requirements by Li and Vitanyi [LV87], Cori and Sopena [CS93], and Saks and Zaharoglou [SZ91].

This paper introduces the *concurrent time-stamping* paradigm and provides the first bounded construction of a concurrent time-stamp system. It provides a modular unbounded-to-bounded transformation, enabling the design of simple unbounded concurrent-time-stamp-based algorithms to problems such as those mentioned above, with the knowledge that each unbounded solution immediately implies a bounded one. Our work allows solutions of the above flavor to two formerly open problems, the bounded-randomized-consensus problem of [Abr88] (which requires one to solve the randomized-consensus problem of [CIL87] without using an atomic coin-flip operation) and the *fifo*-ℓ-exclusion problem of [FLBB79, FLBB89] (see [AD*94] for details). A bounded CTSS solution to the former problem is given in [Sha90], and in [AD*94], Afek et al. use a CTSS to provide the first bounded solution to the latter problem.[2]

Though one might think that the price of introducing a modular unbounded-to-bounded transformation would be a blowup in memory size or number of operations, this is hardly the case. For an n-process system, the construction presented in this paper requires only n registers of $O(n)$ bits each, meeting the lower bound of [IL93] for sequential-time-stamp-system construction. The time complexity is $O(n)$ operations for an update and $O(n^2 \log n)$ for a scan. (Like the unbounded algorithm, the scan consists only of read operations, i.e., no writes.)

One example of the efficiency of the CTSS solutions is given by the famous problem of multireader multiwriter atomic register construction. A simple solution based on transforming the unbounded protocol of Vitanyi and Awerbuch [VA86] using our construction (see [Sha90, G92]) has the same space complexity of the [PB87, Sch88] algorithm, yet it has a better time complexity—$O(n)$ memory accesses for a write, $O(n \log n)$ for a read, as compared with $O(n^2)$ for either in the former solutions. Our implementation is the only known bounded construction of an MRMW atomic register from single-writer multireader (SWMR) atomic registers where the implementation of the MRMW read operation does not require a process to perform an SWMR write. The importance of the readers-do-not-write property was first raised by Lamport in [Lam86a], where he showed the impossibility of a bounded construction where readers do not write of a single-writer single-reader (SWSR) atomic register from SWSR regular ones. Moreover, as explained in [AD*94], this property is important when defining liveness conditions such as *first-come first-enabled* for problems like ℓ-exclusion.

The structure of our presentation is as follows. We begin by describing concurrent time-stamping (sections 2 and 3), first formally using Lamport's axiomatic approach

[1] See [And89a, Blo88, BP87, CIL87, Dij65, DGS88, FLBB79, FLBB89, Kat78, Lam74, Lam77, Lam86b, LH89, LV87, **?**, Ray86, Pet81, Pet83, PB87, VA86].

[2] The only prior known solutions to the *fifo*-ℓ-exclusion problem [DGS88, Pet88] achieve weaker forms of fairness than the original *test-and-set*-based solution of [FLBB79].

[Lam86c, Lam86a] and then informally through a simple unbounded-memory implementation. In sections 4.3 and 4.4, the bounded wait-free CTSS implementation is described. Section 5 provides the final details of the formal specification and the main parts of the proof of the bounded CTSS implementation are presented. Section 6 describes the implications of a bounded CTSS construction on various interprocess-communication problems and gives a summary of research following our work. For berevity, some of the more tedious parts of the correctness proof have been omitted and can be found in [Sha90].

2. A concurrent time-stamp system. The following is a formal definition of a CTSS for a system of processes numbered $1, \ldots, n$. It uses the axiomatic specification formalism of Lamport [Lam86c, Lam86a]. The reader may benefit by checking how the formal properties described below are met by the unbounded implementation described in the next section.

A CTSS is a problem specification with an operational interface. A CTSS that permits n concurrent operations has $2n$ operation types, specifically, $\mathit{labeling}_i(\ell_i)$ and $\mathit{scan}_i(\bar{\ell}, \prec)$ for $i \in \{1, \ldots, n\}$. A $\mathit{labeling}_i$ operation associates an input *value*, ℓ_i, taken from any domain $\mathcal{D}$ with a label.[3] We call ℓ_i the *labeled-value* of operation $\mathit{labeling}_i$. In an application such as an atomic-register construction, the labeled-value would be the value written to the register, while in a mutual-exclusion-type application, where the input values are unimportant, it would be null. A scan_i operation returns as output a pair $(\bar{\ell}, \prec)$, where the *view* $\bar{\ell} = \{\ell_1, \ldots, \ell_n\}$ is an indexed set of labeled-values (one per process) and $\prec$ is a *total order* on these indexes.

Assume that each process' program consists of these two operations, whose execution generates a sequence of *elementary operation executions*, totally ordered by the *precedes* relation (of [Lam86c, Lam86a], denoted "$\longrightarrow$") and where any number of scan operation executions are allowed between any two labeling operation executions. The following,

$$L_i^{[1]} \longrightarrow S_i^{[1]} \longrightarrow L_i^{[2]} \longrightarrow L_i^{[3]} \longrightarrow S_i^{[2]} \longrightarrow S_i^{[3]} \longrightarrow S_i^{[4]} \longrightarrow \cdots,$$

is an example of such a sequence by process i, where $L_i^{[k]}$ denotes process i's kth execution of a labeling operation and $S_i^{[k]}$ is the kth execution of a scan operation. (The superscript $[k]$ is used for notation and is not visible to the processes.) The labeled-value input in each labeling operation execution $L_i^{[k]}$ is denoted by $\ell_i^{[k]}$.[4] A *global-time model* of operation executions is assumed, implying that for any two operation executions, $a \longrightarrow b$ or $b \dashrightarrow a$. (For more details, see section 5.1.)

The elementary operation executions of a CTSS must have following set of properties.

P1: *ordering.* There exists an irreflexive total order $\Longrightarrow$ on the set of all labeling operation executions such that we have the following:

a: *precedence.* For any pair of labeling operation executions $L_p^{[a]}$ and $L_q^{[b]}$ (where p and q are possibly the same process), if $L_p^{[a]} \longrightarrow L_q^{[b]}$, then $L_p^{[a]} \Longrightarrow L_q^{[b]}$.

b: *consistency.* For any scan operation execution $S_i^{[k]}$ that returns $(\bar{\ell}, \prec)$, $p \prec q$ if and only if $L_p^{[a]} \Longrightarrow L_q^{[b]}$.

[3] In order correctly handle initial conditions, the value domain $\mathcal{D}$ must specify some initial value.

[4] In order for a unique labeled-value $\ell_i^{[k]}$ to be associated with each label operation execution $L_i^{[k]}$, the reader can think of $\ell_i^{[k]}$ as a triplet $\langle \ell_i^{[k]}, i, k \rangle$, where the second and third fields are dummy indexes used only for purposes of the specification.

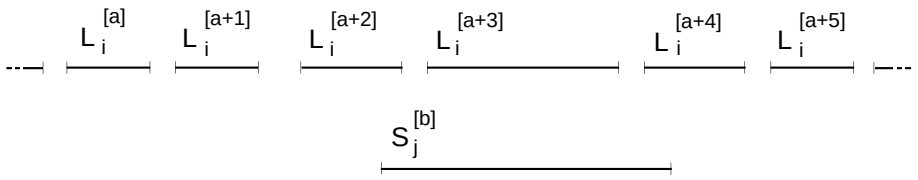


FIG. 1. *Regularity.*

Property P1 formalizes the idea that a CTSS can be envisioned as a black box, inside of which hides a mechanism (a logical clock) associating causally ordered time stamps—from an infinite totally ordered range—with each of the labeled-values entered in labeling operations, and where scanning is like peeping into this black box, each scan returning a view of a part of this hidden ordering.[5] The black box metaphor is used to stress that it suffices to know of the existence of such a total ordering $\Longrightarrow$, while the ordering itself need not be known.

One should bear in mind that the asynchronous nature of the operations allows situations where a scan operation execution overlaps many consecutive labeling operation executions of other processes. Also, several consecutive scans could possibly be overlapped by a single labeling operation execution. It is therefore important that a requirement be made that the view $\bar{\ell}$ returned by $S_i^{[k]}$ be a meaningful one, namely, that it reflect the ordering among labeling operation executions immediately before or concurrent with the scan, and not just any possible set of labeled-values. (In the example of Figure 1, any of the labeled-values $\ell_x^{[a+1]}$ through $\ell_x^{[a+4]}$ can be returned by $S_i^{[k]}$, but not those preceding or following them.) This will eliminate uninteresting trivial solutions and introduce a measure of liveness into the system. This requirement is formalized in the following definition, where $\dashrightarrow$ is the *can affect* relation of [Lam86c, Lam86a].

P2: *regularity.* For any labeled-value $\ell_p^{[a]}$ in $\bar{\ell}$ of $S_i^{[k]}$, $L_p^{[a]} \dashrightarrow S_i^{[k]}$, and there is no $L_p^{[b]}$ such that $L_p^{[a]} \longrightarrow L_p^{[b]} \longrightarrow S_i^{[k]}$.

Although such a *regular* concurrent time-stamp system as P1–P2 would suffice for some applications (as in Lamport's "bakery algorithm" [Lam74]), a more powerful *monotonic* concurrent time-stamp system will be needed in applications such as the *multireader multiwriter* atomic register construction (as in [LV87, VA86]). zTo this end, the following third property is added.

P3: *monotonicity.* For any labeled-value $\ell_p^{[a]}$ in $\bar{\ell}$ of $S_i^{[k]}$, there does not exist an $S_j^{[k']}$ with a labeled-value $\ell_p^{[b]}$ in its view $\bar{\ell}'$, such that $S_i^{[k]} \longrightarrow S_j^{[k']}$ and $L_p^{[b]} \longrightarrow L_p^{[a]}$ (possibly $i = j$).

Monotonicity is the property that in the unbounded natural-number CTSS can be described by saying that the labels of any one process, as read by increasingly later scans, are "monotonically nondecreasing." In other words, later scans cannot read labels smaller than those read by earlier ones. It is important to note, however, that P3 does not imply that labeling and scan operation executions of all processes are serializable, that is, appear to happen atomically. (Figure 2 shows two scan operations that meet property P3 that cannot be serialized.) It does, however, imply the serializability of the scan operation executions of all processes relative to the labeling operation executions of any *one* process.

[5] Notice that there is no requirement that labeled-values returned by different scans must be comparable.

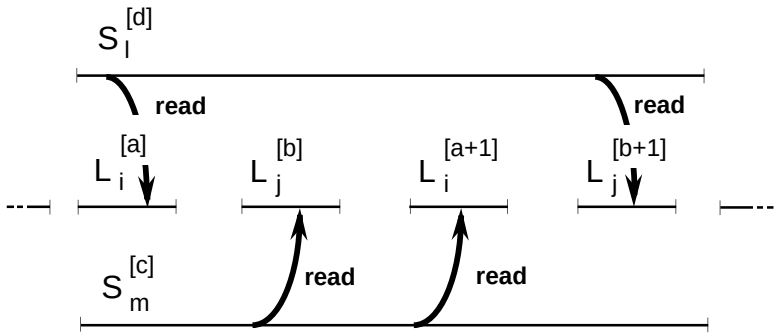


FIG. 2. *Monotonicity does not imply atomicity.*

Property P4 is an extension of part of the regularity property to the $\Longrightarrow$ order.[6] Properties P3 and P4 together imply that all *scan* operations that consider only the "largest" value, where "largest" is based on the $\prec$ ordering, can be serialized with respect to all labeling operations.

P4: *order regularity.* For any labeled-value $\ell_p^{[a]}$ in $\bar{\ell}$ of $S_i^{[k]}$, $S_i^{[k]} \longrightarrow L_q^{[b]}$ implies that $L_p^{[a]} \Longrightarrow L_q^{[b]}$.

3. Unbounded concurrent time-stamping. The basic communication primitive used in our implementations is a single writer multireader atomic register. Our goal is to design an implementation that is *wait-free* [Her91, AG90]: each process' *scan* or *label* operation execution consists of a bounded number of SWMR register operations independently of the pace or type of operations carried out by other processes. Wait-free constructions of SWMR atomic registers from weaker primitives have been shown in [BP87, IL93, Lam86d, SAG94, New87].

We begin with the following simple implementation of a CTSS using SWMR registers of unbounded size. The concurrent time stamp system will consist of n SWMR atomic registers v_i, $i \in \{1..n\}$. Each v_i is written by process i and read by all. Each $labeling_i$ operation writes ℓ_i to register v_i. In our implementation, ℓ_i is a data type consisting of two fields, a labeled-value, denoted $value(\ell_i)$, and its associated label, denoted $label(\ell_i)$. Each $label(\ell_i)$ is a pair of the form $(number_i, i)$, where $number_i$ is a natural number and $i \in \{1..n\}$ is the id of the process writing ℓ_i.

A process i collects the labels and values of other processes by performing a *collect* operation, a reading of all the registers v_j, $j \in \{1..n\}$, once each, in some arbitrary order. The collect operation returns an indexed set $\ell = \{\ell_1, \ldots, \ell_n\}$, that is, one value and associated label per process. The collected elements in ℓ are ordered by $ord(\ell)$, an ordering on their indexes in $\{1..n\}$, such that i is smaller than j if and only if the label $(number_i, i)$ is lexicographically smaller than the label $(number_j, j)$. Figure 3 provides the pseudocode of the *labeling* and *scan* operations for a process i.

To understand how property P1 is met, consider that if the labeling operation execution of ℓ_i by a process i completely preceded the labeling operation execution of ℓ_j by j, then it must be that j chose a label with $number_j > number_i$ since j collected ℓ_i. If they are concurrent, at worst they might both collect the same maximal label and choose $number_i = number_j$, in which case they are ordered by their ids. Thus

[6] The need for property P4 in applications such as the multireader multiwriter atomic register construction of [LV87, VA86] was discovered by Gawlick [G92].

```
procedure labeling(val);
  begin
    ℓ := collect;
    v_i := (val, (max_{j∈{1..n}} number_j + 1, i));
  end;
function scan;
  begin
    ℓ := collect;
    return ({value(ℓ_1) ... value(ℓ_n)}, ord(ℓ));
  end;
```

FIG. 3. *The unbounded* natural-number-*based implementation.*

the lexicographic order on the labels defines a linearization order [HW88] on the concurrent labeling operation executions, that is, an order $\Longrightarrow$ by which they can be thought of as happening sequentially in time. The reader can convince herself that properties P2–P4 follow directly from the use of SWMR atomic registers in the implementation.

It is important to note that the actual $label(\ell_1) \ldots label(\ell_n)$ used in computing $ord(\ell)$ are hidden from the user (scan operations do not return them), and there is thus no way to compare the order among a pair of values returned by different scans.

4. A bounded concurrent time-stamp system.

4.1. Labels and precedence. The bounded implementation presented will be of the exact same form as the unbounded natural-number-based one. The concurrent time-stamp system will consist of n SWMR atomic registers v_i, $i \in \{1..n\}$, each v_i written by process i and read by all. Each value ℓ_i written to register v_i consists, just as in the unbounded case, of two fields, a *labeled-value*, to which the input of a labeling operation is written, and an associated *label*.

Note. In what follows, almost all of the discussion involves only the label field of v_i and not its labeled-value field. In order to simplify the exposition, we choose, with few exceptions, to ignore the existence of the labeled-value field and deal only with the associated label field. Thus, for example, the notation $\ell_i^{[k]}$ will represent only the label field written in a labeling operation execution $L_i^{[k]}$. We trust that the interested reader will be able to add the relevant operations regarding the labeled-value, as in the unbounded implementation in section 3.

Let V denote the range of possible labels and $\stackrel{\nu}{\prec}$ denote an irreflexive and anti-symmetric relation among them. In the unbounded natural-number implementation of a CTSS, V is just the unbounded size set of pairs of natural numbers and integers in $\{1..n\}$ and $\stackrel{\nu}{\prec}$ is the lexicographic total ordering among them. In the following sections, the set of possible label values V of the implementation, together with a relation $\stackrel{\nu}{\prec}$ among them, are defined in terms of a *precedence graph*[7] $(V, \stackrel{\nu}{\prec})$. Each possible label is a node in this graph. The order among the labels in any two registers is the order $\stackrel{\nu}{\prec}$ established by the edges of the precedence graph. A tournament is a complete directed graph. The precedence graph representing labels of the natural-number-based implementation is an acyclic tournament of unbounded size, i.e., a total

[7] The elegant idea of defining the labels and ordering as a tournament graph was introduced by Israeli and Li in [IL93].

order. The definition of the precedence graph will provide the basis for describing the implementation of the labeling and scan operations.

4.2. A bounded precedence graph. The following is the description of the *precedence graph* T^n (see Figure 4). Unlike the unbounded precedence graph defined by the natural numbers, T^n contains cycles.

Define "*A dominates B* in *G*," where A and B are two subgraphs of a graph G (possibly single nodes), to mean that every node of A has edges directed to every node of B. Define the following generalization of the composition operator of [IL93]. The *α-composition*, $G \circ_\alpha H$, of two graphs G and H, where α is a subset of the nodes of G, is the following noncommutative operation:

> Replace every node $v \in \alpha$ of G by a copy of H (denoted H_v), and let H_v (or v) dominate H_u in $G \circ_\alpha H$ if v dominates u in G.

Define the graph T^2 to be the following graph of five nodes: a cycle of three nodes $\{3, 4, 5\}$, where 3 dominates 5, which dominates 4, which in turn dominates 3, all dominating the nodes $\{2, 1\}$, and where node 2, in turn, dominates node 1.

Define the graph T^k (a tournament) inductively as follows:

1. T^1 is a single node.
2. $T^k = T^2 \circ_\alpha T^{k-1}$, where $\alpha = \{5, 4, 3, 1\}$ and $k > 1$.

The graph $T^n = (V, \overset{\vee}{\prec})$ is the precedence graph to be used in the implementation of the labeling and scan algorithms of a concurrent time-stamp system for n processes. For any process i, each node in T^n corresponds to a uniquely defined label value ℓ_i. The label can be viewed as a string $\ell_i[n..1]$ of n digits, where each $\ell_i[k] \in \{1, \ldots, 5\}$ is the digit of the corresponding node in T^2, replaced by a T^k subgraph during the kth step of the inductive construction above. The digit $\ell_i[n]$ is always 1, representing the complete T^n graph, and if in ℓ_i, $\ell_i[k] = 2$, then $\ell_i[j] = 1$ for all $j \in \{k-1..1\}$ (since node 2 is never expanded in the induction step). Therefore, given any label ℓ_i, the T^k subgraph of T^n in which its corresponding node is located is identified by the corresponding prefix $\ell_i[n..k]$.

To assure that based on the graph T^n a total ordering among the label values returned by a scan can be established, we need to break symmetry among processes having the same label. Thus the label ℓ_i is assumed to be concatenated with the id of process i, where label and id are lexicographically ordered. (In terms of the graph T^n, this amounts to no more than assuming that each T^1 graph consists of a total order tournament of n nodes, each process i always choosing the ith node in the order. For simplicity, this point is not further elaborated upon in what follows.)

4.3. The labeling operation. Recall that the *collect* operation by any process i is a reading of all the registers v_j, $j \in \{1..n\}$, once each, in an arbitrary order, returning a set ℓ of labels. The *labeling* operation of a process i is of the form described in Figure 5, where $\mathcal{L} : V^n \times \{1..n\} \mapsto V$ is a *labeling function*, returning a label value ℓ_i "greater than" all other label values.[8] This is the same form as the natural number CTSS, where the labeling function $\mathcal{L}$ returns $(\max_{j \in \{1..n\}} number_j + 1, i)$. However, the interpretation of being "greater than" is not as straightforward as in the natural-number case.

The definition of the labeling function $\mathcal{L}(\ell, i)$ presented below is based on a recursively defined function $\mathcal{L}^k(G, \ell, \ell_{\max})$, which, given a T^k subgraph G of T^n, a set of labels ℓ, and a "maximal" label $\ell_{\max} \in \ell$ in T^k, returns the label of a node in G

[8] Initially, all labels are on node 111..11, the node dominated by all others in T^n.

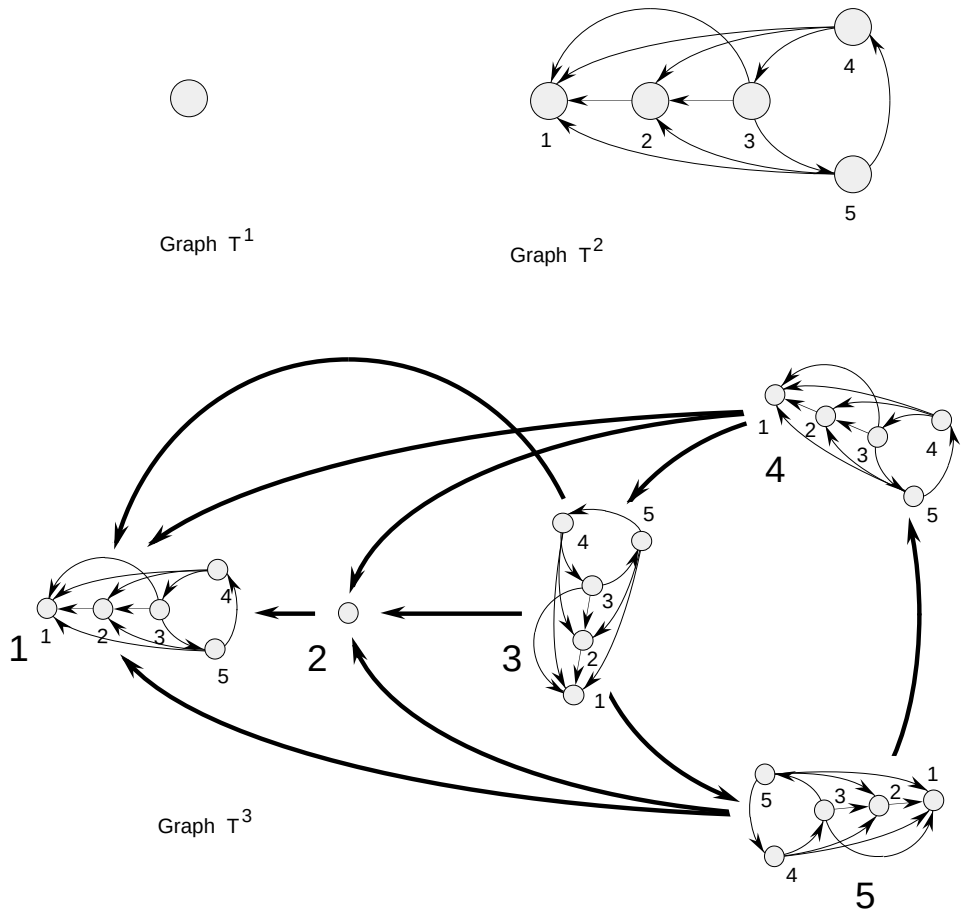


FIG. 4. *The precedence graph.*

procedure *labeling*(*val*);
  begin
    $\ell := collect$;
    $v_i := (val, \mathcal{L}(\ell, i))$;
  end;

FIG. 5. *The labeling operation.*

that is "greater than" the other labels. The reader may benefit by going through Examples 4.1, 4.2, and 4.3 before or during the reading of the code in Figure 6. For simplicity, and since the collected set of labels ℓ remains unchanged in $\mathcal{L}(\ell, i)$ once it is collected (similarly for the variable $\ell_{\max}$ once it is computed), it is treated as a global variable and is not passed as a parameter in all of the utility functions used by $\mathcal{L}(\ell, i)$. The following functions are used in defining $\mathcal{L}$:

num_labels(G)—a function that, for the given label set ℓ, returns how many of the labels are in subgraph G;

```
function L(ℓ, i);
    function L^k(G);
    begin
     1: if k= 1 then return G;
     2: if ℓ_max[n..k] ≠ G
          then return L^{k-1}(G.1);
     3: if ℓ_max[n..k-1] = G.2
          then return L^{k-1}(G.3);
     4: if k > 2 then
          if ℓ_max[k-2] ∈ {2,3,4,5} and
            (ℓ_i[n..k-1] ≠ ℓ_max[n..k-1])
          then return L^{k-1}(G.dom(ℓ_max[k-1]));
     5: if (num_labels(ℓ_max[n..k-1]) < k-1) or
          ((num_labels(ℓ_max[n..k-1]) = k-1) and
          (ℓ_i[n..k-1] = ℓ_max[n..k-1]))
          then return L^{k-1}(G.ℓ_max[k-1])
          else return L^{k-1}(G.dom(ℓ_max[k-1]));
    end L^k;
begin
    ℓ_max := max(dominating_set(ℓ, ℓ_i));
    return L^n(T^n);
end L;
```

FIG. 6. *The labeling function.*

$dom(x)$—a function that, for a given digit $x \in \{1..5\}$ representing a node in the graph T^2, returns the next dominating node, namely, $dom(1) = 2$, $dom(2) = 3$, $dom(3) = 4$, $dom(4) = 5$, and $dom(5) = 3$;

$dominating\_set(\hat{\ell}, \ell_i)$—a function that, for a set of labels $\hat{\ell} \subseteq \ell$ and a label $\ell_i \in \hat{\ell}$, returns a subset of labels $\{\ell_j \in \hat{\ell} \,|\, \ell_i \overset{V}{\prec} \ell_j\} \cup \{\ell_i\}$; and

$max(\hat{\ell})$—a function that, for a set of labels $\hat{\ell} \subseteq \ell$, returns a label

$$(\ell_x \in \hat{\ell} \,:\, |dominating\_set(\hat{\ell}, \ell_x)| \leq |dominating\_set(\hat{\ell}, \ell_j)|, \forall \ell_j \in \hat{\ell}\,),$$

the maximal label, i.e., the one least dominated within this set.

Define $G.x$ to be the *concatenation* of string G and digit x. Figure 6 is thus the definition of the labeling function $\mathcal{L}(\ell, i)$, where the parameter subgraphs G are identified with the relative label prefixes and T^n is identified with the label 1. To give the reader some intuition about the properties of the labeling operation, let it be assumed that one can talk about the values of the labels of all processes at "points in time." To show how the labeling operation executions allow us to define the order $\Longrightarrow$, we will first argue informally that they meet a much simpler requirement, namely, that at any point in time, the following hold:

R1. The labels reflect the precedence among nonconcurrent labeling operation executions.

R2. The subgraph of the precedence graph T^n induced by the labeled nodes (those whose corresponding label is written in some v_i) contains no cycle.

Since T^n is a tournament, R2 implies that at any point in time, all labels are totally ordered. One should notice that these two requirements are easily met by the unbounded implementation since for any $n - 1$ nodes, one can always choose a

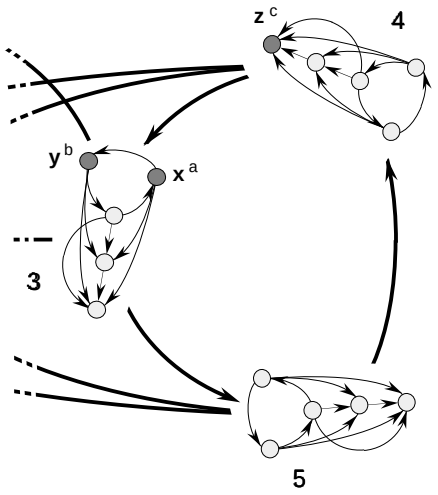


FIG. 7. *Starting state for the examples.*

dominating node in an unbounded total order graph in order to maintain R1, and this will never impair R2 because the graph does not contain cycles.

Let us begin by showing that the labeling operation executions maintain the following two "invariants" at any point in time:

(1) There are labels on at most two of the three nodes in any cycle of any subgraph T^k. (The cycle consists of "supernodes" $\{3, 4, 5\}$, called supernodes since they are actually T^{k-1} subgraphs.)

(2) There are no more than k labels in the cycle of any subgraph T^k.

Maintaining the second invariant is the key to maintaining the first, and the first implies R2.

The manner by which the invariance of (1) and (2) is preserved is explained via several examples. In these examples, T^3 is a precedence graph for a system of three processes x, y, and z. As shown in Figure 7, all of the examples start at a point in time where $\ell_y^{[b]} = 134$, $\ell_x^{[a]} = 135$, and $\ell_z^{[c]} = 141$, that is, all labels are totally ordered by $\precdot$. In the figure, a label such as $\ell_x^{[a]} = 135$ is denoted by shading node 135 and denoting it with the mark x^a.

Example 4.1. Assume that the following sequence of labeling operation executions occur sequentially. Process y performs $L_y^{[b+1]}$, reading $\ell_x^{[a]}$, $\ell_y^{[b]}$, and $\ell_z^{[c]}$ and moving based on $\mathcal{L}(\ell, y)$ to $\ell_y^{[b+1]} = 142$. Process z performs $L_z^{[c+1]}$, reading the new label $\ell_y^{[b+1]}$. It thus moves to the T^2 subgraph 14, following the rule that the node chosen should be the "lowest node dominating all other nodes with labels." This is actually the most basic rule implied by the definition of $\mathcal{L}$. The move to a dominating node is intended to meet R1.

Processes y and z can continue forever to choose $\ell_y^{[b+2]} = 144$, $\ell_z^{[c+2]} = 145$, $\ell_y^{[b+3]} = 143, \ldots$ (that is, move in the cycle of 14), maintaining the above invariants, because the T^2 graph is a precedence graph for two processes. If at some point x moves, in $L_x^{[a+1]}$ it will read the labels of both z and y as being in the T^2 subgraph 14. A T^2 subgraph is a precedence graph able to accommodate two labels and no more.

Since *num_labels*$('14') = 2$ in $L_x^{[a+1]}$, that is, there are already two labels in the T^2 subgraph, by line 5 of $\mathcal{L}(\ell, i)$, x will move to $\ell_x^{[a+1]} = 151$, and so on. □

The reader can convince herself that following any labeling operation execution $L_z^{[c]}$ by some process z, the above invariants hold. Furthermore, for the set of labels of processes y ($y \neq z$) that were read in $L_z^{[c]}$'s collect operation (denoted $read(L_z^{[c]})$), it is the case that

$$(\forall \ell_y^{[b]} \in read(L_z^{[c]}))\,(\ell_y^{[b]} \preceq \ell_z^{[c]}). \tag{3}$$

This invariant—that the new label chosen is greater than all those read—is the basis for meeting requirement R1.

As seen in the following example, in the concurrent case, more than k labels may move into the same T^k subgraph at the same time. It is thus not immediately clear why the second invariant holds.

Example 4.2. Assume that the following sequence of labeling operation executions occur concurrently. Processes x and y begin performing $L_x^{[a+1]}$ and $L_y^{[b+1]}$ concurrently, reading $\ell_x^{[a]}$, $\ell_y^{[b]}$, and $\ell_z^{[c]}$ and computing $\mathcal{L}$ such that $\ell_x^{[a+1]} = \ell_y^{[b+1]} = 142$. If they then continue to complete their operations by writing their labels, though they choose the same node, they were concurrent and can be ordered by relative id. If any of them were to continue to perform a new labeling operation, since *num_labels*$('14') > 2$, it would choose label 151, not entering the cycle. However, let us suppose that they do not both complete writing their labels, that is, x stops just before writing $\ell_x^{[a+1]}$ to v_x, while y writes $\ell_y^{[b+1]} = 142$. Process z then performs $L_z^{[c+1]}$, reading the new label $\ell_y^{[b+1]}$ and the old label $\ell_x^{[a]}$, thus moving to $\ell_z^{[c+1]} = 143$. Processes y and z continue to move into and in the cycle of the T^2 subgraph 14 since they continue to read x's old label. Then at some point, x completes $L_x^{[a+1]}$, and there are three labels in 14 (two of them in the cycle). However, if x now performs a new labeling $L_x^{[a+2]}$, it will read the labels of both x and y as being in 14. Since *num_labels*$('14') > 2$, by line 5 of $\mathcal{L}(\ell, i)$, x will move to $\ell_x^{[a+2]} = 151$, not entering the cycle. □

If nodes 1 and 2 did not exist in a T^2 subgraph (that is, each T^2 subgraph was a cycle of three nodes), a process' first move into T^k would be onto a node of the cycle. The reader can verify that the sequence of operations in Example 4.2, given that T^2 is just a cycle, would cause the labels of x, y, and z to end up each on a different node of the cycle, contradicting the first invariant. Based on the existence of nodes 1 and 2, this does not occur.

The following is intended to explain to the reader why for a given level k ($k = 2$ in the example), even if more than k processes move into a T^k subgraph *without* reading one another's labels, at most k of them will enter the cycle in T^k. The reason is the following well-known *flag principle*[9]:

> If there are $k + 1$ people, each of which first raises a flag and then counts the number of raised flags, at least one person must see $k + 1$ flags raised.

By the definition of the labeling function $\mathcal{L}$, each process moving into the cycle of a T^k subgraph must first move to either supernode 1 or 2 in T^k, and only then can it perform a labeling into the cycle. The move to 1 or 2 is the raising of the flag, and the move into the cycle is the counting of all flags.

[9] The proof follows since the last person to start counting flags must have seen $k + 1$ flags raised.

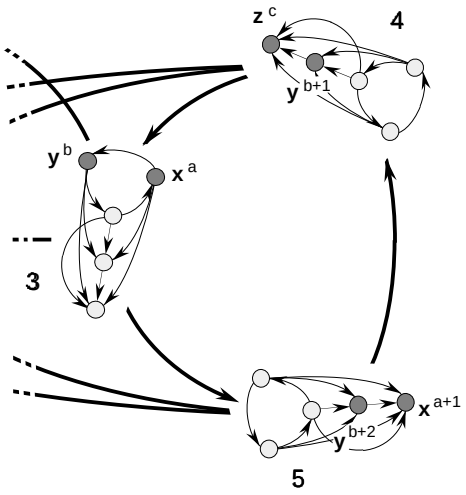


FIG. 8. *A collect returning a cycle.*

Example 4.3 below, which is depicted in Figure 8, shows that even though by the above there are at most k labels at a time in any T^k subgraph, the sets of labels *read* in a labeling operation execution may contain cycles.

Example 4.3. Process z begins performing $L_z^{[c+1]}$, reading $\ell_x^{[a]} = 135$. Process y then performs $L_y^{[b+1]}$, reading $\ell_x^{[a]}$, $\ell_y^{[b]}$, and $\ell_z^{[c]}$ and moving to $\ell_y^{[b+1]} = 142$. Process x performs $L_x^{[a+1]}$, reading the new label $\ell_y^{[b+1]}$ and $\ell_z^{[c]}$ and thus, by line 5 of $\mathcal{L}$, moving to $\ell_x^{[a+1]} = 151$. Process y then performs $L_y^{[b+2]}$, reading $\ell_x^{[a+1]}$ and moving to $\ell_y^{[b+2]} = 152$. Finally, process z reads $\ell_y^{[b+2]}$. It thus read $\ell_x^{[a]} = 135$, $\ell_y^{[b+2]} = 152$, and $\ell_z^{[c]} = 141$, three labels on a cycle. □

In order to select a label that dominates all others, z must establish where the "maximal label" among them is. To overcome the problem that the labels read form cycles (as in the example above), the labeling function $\mathcal{L}(\ell, z)$ does not take into account "old values" such as $\ell_x^{[a]}$; it considers only the labels that dominate the current label $\ell_z^{[c]}$.

In order to maintain the first invariant, z should move to $\ell_z^{[c+1]} = 131$ to dominate the current labels of both x and y without moving directly into the cycle. However, there is seemingly a problem since z did not read the label $\ell_x^{[a+1]} = 151$; so how can it know that there are already two labels in the T^2 subgraph 15? The solution is based on the fact that z can indirectly deduce the existence of $\ell_x^{[a+1]} = 151$. By the first invariant, in all of the cycle of T^3, there are at most three labels. In order to move to $\ell_y^{[b+1]} = 152$, y must have read some label in node 151 of the T^2 subgraph 15. By simple elimination, this must be a label of x. This rule is maintained by the application of line 4 in $\mathcal{L}^k$.

If the above scenario had occurred in the cycle of a T^k graph, where $k > 3$, then in order to allow the same reasoning as above, it would have to be that z's reading $\ell_y^{[b+2]} = 152$ (or $\ell_y^{[b+2]} \in \{153, 154, 155\}$) would imply that there are $k-2$ labels apart from that of y in the T^{k-1} subgraph 15. It would thus have to be that if $\ell_y^{[b+2]}$ is

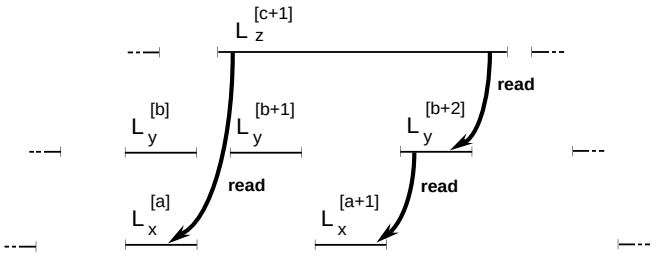


FIG. 9. *The observed relation.*

on supernode 2 in 15, it already established the existence of $k-2$ (and not just one) other labels in supernode 1.

It is for this purpose that supernode 1 of any T^k graph, where $k > 2$, is *not* a single node but a T^{k-1} subgraph. This creates a situation whereby as long as there are $k-1$ or fewer labels in T^k, all labels enter and move around in supernode 1. Supernode 2 can be chosen in $L_y^{[b+2]}$ only if $k-1$ labels were established by it as being in supernode 1 (i.e., supernode 1 is full). Since supernode 2 is a "bridge" that some process must "cross" (choose) before any process can move into the cycle, the above reasoning for z holds in case it read $\ell_y^{[b+2]} \in \{152, 153, 154, 155\}$.

Although the above invariants hold, it follows from Example 4.3 that the property that the chosen new label is greater than all those read, true for sequential labeling operation executions, does not hold in the concurrent case. Fortunately, there is a similar property that does hold, a property that will prove important in the implementation of the scan. Recall that $read(L_y^{[b]})$ denotes the set of label values read in the collect of $L_y^{[b]}$. Let us define the following *observed* relation among labeling operation executions to be the transitive closure of the *read* relation.

DEFINITION 4.1. *A labeling operation execution* $L_x^{[a]}$ *is* observed *by* $L_y^{[b]}$ (*denoted* $L_x^{[a]} \xrightarrow{\text{obs}} L_y^{[b]}$) *if* $\ell_x^{[a]} \in read(L_y^{[b]})$ *or there exists an* $L_z^{[c]}$ *such that* $\ell_z^{[c]} \in read(L_y^{[b]})$ *and* $L_x^{[a]} \xrightarrow{\text{obs}} L_z^{[c]}$.

DEFINITION 4.2. *Let the* maximal observed *set* $max\_obs(L_x^{[a]})$ *be defined as*

$$\{L_y^{[b]} \mid y \in \{1..n\},\ y \neq x,\ L_y^{[b]} \xrightarrow{\text{obs}} L_x^{[a]} \text{ and}$$
$$(\forall L_y^{[b']})\ (\text{if } L_y^{[b]} \longrightarrow L_y^{[b']}, \text{ then } L_y^{[b']} \not\xrightarrow{\text{obs}} L_x^{[a]})\}.$$

It thus consists of the "latest" of labeling operation executions observed for each process. In a concurrent execution, instead of invariant (3) stating that the new label chosen is greater than all the labels read, it is the case that

$$(\forall \ell_y^{[b]} \in max\_obs(L_x^{[a]}))\,(\ell_y^{[b]} \not\prec \ell_x^{[a]}). \tag{3'}$$

The new label chosen is greater than the latest of those observed for each process. As shown in Figure 9, for the labeling $L_z^{[c+1]}$ of Example 4.3, although z read $\ell_x^{[a]} = 143$ and $\ell_z^{[c+1]} \not\prec \ell_x^{[a]}$, it is the case that its maximal observed label is $\ell_x^{[a+1]}$, and $\ell_x^{[a+1]} \not\prec \ell_z^{[c+1]}$.

Finally, the following is the irreflexive total order $\Longrightarrow$ on the labeling operation executions as required by property P1.

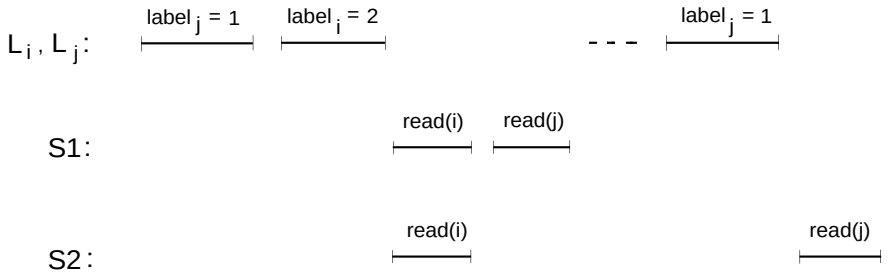


FIG. 10. *Ambiguity given bounded labels.*

DEFINITION 4.3. *Given any two distinct labeling operation executions* $L_x^{[a]}$ *and* $L_y^{[b]}$, $L_x^{[a]} \Longrightarrow L_y^{[b]}$ *if either*

1. $L_x^{[a]} \xrightarrow{\text{obs}} L_y^{[b]}$ *or*
2. $L_x^{[a]} \not\xrightarrow{\text{obs}} L_y^{[b]}$, $L_y^{[b]} \not\xrightarrow{\text{obs}} L_x^{[a]}$, *and* $\ell_x^{[a]} \precdot \ell_y^{[b]}$.

Since with every $L_x^{[a]}$ there is an associated label $\ell_x^{[a]}$, $\Longrightarrow$ can be seen as a "lexicographical" order on pairs $(L_x^{[a]}, \ell_x^{[a]})$. The first element in the pair is ordered by $\xrightarrow{\text{obs}}$, a partial order that is consistent with the ordering $\longrightarrow$. (If $L_x^{[a]} \longrightarrow L_y^{[b]}$, then in $L_y^{[b]}$, y read $\ell_x^{[a]}$ or a later label.) The second element is ordered by $\precdot$, an irreflexive and antisymmetric relation. Parts of the rather involved reasoning as to why the "static" relation $\precdot$ on the labels completes the "dynamic" partial order $\xrightarrow{\text{obs}}$ to a total order on all labeling operation executions are provided in section 5.9. The main difficulty is in establishing transitivity. The intuition as to why $\Longrightarrow$ is transitive is based on the fact that "at any point in time," the current labels of all processes are totally ordered, that is, no three labels are on three different supernodes of a cycle in any T^k subgraph. The reader is encouraged to try to bring about a scenario where there are three labeling operation executions such that

$$L_x^{[a]} \Longrightarrow L_y^{[b]} \Longrightarrow L_z^{[c]} \Longrightarrow L_x^{[a]}$$

while keeping in mind that $\xrightarrow{\text{obs}}$ is transitive. It will become clear that this requires that at some point in time, there will be three labels of x, y, and z on three different supernodes of a cycle in some T^k subgraph, a contradiction.

4.4. The scan operation. The scan operation returns a pair $(\bar{\ell}, \prec)$. In the scan operation of the unbounded label implementation, the linearization order among the labeling operation executions can be determined just by reading the labels since the order among any two operations is just the order among their associated labels. However, as Example 4.4 shows, if labels are taken from a bounded range (and therefore the same labels are repeatedly used), a process scanning the labels concurrently with ongoing labeling operations cannot deduce the order $\Longrightarrow$ from the order of the labels alone.

Example 4.4. In Figure 10, segments represent operation-execution intervals, where time runs from left to right. Two processes i and j perform labeling operations sequentially, j followed by i, followed by many labelings, until eventually the labels are used, and j, for example, uses the same label as before. A third process z performs a scan concurrently with the labelings, reading $label_i$ and then $label_j$. $S1$ and $S2$

represent possible executions of this same scan, the only difference being that many labeling operations of other processes occurred between the reads in $S2$. In both the case where the scan is of the form $S1$ and the case where it is of the form $S2$, the values collected are $label_i = 2$ and $label_j = 1$, where the order among the labels is, say, $1 < 2$. However, in the case of $S1$, j's labeling preceded i's, while in $S2$, i's labeling preceded j's. Thus the order of the labels is not the order among the labeling operations. □

However, we do wish to provide the exact form of solution as in the unbounded case, where just by reading the labels, the scanning process can return a set of labels and the order among them. From Example 4.4, it should be clear that the order $\prec$ returned by the scan cannot be the order $\Longrightarrow$ among the associated labels of labeled-values in $\bar{\ell}$. Nevertheless, the requirement of property P1b is that $\prec$ be consistent with $\Longrightarrow$ for the set of labeling operation executions of labeled-values in $\bar{\ell}$. The key to the solution is to perform many collections of labels and then, based on the properties proven in what follows, return n of them for which $\prec$ can be determined.

The scan algorithm thus consists of two main steps, a sequence of $8n \log n$ *collect* operations[10] and an analysis phase of the collected labels to select a set $\bar{\ell}$ and an order $\prec$.

The $8n \log n$ *collect* operations are logically divided into n phases, where each phase consists of $\log n$ levels, each of eight collects. We use the notation $\ell^{c,m,k}, c \in \{1..8\}$, $m \in \{1..\lceil \log n \rceil\}$, and $k \in \{1..n\}$, to denote variables, each holding a set of labels $\{\ell_1^{c,m,k}, \ldots, \ell_n^{c,m,k}\}$ collected in the cth *collect* operation execution of the mth *level* of the kth *phase*. Let $half(r)$ and $other\_half(r)$ be complementary functions that, for a given set r, return two disjoint subsets $r1$ and $r2$ such that $r1 \cup r2 = r$ and $-1 \leq ||r1|| - ||r2|| \leq 1$.

The scan algorithm, presented in Figure 11, returns the indexed set of labeled-values $\bar{\ell}$, one of each process, and an ordering $\prec$ on their indexes. This order is represented by the vector $O[1..n]$, holding a *permutation* of the indexes in $\{1..n\}$, the number in the ith position representing the ith largest element in the order. The scan algorithm begins with a sequence of $8n\lceil \log n \rceil$ collect operation executions, for which the returned labels are all saved in the variables $\ell^{c,m,k}, c \in \{1..8\}$, $m \in \{1..\lceil \log n \rceil\}$, and $k \in \{1..n\}$. The remainder of the algorithm defines how to choose n of these labels, one per process, for which $\prec$ (i.e., $\Longrightarrow$) can be established. The following is an outline of how this selection process is performed. A formal proof of its correctness can be found in section 5.

By the order of label collection, the labels read in phase $k = 1$ are the earliest to have been collected and those for $k = n$ the last. Notice that from the $8\lceil \log n \rceil$ collected label sets of each phase, the algorithm selects one label. The selected label in the kth phase will be the k largest in the order $\prec$. As it turns out, in order to be able to show that it is the kth largest, it suffices that the following condition holds (slightly abusing notation in the definition).

Condition 1. For the label $\ell_s^{8,\lceil \log n \rceil,k}$, collected in the $\lceil \log n \rceil$th level of the kth phase, and any label $\ell_y^{8,1,k}$ of a process $y \in R$, collected in the first level of the kth phase, it is the case that $L_y^{8,1,k} \Longrightarrow L_s^{8,\lceil \log n \rceil,k}$.

To prove that this condition suffices, let it be shown that if it is maintained, the labeling operation execution of a label returned in a phase $k' < k$ precedes (in the

[10] Note that, as mentioned in section 1, the *scan* algorithm requires a scanning process *only to read* other's labels and does not require it to write.

```
function scan;
    function select(m,k,r);
    begin
        if ||r|| = 1 then return (x : x ∈ r);
          else
            x := select(m−1, k, half(r));
            y := select(m−1, k, other_half(r));
            if (∃c1, c2 ∈ {1..8}) (c1 < c2) ∧ (ℓ_x^{c1,m,k} ⪯ ℓ_y^{c2,m,k})
              then return y
              else return x
        fi fi;
    end select;
begin
    R := {1..n};
    ℓ̄ := ∅;
    for k := 1 to n do
        for m := 1 to ⌈log n⌉ do
            for c := 1 to 8 do
                ℓ^{c,m,k} := collect
    od od od;
    for k := n downto 1 do
        s := select(⌈log n⌉, k, R);
        ℓ̄ := ℓ̄ ∪ {value(ℓ_s^{8,⌈log n⌉,k})};
        O[s] := k;
        R := R − {s};
    od;
    return (ℓ̄, O);
end scan;
```

FIG. 11. *The scan algorithm.*

ordering $\Longrightarrow$) that of the label returned in phase k. The following shows that this is the case for the labels $\ell_x^{8,\lceil \log n\rceil,k}$, $\ell_y^{8,\lceil \log n\rceil,k-1}$, and $\ell_z^{8,\lceil \log n\rceil,k-2}$ returned in phases k, $k-1$, and $k-2$, respectively. The same line of proof can be extended inductively to all $k' < k$.

By Condition 1, $L_y^{8,1,k} \Longrightarrow L_x^{8,\lceil \log n\rceil,k}$. Since the read of $\ell_y^{8,1,k}$ was performed after that of $\ell_y^{8,\lceil \log n\rceil,k-1}$, either the label of the same labeling operation execution was read in both cases or $L_y^{8,\lceil \log n\rceil,k-1} \Longrightarrow L_x^{8,\lceil \log n\rceil,k}$. By similar reasoning, $L_z^{8,\lceil \log n\rceil,k-2} \Longrightarrow L_y^{8,\lceil \log n\rceil,k-1}$, which by the transitivity of $\Longrightarrow$ establishes $L_z^{8,\lceil \log n\rceil,k-2} \Longrightarrow L_x^{8,\lceil \log n\rceil,k}$.

It remains to be shown that the label returned in any phase, determined by the *select* function, meets Condition 1. The *select* function is a recursively defined "winner-take-all"-type algorithm among the processes in R. In any given phase, R is the set of processes for which a label has not been selected in earlier phases. The *select* function returns the id of the "winner," a process s that meets Condition 1. At any level m of the application of $select(m, k, r)$, the winners of the selections at level $m-1$ are paired up, and from each pair one "winner" process is selected to be passed

on to the $(m+1)$th level of selection. After at most $\lceil \log ||R|| \rceil$ levels, s, the winner of all selections, is returned.

Based on the definition of the *select* function, maintaining the following condition two suffices to assure that the label of the process s returned by $select(m, k, r)$ meets Condition 1.

Condition 2. Of the two processes x and y in the application of *select* at level m of phase k, the one returned, say x, is such that $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$, where $\ell_y^{1,m,k}$ and $\ell_x^{8,m,k}$, respectively, are the labels associated with these labeling operation executions.

Maintaining Condition 2 suffices for the following reason. If at level m process x was selected between x and y and at level $m-1$ process y was selected between y and z, by the same line of proof as above, from $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$ and $L_z^{1,m-1,k} \Longrightarrow L_y^{8,m-1,k}$, it follows that $L_z^{8,m-2,k} \Longrightarrow L_x^{8,m,k}$. By induction, this implies Condition 1.

It remains to be shown that Condition 2 can be met. Recall Example 4.4, which implies that it is impossible to establish the order $\Longrightarrow$ among two labeling operation executions from the order among their associated labels alone. To overcome this problem, instead of attempting to decide the order between two given labeling operation executions, the algorithm will choose a pair out of several given labeling operation executions for which the order $\Longrightarrow$ can be determined. Thus to allow the *select* operation at level m of phase k to choose a "winner" process, say x, for which $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$, labels of x and y from eight consecutive collects will be analyzed.

Let it first be shown that if the following condition holds for y, namely, if it is the case that

$$[Condition\ 3.] \qquad (\exists c1, c2 \in \{1..8\})\,(c1 < c2) \wedge (\ell_x^{c1,m,k} \not\preceq \ell_y^{c2,m,k}),$$

then $L_x^{c1,m,k} \Longrightarrow L_y^{c2,m,k}$. (Because of the order of label collecting, this will imply $L_x^{1,m,k} \Longrightarrow L_y^{8,m,k}$.) Assume by way of contradiction that $L_x^{c1,m,k} \Longrightarrow L_y^{c2,m,k}$. Since $\ell_x^{c1,m,k} \not\preceq \ell_y^{c2,m,k}$, it must be by the definition of $\Longrightarrow$ that $L_y^{c2,m,k} \xrightarrow{obs} L_x^{c1,m,k}$. It cannot be that $\ell_y^{c2,m,k} \in max\_obs(L_x^{c1,m,k})$ since by the properties of the labeling scheme, for the label $\ell_y^{[b]} \in max\_obs(L_x^{c1,m,k})$, $\ell_y^{[b]} \not\preceq \ell_x^{c1,m,k}$. Thus there must be a different labeling operation execution $\ell_y^{[b]} \in max\_obs(L_x^{c1,m,k})$, $L_y^{c2,m,k} \longrightarrow L_y^{[b]}$. This label $\ell_y^{[b]}$ was already observed (i.e., must have been written) before the end of the read of $\ell_x^{c1,m,k}$. Thus $\ell_y^{[b]}$ or a label later than it must have been read instead of $\ell_y^{c2,m,k}$ in the *collect* $c2$ of level m in phase k, a contradiction.

It remains to be shown that if Condition 3 does not hold for y, it is the case that $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$, and x can be correctly returned. Assume by way of contradiction that Condition 3 does not hold for y. By the same arguments as above, it cannot be that Condition 3 holds for x, that is, $(\exists c1, c2 \in \{1..8\})\ (c1 < c2) \wedge (\ell_y^{c1,m,k} \not\preceq \ell_x^{c2,m,k})$. Therefore, it must be that there are four nonconsecutive collects of $\ell^{\,c1,m,k}$, $c1 \in \{1,3,5,7\}$, and four nonconsecutive collects of $\ell^{\,c2,m,k}$, $c2 \in \{2,4,6,8\}$, such that the labels $\ell_y^{c1,m,k}$, $c1 \in \{1,3,5,7\}$, are all different from one another and the labels $\ell_x^{c2,m,k}$, $c2 \in \{2,4,6,8\}$, are all different from one another. The reason is that if any two of them, say $\ell_y^{3,m,k}$ and $\ell_y^{5,m,k}$, are the same, then in order for Condition 3 not to hold for x $c1 = 4$ and $c2 = 3$, it must be that $\ell_x^{4,m,k} \not\preceq \ell_y^{3,m,k}$. However, since $\ell_y^{3,m,k}$ and $\ell_y^{5,m,k}$ are the same, it would follow that $\ell_x^{4,m,k} \not\preceq \ell_y^{5,m,k}$, and Condition 3 would hold for y, a contradiction.

To complete the proof, it remains to be shown that if the labels $\ell_y^{c1,m,k}$, $c1 \in \{1,3,5,7\}$, are all different from one another and the labels $\ell_x^{c2,m,k}$, $c2 \in \{2,4,6,8\}$,

are all different from one another, then $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$. The situation above is such that during the eight collect operations, each of the processes x and y executed a new labeling operation at least three times. It can be formally shown[11] that after x and y moved at least three times, the third new labeling operation execution $L_x^{8,m,k}$ occurred completely after the initial labeling of y, that is, after $L_y^{1,m,k} \longrightarrow L_x^{8,m,k}$ (see Figure 13 in section 5.8). The scan thus takes $O(n^2 \log n)$ read operations.

As a final comment, note that for algorithms where only the maximum label is required and not a complete order among all returned labels (as in the construction of an MRMW atomic register or solutions to the *mutual exclusion* problem), only one phase of label collection is required, that is, only $8 \log n$ collects.[12]

5. Correctness proof.

5.1. A short review of Lamport's formal theory. This is a minimal outline (due to Ben-David [Ben88]) of Lamport's formalism, on which the correctness proof in this chapter is based. The reader is encouraged to consult [Lam86c, Lam86d, Lam86a, Lam86b] for an elaborate presentation and discussion.

Lamport bases his formal theory on two abstract relations over operation executions. For operation executions A and B, "$A \longrightarrow B$" stands for "A precedes B" and "$A \dashrightarrow B$" stands for "A can causally affect B."

A *system execution* is a triple $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$, where φ is a set of *operation executions* and $\longrightarrow$ and $\dashrightarrow$ are binary relations over φ. Lamport offers the following axioms:

A1. $\longrightarrow$ is an irreflexive transitive relation.

A2. If $A \longrightarrow B$, then $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

A3. If ($A \longrightarrow B$ and $B \dashrightarrow C$) or ($A \dashrightarrow B$ and $B \longrightarrow C$), then $A \dashrightarrow C$.

A4. If $A \longrightarrow B \dashrightarrow C \longrightarrow D$, then $A \longrightarrow D$.

A5. For any A, the set of B such that $A \not\longrightarrow B$ is finite.

An intuition for these axioms can be gained by considering the following model for it. Let $\mathcal{E}$ be a partially ordered set of events and let φ be a collection of nonempty subsets of $\mathcal{E}$. For A and B in φ, define $A \longrightarrow B$ if and only if $(\forall a \in A)\ (\forall b \in B)\ (a < b)$ (in the sense of $\mathcal{E}$) and $A \dashrightarrow B$ if and only if $(\exists a \in A)\ (\exists b \in B)\ (a < b)$. A straightforward checking shows that such models satisfy axioms A1–A4 and also the following axiom:

A4*. If $A \dashrightarrow B \longrightarrow C \dashrightarrow D$, then $A \dashrightarrow D$.

This last axiom was suggested by Abraham[13] in [AB87], where a completeness theorem was proven for the above-mentioned class of models with respect to axioms {A1, A2, A3, A4, A4*}. An important class of models is obtained when $\mathcal{E}$ is a linear (total) ordering. In such a case, the system satisfies an additional axiom:

Global time. For all A and B, it is the case that either $A \dashrightarrow B$ or $B \longrightarrow A$ but not both.

The above axioms can be extended to nonterminating operation executions as described in [Lam86c]. Added on top of these axioms are the communication axioms, in our case axioms B0–B5 of [Lam86d], for communication via shared registers. These axioms formalize the behavior of a single-writer multireader atomic register. In a few words, axioms B0–B4 define what constitutes *regular* register behavior, namely, that reads can return only values that

[11] This claim is *not* true if fewer than three new labelings took place.

[12] The number of collects in each phase can be lowered to $5 \log n$ if one gives up the property that the order of reads in a collect be arbitrary.

[13] Ben-David was later informed that this result was obtained independently by Anger.

- were actually written,
- were written before the end of the read, and
- were not overwritten before the beginning of read.

Axiom B5 is added to these, which restricts the allowed behavior of the register by requiring that reads and writes be linearizable. Such a register that abides by axioms B0–B5 is called *atomic* since, in effect, its behavior is equivalent to one in which all reads and writes are "atomic," that is, occur in nonoverlapping intervals of time.

5.2. Proof outline. The proof will follow Definition 8 of [Lam86a], namely, that a system **S** *implements* a system **H** if there is a mapping $m : \mathbf{S} \mapsto \mathbf{H}$ such that for every system execution $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$ in **S**, $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$ implements $m(\langle \varphi, \longrightarrow, \dashrightarrow \rangle)$. The definition of a system execution used in what follows is that of [Lam86a] under the assumption of global time. Theorem 5.1 below establishes the correctness of the implementation.

THEOREM 5.1. *The system defined by the labeling and scan procedures implements a concurrent time-stamp system.*

In order to prove the theorem correct, the systems involved need to be formally defined and a mapping between them must be established.

5.3. System definitions. The labeling and scan procedures of the previous sections define a system **S**, the set of all system executions that consist of reads and writes of the single-writer multireader atomic registers $v_1, \ldots, v_n$, such that the only operations on these registers are the ones indicated by the scan and labeling algorithms. Formally, **S** contains all system executions $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$ such that we have the following:

1. φ consists of reads and writes of single-writer multireader atomic registers $v_1, \ldots, v_n$ (with register axioms B0–B5 restricting such read and write operations [Lam86b]).

2. Each v_x is written by process x and read by all processes in $\{1..n\}$, where $r_x^{[k]}(y)$ ($w_x^{[k]}(x)$) denote the kth read (respectively, write) of v_x by process y (respectively, x).

3. The read and write operation executions of a process x are totally ordered by $\longrightarrow$.

4. For any process z and any x and y:

(a) If the read operation $r_x^{[k]}(z)$ occurs, then $r_y^{[k]}(z)$ occurs, $r_x^{[k-1]}(z) \longrightarrow r_y^{[k]}(z)$, and if for some $w_z^{[k']}(z)$, $r_x^{[k]}(z) \longrightarrow w_z^{[k']}(z)$, then $r_y^{[k]}(z) \longrightarrow w_z^{[k']}(z)$.

(b) For any two writes $w_z^{[k]}(z)$ and $w_z^{[k+1]}(z)$, there exists a set of read operation executions

$$\mathcal{R}_{k+1} = \{r_x^{[\alpha]}(z) \,|\, w_z^{[k]}(z) \longrightarrow r_x^{[\alpha]}(z) \longrightarrow w_z^{[k+1]}(z),\ \alpha \in \{0, 1, \ldots\}\},$$

of reads of v_x such that $||\mathcal{R}_{k+1}|| \bmod (8n\lceil \log n \rceil) = 1$.

(c) For every $r_x^{[k]}(z)$, $r_x^{[k]}(z) \in \mathcal{R}_r$, for some r.

This fourth condition formalizes some of the semantics of labeling and scan procedures. It states that every read is part of a *collect* operation consisting of a sequence of reads, one of each register, each collection ending before the next begins, and that reads and writes are bunched in groups of either $8n \log n$ collects or a collect followed immediately by a write.

The following is a formal definition of **H**, a concurrent time-stamp system.

DEFINITION 5.1. *A* concurrent time-stamp system *is a set of system executions* $\langle \psi, \longrightarrow, \dashrightarrow \rangle$ *that have properties* P0–P4.

Properties P1–P4 are as defined earlier, and the following is the definition of P0.

P0. The set of operation executions on the CTSS is the set $\psi = \bigcup_i \psi_i$, where each ψ_i, the set of operation executions by process i, is as follows:

- *A finite or infinite set of labeling operation executions* $\{L_i^{[1]}, L_i^{[2]}, \ldots\}$: A unique *labeled-value* $\ell_i^{[k]}$ is associated with each $L_i^{[k]}$. The set of possible labeled-values can be from any range. For example, if an atomic register is to be implemented, the labeled-value can be the value written to the register. Given that the value $\ell_i^{[k]}$ may repeatedly appear, in order that a unique labeled-value be associated with each $L_i^{[k]}$, let $\ell_i^{[k]}$ be the triplet $\langle \ell_i^{[k]}, i, k\rangle$, where i and k are dummy fields and only $\ell_i^{[k]}$ is visible to the user. There is thus a one-to-one mapping from labeled-values to labeling operations.
- *A finite or infinite set of scan operation executions* $\{S_i^{[1]}, S_i^{[2]}, \ldots\}$: A *view* $\bar{\ell} = \{\ell_1^{[k_1]}, \ldots, \ell_n^{[k_n]}\}$ is returned by each scan, with different labeled-values associated with labeling operation executions of different processes.
- *An initial labeling operation execution* $L_i^{[0]}$ *with labeled-value* $\ell_i^{[0]}$.

The initial labeling $L_i^{[0]} \longrightarrow S_j^{[k]}$ for any i, j, and k. (This is the same as assuming that there is some initial labeled-value for any process i that a scan will obtain if it preceded any labeling operation of i.) All operation executions in ψ_i are totally ordered by $\longrightarrow$, that is, they occur sequentially.

5.4. The mapping. By Definition 8 of [Lam86a], to show that the labeling and scan procedures implement a CTSS, a mapping m from **S** to **H** must be defined. In the definition of the labeling and scan procedures, for each system execution $\langle \varphi, \longrightarrow, \dashrightarrow\rangle$ of **S**, the set of operation executions $m(\varphi)$ of $m(\langle \varphi, \longrightarrow, \dashrightarrow\rangle)$ is the following *higher-level view* of $\langle \varphi, \longrightarrow, \dashrightarrow\rangle$:

1. Each *labeling* operation execution $L_i^{[k]}$ consists of a set $r_1^{[k']}(i), \ldots, r_n^{[k']}(i)$ of reads followed by a write $w_i^{[k]}(i)$, where $k' = \max\{\alpha \mid r_j^{[\alpha]}(i) \longrightarrow w_i^{[k]}(i)\}$.
2. Each *scan* operation execution by process i is a set of reads

$$\{r_j^{[\alpha]}(i) \mid j = 1..n,\ \alpha = k..k + 8n\lceil \log n\rceil \text{ and}$$
$$(\neg\exists w_i^{[k']}(i),\ r_j^{[\alpha]}(i) \longrightarrow w_i^{[k']}(i) \longrightarrow r_j^{[\alpha+1]}(i))\},$$

all in φ and no element of which is part of another *scan* or *labeling*.

The set $m(\varphi)$ meets conditions H1 and H2 of Definition 4 of [Lam86a], that is, each of its elements is a finite and nonempty set of elements of φ and each element of φ belongs to a finite, nonzero number of elements of $m(\varphi)$. It is thus a higher-level view of φ. (In fact, this implies that the labeling and scan operations as implemented are wait-free since waiting means that a higher-level operation takes an infinite number of lower-level ones.) To complete the description of the mapping m, the precedence relations $\xrightarrow{H}$ and $\dashrightarrow^{H}$ must be defined so that $m(\langle \varphi, \longrightarrow, \dashrightarrow\rangle)$ is defined as $\langle m(\varphi), \xrightarrow{H}, \dashrightarrow^{H}\rangle$.

By choosing $\xrightarrow{H}$ and $\dashrightarrow^{H}$ to be the induced relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$ as defined by equation 2 of [Lam86a] (by equation 2, choosing the induced precedence relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$ for $\xrightarrow{H}$ and $\dashrightarrow^{H}$ simply means that the ordering among the higher-level *scan* and *labeling* operation executions is that of the reads and writes implementing them), axioms A1–A5 are met, implying that $\langle m(\varphi), \xrightarrow{H}, \dashrightarrow^{H}\rangle$ is indeed a system execution. Since condition H3 of Definition 5 of [Lam86a] is satisfied by the induced precedence

relations,[14] $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$ implements $\langle m(\varphi), \xrightarrow{H}, \dashrightarrow^{H} \rangle$.

Having defined the system $\langle m(\varphi), \xrightarrow{H}, \dashrightarrow^{H} \rangle$, it remains to be shown that it is indeed a CTSS, that is, is in **H**. This amounts to showing that $\langle m(\varphi), \xrightarrow{*}, \dashrightarrow^{*} \rangle$ satisfies properties P0–P4.

5.5. Properties P0 and P2–P4. The proof that $\langle m(\varphi), \xrightarrow{*}, \dashrightarrow^{*} \rangle$ meets property P0 follows by applying equation 2 of [Lam86a] to $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$ (again, this amounts to defining the high-level order among *scan* and *labeling* operation executions to be that among the reads and writes implementing them) and observing the following:

1. The labeled-value $\ell_i^{[k]}$ associated with each labeling operation $L_i^{[k]}$ is just the labeled-value part written to v_i by the write $w_i^{[k]}(i)$ of process i. (Recall that there is also a label part of v_i.)

2. Any labeled-value returned by a scan is the result of some write $w_i^{[k]}(i)$.

3. The initial labeling $L_i^{[0]}$ is the write of some initial labeled-value and label 11..1 to register v_i.

The proof that $\bar{\ell}$ and $\succ$ are a *view* and an *irreflexive total order* on its elements follows from the definition of the *scan procedure*. Since v_i, $i \in \{1..n\}$ are SWMR atomic registers, applying equation 2 of [Lam86a] together with register axioms B0–B5 to $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$ yields the proof that $\langle m(\varphi), \xrightarrow{*}, \dashrightarrow^{*} \rangle$ satisfies properties P2, P3, and P4. The details are left to the reader.

To simplify the presentation, for the remainder of this section, we use the notation $\ell_i^{[k]}$ to denote the label part of the value written to register v_i in the labeling operation execution $L_i^{[k]}$. We will use the notation $value(\ell_i^{[k]})$ to refer to the labeled-value part.

5.6. Properties of the observed relation. As part of the notation used in what follows, $r_j(L_i^{[k]})$ and $w(L_i^{[k]})$ will denote, respectively, the reading of v_j and writing of v_i during a labeling operation execution $L_i^{[k]}$. Also, let $m(\varphi)^L \subseteq m(\varphi)$ denote the set of all labeling operation executions in $m(\varphi)$. To prove that $\langle m(\varphi), \xrightarrow{*}, \dashrightarrow^{*} \rangle$ meets property P1, the relation $\Longrightarrow$ on the labeling operation executions in $m(\varphi)$ should be shown to be an irreflexive total order. The definition of this relation (Definition 4.3) is based on that of the relation $\dashrightarrow^{obs}$ (Definition 4.1).

The following lemma establishes the properties of $\dashrightarrow^{obs}$, later used to establish the properties of $\Longrightarrow$.

LEMMA 5.1. *The relation $\dashrightarrow^{obs}$ is an irreflexive partial order on the labeling operation executions in $m(\varphi)$, such that for any two labeling operations $L_i^{[a]}$ and $L_j^{[b]}$, if $L_i^{[a]} \xrightarrow{*} L_j^{[b]}$, then $L_i^{[a]} \dashrightarrow^{obs} L_j^{[b]}$.*

Proof. Since $r_j(L_i^{[a]}) \longrightarrow w_i(L_i^{[a]})$ for any j, it follows that $\dashrightarrow^{obs}$ is irreflexive. The rest of the proof is based on the three claims below.

CLAIM 5.1.1 (transitive). *For any three labeling operation executions $L_i^{[a]}$, $L_j^{[b]}$, and $L_k^{[c]}$, if $L_i^{[a]} \dashrightarrow^{obs} L_j^{[b]} \dashrightarrow^{obs} L_k^{[c]}$, then $L_i^{[a]} \dashrightarrow^{obs} L_k^{[c]}$.*

Proof. The proof is by induction on the length of the minimal production sequence of the production of $L_j^{[b]} \dashrightarrow^{obs} L_k^{[c]}$. If $||L_j^{[b]} \dashrightarrow^{obs} L_k^{[c]}|| = 1$, then by definition $r_j(L_k^{[c]}) = \ell_j^{[b]}$ and $L_i^{[a]} \dashrightarrow^{obs} L_j^{[b]}$, which by the definition of $\dashrightarrow^{obs}$ implies that

[14] Definition 5 of [Lam86a] states that a lower-level system execution $\langle \varphi, \longrightarrow, \dashrightarrow \rangle$ implements a higher-level one $\langle \psi, \xrightarrow{H}, \dashrightarrow^{H} \rangle$ if ψ is a higher-level view of φ and condition H3 holds, that is, for any G, $H \in \psi$, if $G \xrightarrow{*} H$, then $G \xrightarrow{H} H$.

$L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_k^{[c]}$. Assume that the induction hypothesis holds for every $r' < r$. Let $||L_j^{[b]} \overset{\text{obs}}{\dashrightarrow} L_k^{[c]}|| = r$. By definition, there exists an $L_j^{[b']}$ such that $L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b']}$, where $||L_j^{[b]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b']}|| = r-1$ and $r_j(L_k^{[c]}) = \ell_j^{[b']}$. By the induction hypothesis, $L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b']}$, which by definition implies that $L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_k^{[c]}$. □

COROLLARY 5.1. *If $L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b]}$, then there exists a read $r_i(L_\alpha^{[\beta]}) = \ell_i^{[a]}$ such that $w(L_i^{[a]}) \dashrightarrow r_i(L_\alpha^{[\beta]}) \longrightarrow w(L_j^{[b]})$ for some α and β (possibly $\alpha = j$ and $\beta = b$).*

Proof. The proof is by induction on the length of the minimal production of the observation sequence, as in the previous claim. If $||L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b]}|| = 1$, then by definition $\alpha = j$ and $\beta = b$. Assume that the induction hypothesis holds for every $r' < r$. By the minimality of the production sequence, there must exist an $L_\alpha^{[\beta]}$, $r_i(L_\alpha^{[\beta]}) = \ell_i^{[a]}$, such that $||L_\alpha^{[\beta]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b]}|| = r-1$. By the induction hypothesis, there exist α' and β' such that

$$w(L_\alpha^{[\beta]}) \dashrightarrow r_\alpha(L_{\alpha'}^{[\beta']}) \longrightarrow w(L_j^{[b]}),$$

where possibly $\alpha' = j$ and $\beta' = b$. By the definition of the labeling operation, $r_i(L_\alpha^{[\beta]}) \longrightarrow w(L_\alpha^{[\beta]})$, and so

$$r_i(L_\alpha^{[\beta]}) \longrightarrow w(L_\alpha^{[\beta]}) \dashrightarrow r_\alpha(L_{\alpha'}^{[\beta']}) \longrightarrow w(L_j^{[b]}),$$

which by axiom A4 implies $r_i(L_\alpha^{[\beta]}) \longrightarrow w(L_j^{[b]})$. Since $r_i(L_\alpha^{[\beta]}) = \ell_i^{[a]}$, it follows by atomic register axiom B5 that $w(L_i^{[a]}) \dashrightarrow r_i(L_\alpha^{[\beta]}) \longrightarrow w(L_j^{[b]})$. □

CLAIM 5.1.2 (antisymmetric). *For any two distinct labeling operation executions $L_i^{[a]}$ and $L_j^{[b]}$, if $L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b]}$, then $L_j^{[b]} \overset{\text{obs}}{\not\dashrightarrow} L_i^{[a]}$.*

Proof. Assume by way of contradiction that $L_i^{[a]} \overset{\text{obs}}{\dashrightarrow} L_j^{[b]}$ and $L_j^{[b]} \overset{\text{obs}}{\dashrightarrow} L_i^{[a]}$. Thus by Corollary 5.1, for some α, β, γ, and δ (possibly $\alpha = j$, $\beta = b$, $\gamma = i$, or $\delta = a$),

$$w(L_i^{[a]}) \dashrightarrow r_i(L_\alpha^{[\beta]}) \longrightarrow w(L_j^{[b]}) \text{ and}$$
$$w(L_j^{[b]}) \dashrightarrow r_j(L_\gamma^{[\delta]}) \longrightarrow w(L_i^{[a]}).$$

Since this implies

$$w(L_i^{[a]}) \dashrightarrow r_i(L_\alpha^{[\beta]}) \longrightarrow w(L_j^{[b]}) \dashrightarrow r_j(L_\gamma^{[\delta]}),$$

by axiom A4*, $w(L_i^{[a]}) \dashrightarrow r_j(L_\gamma^{[\delta]})$. By $w(L_i^{[a]}) \dashrightarrow r_j(L_\gamma^{[\delta]})$ and $r_j(L_\gamma^{[\delta]}) \longrightarrow w(L_i^{[a]})$, a contradiction to the axiom of global time is derived. □

CLAIM 5.1.3 (consistent). *If $L_x^{[a]} \overset{*}{\longrightarrow} L_y^{[b]}$, then $L_x^{[a]} \overset{\text{obs}}{\dashrightarrow} L_y^{[b]}$.*

Proof. If $x = y$, then $r_x(L_y^{[a+1]}) = \ell_x^{[a]}$, and by induction, for $b > a$, $L_x^{[a]} \overset{\text{obs}}{\dashrightarrow} L_y^{[b]}$. If $x \neq y$, then by register axioms B0–B4, since $w(L_x^{[a]}) \longrightarrow r_x(L_y^{[b]})$, either $r_x(L_y^{[b]}) = \ell_x^{[a]}$ (implying $L_x^{[a]} \overset{\text{obs}}{\dashrightarrow} L_y^{[b]}$) or there exists an $L_x^{[a']}$, $L_x^{[a]} \overset{*}{\longrightarrow} L_x^{[a']}$, where $r_x(L_y^{[b]}) = \ell_x^{[a']}$, which by the transitivity of $\overset{\text{obs}}{\dashrightarrow}$ (Claim 5.1.1) implies $L_x^{[a]} \overset{\text{obs}}{\dashrightarrow} L_y^{[b]}$. This completes the proof of Lemma 5.1. □

The following lemma formalizes the property that whenever a new label $\ell_x^{[a]}$ is selected, it is greater (by the ordering $\not\prec$) then the latest label observed in $L_x^{[a]}$ for any process $y \neq x$.

LEMMA 5.2. *For any labeling operation execution $L_x^{[a]}$, it is the case that*

$$(\forall \ell_y^{[b]} \in max\_obs(L_x^{[a]}))\,(\ell_y^{[b]} \not\prec \ell_x^{[a]}).$$

To simplify the exposition, the the proof is differed to section 5.9, where it is joined with the proof of Claim 5.3.2.

5.7. Property P1a. The following lemma asserts that $\Longrightarrow$ meets part a of property P1.

LEMMA 5.3. *The relation $\Longrightarrow$ is an irreflexive total order on the labeling operation executions in $m(\varphi)$ such that for any two labeling operations $L_i^{[a]}$ and $L_j^{[b]}$, if $L_i^{[a]} \xrightarrow{*} L_j^{[b]}$, then $L_i^{[a]} \Longrightarrow L_j^{[b]}$.*

Proof. By Definition 4.3, the relation $\Longrightarrow$ is irreflexive and total, and is consistent with the ordering $\xrightarrow{*}$ among the labeling operation executions in $m(\varphi)$. The following two claims complete the proof by showing that it is also antisymmetric and transitive.

CLAIM 5.3.1 (antisymmetric). *For any two distinct labeling operation executions $L_i^{[a]}$ and $L_j^{[b]}$, if $L_i^{[a]} \Longrightarrow L_j^{[b]}$, then $L_i^{[a]} \not\Longleftarrow L_j^{[b]}$.*

Proof. Assume by way of contradiction that for two distinct labeling operation executions, $L_i^{[a]} \Longrightarrow L_j^{[b]}$ and $L_i^{[a]} \Longleftarrow L_j^{[b]}$. Since $\xrightarrow{obs}$ is antisymmetric, it is not the case that both $L_i^{[a]} \xrightarrow{obs} L_j^{[b]}$ and $L_j^{[b]} \xrightarrow{obs} L_i^{[a]}$ hold. Thus if $L_i^{[a]} \xrightarrow{obs} L_j^{[b]}$, $L_j^{[b]} \not\Longrightarrow L_i^{[a]}$ even if $\ell_j^{[b]} \not\prec \ell_i^{[a]}$, a contradiction. Thus it must be the case that $\ell_j^{[b]} \not\prec \ell_i^{[a]}$ and $\ell_i^{[a]} \not\prec \ell_j^{[b]}$, which contradicts the definition of the ordering $\not\prec$ of the labels. □

CLAIM 5.3.2 (transitive). *For any three labeling operation executions $L_i^{[a]}$, $L_j^{[b]}$, and $L_k^{[c]}$, $L_i^{[a]} \Longrightarrow L_j^{[b]} \Longrightarrow L_k^{[c]}$ implies $L_i^{[a]} \Longrightarrow L_k^{[c]}$.*

Due to its extreme length and to simplify the presentation, the proof is deferred to section 5.9.

This completes the proof of Lemma 5.3. □

5.8. Property P1b. It remains to be proven that $\langle m(\varphi), \xrightarrow{*}, \dashrightarrow^{*} \rangle$ meets part b of property P1, that is, for any scan operation execution $S_i^{[k]}$ that returns $(\bar{\ell}, \prec)$, where

$$value(\ell_x^{[a]}),\ value(\ell_y^{[b]}) \in \bar{\ell},$$

it is the case that $x \prec y$ if and only if $L_x^{[a]} \Longrightarrow L_y^{[b]}$. Since both $\prec$ and $\Longrightarrow$ are irreflexive total orders, it suffices to show the "only if" direction. By the definition of the scan implementation, the returned order $\prec$ among the indexes of labeled-values in $\bar{\ell}$ is just the ordering among the collection phases in which they were selected. Thus it suffices to prove that in any scan operation execution $S_i^{[k]}$ that returns $(\bar{\ell}, \prec)$, if $value(\ell_x^{[a]})$ was returned in phase k' and $value(\ell_y^{[b]})$ was returned in phase k, where $k' < k$, then $L_x^{[a]} \Longrightarrow L_y^{[b]}$. This is captured by the following lemma (slightly abusing notation).

LEMMA 5.4. *In any scan operation execution, for any i such that $O[i] < O[j]$, where $value(\ell_i^{8,\lceil \log n \rceil, O[i]}), value(\ell_j^{8,\lceil \log n \rceil, O[j]}) \in \bar{\ell}$, it is the case that $L_i^{8,\lceil \log n \rceil, O[i]} \Longrightarrow L_j^{8,\lceil \log n \rceil, O[j]}$.*

Proof. The general outline of the proof is as follows. Recall that a phase of the scan execution consists of $8 \log n$ collect operation executions, where each consecutive

eight of them are called a *level* in the phase. The "first" level is the earliest collected and the "$\log n$th" is the latest. The proof begins with Claim 5.4.1, which states that the relation between two labels in any two collects can be extended to the collects preceding and following them. Then in Claims 5.4.3, 5.4.4, and 5.4.5, it is shown that among the labels of any eight collects in a level of the scan, two labels can be chosen for which the order $\Longrightarrow$ is known. Based on Claim 5.4.1 and the transitivity of $\Longrightarrow$, the results of comparing labels of x and y in one level and y and z in a lower level are extended to relate those of x and z, allowing us to show (Claim 5.4.6) that for any k and R, if s is returned by $select(\lceil \log n \rceil, k, R)$, then $L_i^{1,1,k} \Longrightarrow L_s^{8,\lceil \log n \rceil,k}$ for all $i \in R - \{s\}$. Finally, transitivity is used again to prove Lemma 5.4, that is, that the results of different phases (*select* executions) are comparable and that the order $\Longrightarrow$ among the labels returned is the order of the phases.

To simplify the presentation, in what follows, indexes will be dropped when it is clear from the context what they should be. This will include the index of the process i performing the *scan* or *collect* operation. The notation C_w will denote $C_i^{[w]}$, the wth collect operation execution performed during a given scan. A label associated with $L_x^{[a]}$, read in any C_w, will be denoted by $\ell_{x,w}^{[a]}$ or ℓ_x^w, and the labeling operation execution $L_x^{[a]}$ itself will be similarly denoted by $L_{x,w}^{[a]}$ or L_x^w.

The following claim will be used to assert that the relation between two labels in any two collects can be extended to the collects preceding and following them. More specifically, this claim asserts that if the label of x is ordered before that of y, where x's label was collected in a collect C_{w+1}, earlier than C_{w+2} in which y's was collected, then any label of x collected in collect C_w that precedes collect C_{w+1} must be ordered before that of y and, similarly, any label of y from collect C_{w+3} must be ordered after that of x.

CLAIM 5.4.1. *If* $C_w \xrightarrow{*} C_{w+1} \xrightarrow{*} C_{w+2}$ (*or* $C_{w+1} \xrightarrow{*} C_{w+2} \xrightarrow{*} C_{w+3}$) *and if for some* $\ell_{x,w+1}^{[a]}$ *and* $\ell_{y,w+2}^{[b]}$, $L_x^{[a]} \Longrightarrow L_y^{[b]}$, *then for any* $\ell_{x,w}^{[c]}$ (*similarly,* $\ell_{y,w+3}^{[d]}$), *it is the case that* $L_x^{[c]} \Longrightarrow L_y^{[b]}$ (*similarly,* $L_x^{[a]} \Longrightarrow L_y^{[d]}$).

Proof. For any x, if $a \neq c$, that is, if they are of different labeling operations, then it must be the case that $L_x^{[c]} \xrightarrow{*} L_x^{[a]}$. The reason is that if this were not the case, then since $\ell_{x,w}^{[c]}$ was read in C_w and $L_x^{[a]} \xrightarrow{*} L_x^{[c]}$, by atomic register axiom B5, it could not be the case that $\ell_{x,w+1}^{[a]}$ was read in the later collect C_{w+1}, a contradiction. By Definition 4.3, it is thus the case that $L_x^{[c]} \Longrightarrow L_x^{[a]} \Longrightarrow L_y^{[b]}$, which by transitivity (Claim 5.3.2) implies $L_x^{[c]} \Longrightarrow L_y^{[b]}$. By a similar proof, $L_x^{[a]} \Longrightarrow L_y^{[d]}$. □

CLAIM 5.4.2. *For any eight collect operation executions of level* m *of phase* k *in a given scan operation execution, if the condition*

$$(\exists c1, c2 \in \{1..8\})\ (c1 < c2) \wedge (\ell_x^{c1,m,k} \overset{V}{\prec} \ell_y^{c2,m,k}),$$

holds, then $L_x^{1,m,k} \Longrightarrow L_y^{8,m,k}$, *and otherwise* $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$.

Proof. The following claim (Claim 5.4.3) establishes that there are three complementary conditions (one of the three must always hold) on the labels in the eight collects:

1. There are a label of y and a label of x where the label of y was collected in a later collect than that in which x was collected and where the label of y is greater (by $\prec$) than the label of x.

2. This is the first condition with the roles of x and y reversed.

3. The labels of x and y have each changed at least three times during these eight collect operation executions.

The claims that follow show that if the first condition holds, $L_x^{1,m,k} \Longrightarrow L_y^{8,m,k}$, and if one of the other two holds, then $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$. More formally, we have the following.

CLAIM 5.4.3. *For the* 16 *labels* $\ell_x^{c1,m,k}$ *and* $\ell_y^{c2,m,k}$, $c1, c2 \in \{1..8\}$, *collected in level* m *of phase* k *of a scan operation execution, one of the following three conditions must hold:*

1. $(\exists c1, c2 \in \{1..8\})\ (c1 < c2) \wedge (\ell_x^{c1,m,k} \not\prec \ell_y^{c2,m,k})$.
2. $(\exists c1, c2 \in \{1..8\})\ (c1 < c2) \wedge (\ell_y^{c1,m,k} \not\prec \ell_x^{c2,m,k})$.
3. *The four labels* $\ell_x^{2,m,k}$, $\ell_x^{4,m,k}$, $\ell_x^{6,m,k}$, *and* $\ell_x^{8,m,k}$ *differ from one another according to* $\prec$, *and the four labels* $\ell_y^{1,m,k}$, $\ell_y^{3,m,k}$, $\ell_y^{5,m,k}$, *and* $\ell_y^{7,m,k}$ *also differ from one another according to* $\prec$.

Proof. Let it be shown that if condition 3 does not hold, then either condition 1 or 2 holds. If condition 3 does not hold, then either

$$(\exists c1, c2 \in \{1..8\})\ (c1 + 1 < c2) \wedge (\ell_x^{c1,m,k} = \ell_x^{c2,m,k})$$

or

$$(\exists c1, c2 \in \{1..8\})\ (c1 + 1 < c2) \wedge (\ell_y^{c1,m,k} = \ell_y^{c2,m,k})$$

Note that labels of the same process can be the same, as denoted by the equivalence sign, though by definition those of different processes always differ by $\not\prec$. Without loss of generality, assume that the first condition holds. Then by definition, there must exist a label $\ell_y^{c,m,k}$, $c1 < c < c2$. If $\ell_x^{c1,m,k} \not\prec \ell_y^{c,m,k}$, then condition 1 holds and the claim is proven. Thus it must be the case that $\ell_y^{c,m,k} \not\prec \ell_x^{c1,m,k}$. However, since $\ell_x^{c1,m,k} = \ell_x^{c2,m,k}$, it is the case that $\ell_y^{c,m,k} \not\prec \ell_x^{c2,m,k}$, and condition 2 holds. ∎

By direct application of Claim 5.4.1, the following claim (Claim 5.4.4) implies that if condition 1 of Claim 5.4.3 holds, then $L_x^{1,m,k} \not\prec L_y^{8,m,k}$, and similarly, if condition 2 holds, then $L_y^{1,m,k} \not\prec L_x^{8,m,k}$. (This follows by exchanging the roles of x and y in Claim 5.4.4 below.)

CLAIM 5.4.4. *If* $\ell_x^{c1,m,k} \not\prec \ell_y^{c2,m,k}$, *then* $L_x^{c1,m,k} \Longrightarrow L_y^{c2,m,k}$.

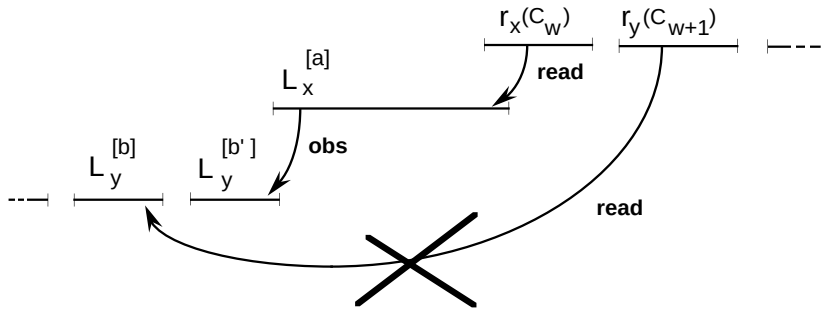


FIG. 12. *Greater and later implies precedence.*

Proof. For simplicity, let $(c1, m, k) = w$ (the label $\ell_x^{c1,m,k}$ read is $\ell_{x,w}^{[a]}$, that is, of labeling operation execution $L_x^{[a]}$) and $(c2, m, k) = w+1$ (similarly, $\ell_y^{c2,m,k}$ is $\ell_{y,w+1}^{[b]}$). The outline of the proof appears in Figure 12. Assume by way of contradiction that $\ell_{x,w}^{[a]} \not\prec \ell_{y,w+1}^{[b]}$ and $L_y^{[b]} \Longrightarrow L_x^{[a]}$. By Definition 4.3, it must be that $L_y^{[b]} \xrightarrow{\text{obs}} L_x^{[a]}$. By

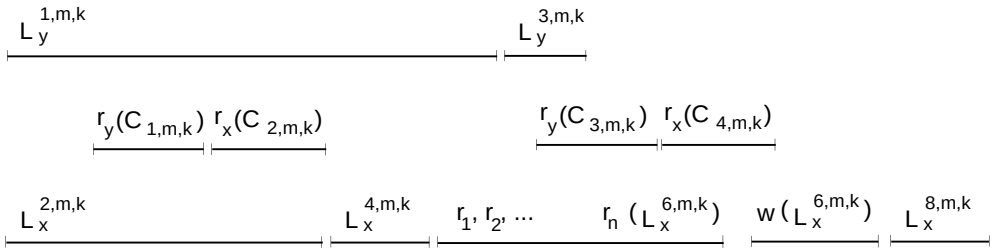


FIG. 13. *Three labeling "moves" are necessary.*

Lemma 5.2, it cannot be that $L_y^{[b]} \in max\_obs(L_x^{[a]})$. Thus there must exist an $L_y^{[b']}$, $b < b'$, such that $L_y^{[b']} \in max\_obs(L_x^{[a]})$. By Corollary 5.1, since $L_y^{[b']} \xrightarrow{obs} L_x^{[a]}$, there must exist some $r_y(L_\alpha^{[\beta]}) = \ell_y^{[b']}$ such that

$$w(L_y^{[b']}) \dashrightarrow r_y(L_\alpha^{[\beta]}) \longrightarrow w(L_x^{[a]}),$$

where possibly $\alpha = x$ and $\beta = a$. Since $r_y(C_{w+1}) = \ell_y^{[b]}$ and $r_y(L_\alpha^{[\beta]}) = \ell_y^{[b']}$, $b < b'$, by atomic register axiom B5, it must be that $r_y(C_{w+1}) \dashrightarrow r_y(L_\alpha^{[\beta]})$. Similarly, since $\ell_x^{[a]}$ was read in C_w, it must be by axiom B5 that $w(L_x^{[a]}) \dashrightarrow r_x(C_w)$. Thus

$$r_y(C_{w+1}) \dashrightarrow r_y(L_\alpha^{[\beta]}) \longrightarrow w(L_x^{[a]}) \dashrightarrow r_x(C_w),$$

which by axiom A4* of [AB87] implies that $r_y(C_{w+1}) \dashrightarrow r_x(C_w)$, a contradiction to $C_w \xrightarrow{*} C_{w+1}$. □

To complete the proof, it remains to be shown that if the first two conditions of Claim 5.4.3 do not hold (in which case the third one does), it is the case that $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$. One can intuitively think of this claim as stating that if each of the processes x and y "moved" (chose a new label) three times, the original labeling operation of y—before the three new ones—must have been completely before the latest labeling operation of x and so precedes it by $\Longrightarrow$. The reason that one needs three "moves" to assure this property becomes clear from the proof. The example in Figure 13 shows why if fewer "moves" are made by each, the property does not hold.

CLAIM 5.4.5. *If the four labels $\ell_x^{2,m,k}$, $\ell_x^{4,m,k}$, $\ell_x^{6,m,k}$, and $\ell_x^{8,m,k}$ differ from one another according to $\prec$ and the four labels $\ell_y^{1,m,k}$, $\ell_y^{3,m,k}$, $\ell_y^{5,m,k}$, and $\ell_y^{7,m,k}$ also differ from one another according to $\prec$, then $L_y^{1,m,k} \Longrightarrow L_x^{8,m,k}$.*

Proof. By serialization axiom B5 of reads and writes from the atomic registers v_x and v_y, it must be the case that

$$L_y^{1,m,k} \xrightarrow{*} w(L_y^{3,m,k}) \dashrightarrow r_y(C_{3,m,k}) \longrightarrow r_x(C_{4,m,k}) \dashrightarrow w(L_x^{6,m,k}) \xrightarrow{*} L_x^{8,m,k}.$$

By applying axiom A4 twice, it follows that $L_y^{1,m,k} \xrightarrow{*} L_x^{8.m,k}$, which by Definition 4.3 implies that $L_y^{1,m,k} \Longrightarrow L_x^{8.m,k}$. □

This completes the proof of Claim 5.4.2. □

The following claim proves the correctness of the recursive procedure *select*.

CLAIM 5.4.6. *For any k and R, if s is returned by select$(\lceil \log n \rceil, k, R)$, then $L_i^{1,1,k} \Longrightarrow L_s^{8,\lceil \log n \rceil,k}$ for all $i \in R - \{s\}$.*

Proof. First, observe that for $||r||$, the size of the input set of $select(m, k, r)$, it follows by simple induction (given that initially $||r|| \leq n$) that $m \geq \lceil \log ||r|| \rceil$. The proof of the claim will thus be by induction on $||r|| \in \{1..\lceil R \rceil\}$.

For $||r|| = 1$, the claim follows vacuously. For $||r|| = 2$, since $m \geq \lceil \log ||r|| \rceil = 1$, the claim follows from Claim 5.4.2. Assume that the claim holds for $||r|| < t$, and let the claim be proven for $||r|| = t$. Since $||half(r)||, ||other\_half(r)|| \leq t/2$, by the induction hypothesis applied to $select(\lceil \log t \rceil - 1, k, half(r))$ and $select(\lceil \log t \rceil - 1, k, other\_half(r))$, it follows that

$$(\forall i \in half(r))\ (L_i^{1,1,k} \Longrightarrow L_x^{8,\lceil \log t \rceil - 1,k}) \quad \text{and}$$
$$(\forall i \in other\_half(r))\ (L_i^{1,1,k} \Longrightarrow L_y^{8,\lceil \log t \rceil - 1,k}).$$

By Claim 5.4.2, without loss of generality, it can be assumed that $L_y^{1,\lceil \log t \rceil,k} \Longrightarrow L_x^{8,\lceil \log t \rceil,k}$ in $select(\lceil \log t \rceil, k, r)$. Thus since $C_{8,\lceil \log t \rceil - 1,k} \xrightarrow{*} C_{1,\lceil \log t \rceil,k}$, by Claim 5.4.1 and the above,

$$L_i^{1,1,k} \Longrightarrow L_y^{8,\lceil \log t \rceil - 1,k} \Longrightarrow L_y^{1,\lceil \log t \rceil,k} \Longrightarrow L_x^{8,\lceil \log t \rceil,k}$$

for every $i \in other\_half(r)$. By transitivity (Claim 5.3.2), it is the case that $L_i^{1,1,k} \Longrightarrow L_x^{8,\lceil \log t \rceil,k}$ for every $i \in other\_half(r)$. Similarly, by Claim 5.4.1 and the above, given that $C_{8,\lceil \log t \rceil - 1,k} \xrightarrow{*} C_{1,\lceil \log t \rceil,k}$, it follows that

$$L_i^{1,1,k} \Longrightarrow L_x^{8,\lceil \log t \rceil - 1,k} \Longrightarrow L_x^{8,\lceil \log t \rceil,k}$$

for every $i \in half(r)$. Again by transitivity, it is the case that $L_i^{1,1,k} \Longrightarrow L_x^{8,\lceil \log t \rceil,k}$ for every $i \in half(r)$, and the claim follows. □

Based on the above claims, the proof can be completed by showing that in any scan operation execution, for any i such that $O[i] < O[j]$, where $value(\ell_i^{8,\lceil \log n \rceil,O[i]}), value(\ell_j^{8,\lceil \log n \rceil,O[j]}) \in \bar{\ell}$, it is the case that $L_i^{8,\lceil \log n \rceil,O[i]} \Longrightarrow L_j^{8,\lceil \log n \rceil,O[j]}$. The proof is by induction on k, where $O[i] := k$ in phase k of a scan operation execution. For $k = n$, since there exists no $k', k < k'$, there is no $O[i] < O[j]$, and the claim holds vacuously. Assume that for some $k < n$, the claim holds for all $k', k < k' \leq n$. Let it be proven for k.

Since $k < n$, there is an $O[\alpha] = k + 1$ for some $\alpha \in \{1..n\} - \{i\}$ (possibly $\alpha = j$), where $value(\ell_\alpha^{8,\lceil \log n \rceil,k+1}) \in \bar{\ell}$ of the scan operation execution, that is, the returned labeled-value for process α. By Claim 5.4.6,

$$L_i^{1,1,k+1} \Longrightarrow L_\alpha^{8,\lceil \log n \rceil,k+1}$$

for $i \in R$. By Claim 5.4.1, since $C_{8,\lceil \log n \rceil,k} \xrightarrow{*} C_{1,1,k+1}$, it is the case that

$$L_i^{8,\lceil \log n \rceil,k} \Longrightarrow L_\alpha^{8,\lceil \log n \rceil,k+1}.$$

If $\alpha = j$ the lemma follows. If not, by the induction hypothesis, it follows that for any $O[j]$, $k+1 < O[j]$,

$$L_i^{8,\lceil \log n \rceil,k} \Longrightarrow L_\alpha^{8,\lceil \log n \rceil,k+1} \Longrightarrow L_\alpha^{8,\lceil \log n \rceil,O[j]}.$$

By the transitivity of $\Longrightarrow$ (Claim 5.3.2), it then follows that $L_i^{8,\lceil \log n \rceil,O[i]} \Longrightarrow L_j^{8,\lceil \log n \rceil,O[j]}$. □

5.9. Proof of precedence and transitivity. To complete the proof of Theorem 5.1, it remains to be proven that Lemma 5.2 and Claim 5.3.2 hold.

5.9.1. Preliminaries. Given that the definitions of both the graph T^n and the labeling function $\mathcal{L}$ are inductive on k, the first two parts of the following definition simply define the notation to be used in relating labels. The third part is the notion of *inside*(X). X identifies a specific labeling operation execution. In this labeling operation execution, the label chosen was in a certain T^k subgraph on level k. X also identifies this T^k subgraph. The set of labeling operation executions in *inside* are those performed inside T^k from the latest time the process moved into T^k and up to its labeling operation execution X. The *min* is simply the earliest in a sequence of labeling operation executions. For example, *min*(*inside*(X)) is the first among the moves since the process performing X entered T^k.

DEFINITION 5.2. *For $k \in \{1..n\}$ and $\xrightarrow{*}$, the ordering on labeling operation execution, we have the following notation:*

- *Let $\ell_y^{[b]} \overset{k}{=} \ell_x^{[a]}$ denote that $\ell_y^{[b]}[n..k-1] = \ell_x^{[a]}[n..k-1]$ for $k \geq 2$.*
- *Let $\ell_y^{[b]} \overset{k}{\neq} \ell_x^{[a]}$ (similarly, $\ell_y^{[b]} \overset{k}{\prec} \ell_x^{[a]}$) denote that $\ell_y^{[b]}[n..k] \overset{k+1}{=} \ell_x^{[a]}[n..k]$ and $\ell_y^{[b]}[k-1] \neq \ell_x^{[a]}[k-1]$ (similarly, $\ell_y^{[b]}[k-1]$ is dominated by $\ell_x^{[a]}[k-1]$).*
- *Let inside$(\ell_x^{[a]}[n..k])$ be a set of operation executions*

$$\{L_x^{[\alpha]} \mid \alpha = \text{ a or } L_x^{[\alpha]} \xrightarrow{*} L_x^{[a]} \text{ and } \ell_x^{[\alpha]} \overset{k+1}{=} \ell_x^{[a]} \text{ and } (\forall L_x^{[a']})\ (\text{if } L_x^{[\alpha]} \xrightarrow{*} L_x^{[a']} \xrightarrow{*} L_x^{[a]}, \text{ then } \ell_x^{[a']} \overset{k+1}{=} \ell_x^{[a]})\}.$$

- *Let the min of a set of labeling operation executions totally ordered by $\xrightarrow{*}$ be the least element in the ordering.*

If $\ell_x^{[a-1]} \overset{k}{=} \ell_x^{[a]}$, $k = 2$ (the same label by the same process), then let the convention be that $\ell_x^{[a-1]} \overset{k}{\neq} \ell_x^{[a]}$, where $k = 1$ (and similarly for any two equal labels of different labelings by the same process).

5.9.2. The order of induction. The proof of Claim 5.3.2 and Lemma 5.2 will proceed by induction on the system execution $\langle m(\varphi)^L, \xrightarrow{*}, \dashrightarrow^{*} \rangle$ consisting of all labeling operation executions in $m(\varphi)^L$. (Recall that $m(\varphi)^L$ is the set of labeling operation executions in $m(\varphi)$.) The induction base will be the subexecution $m(\varphi)^{L'} = \{L_1^{[0]}, \ldots, L_n^{[0]}\}$ of $m(\varphi)^L$. The induction will proceed to larger subexecutions $m(\varphi)^{L'}$, where $m(\varphi)^{L'} \subseteq m(\varphi)^L$. The subexecution in each step of the induction will include one $L_i^{[a]} \in m(\varphi)^L$ more than its preceding one. The induction order on $\langle m(\varphi)^L, \xrightarrow{*}, \dashrightarrow^{*} \rangle$ is thus that $m(\varphi)^{L'} \cup \{L_i^{[a]}\}$ follows $m(\varphi)^{L'}$, where $L_i^{[a]} \in m(\varphi)^L - m(\varphi)^{L'}$, if for any $L_j^{[b]} \in m(\varphi)^L - m(\varphi)^{L'}$, it is the case that either

- $L_i^{[a]} \dashrightarrow^{\text{obs}} L_j^{[b]}$ or
- $L_j^{[b]} \not\dashrightarrow^{\text{obs}} L_i^{[a]}$ and for $\ell_i^{[a-1]} \overset{k}{\prec} \ell_i^{[a]}$ and $\ell_j^{[b-1]} \overset{k'}{\prec} \ell_j^{[b]}$, it is the case that $k' > k$ or that $k' = k$ and $j > i$.

The order is thus to add the labeling operation executions that observed a greater part of the execution later, and if no such labeling operation execution can be identified, settle on choosing the one that was a move (a change in the label) on the lowest-level k.

To see that the above defines a total order of induction, note that $\dashrightarrow^{\text{obs}}$ is a partial order, and if two labels are not related by $\dashrightarrow^{\text{obs}}$, they are ordered by the order $<$ on the level in the graph in which they made their last move and by the id if they have

the same level. Since $<$ together with the id forms a total order that is independent of the partial order $\xrightarrow{\text{obs}}$, the above order of induction is total.

5.9.3. The induction hypothesis. The induction hypothesis consists of I1 $\wedge$ I2 $\wedge$ I3 $\wedge$ I4, where I1–I4 are as follows:

I1. For any $L_y^{[b]} \in max\_obs(L_x^{[a]})$, it is the case that $\ell_y^{[b]} \not\preccurlyeq \ell_x^{[a]}$.

I2. The relation $\Longrightarrow$ is transitive.

I3. For any $L_x^{[a]}$ and $L_y^{[b]}$, where

- $\ell_y^{[b]}[k-1]$, $\ell_x^{[a]}[k-1] \in \{3,4,5\}$ and
- $\ell_y^{[b]} \overset{k}{\neq} \ell_x^{[a]}$, $k \geq 2$,

if there exist labeling operation executions $\ell_x^{[a-1]}$ and $\ell_y^{[b-1]}$, where

- $\ell_x^{[a-1]} \overset{k}{\neq} \ell_x^{[a]}$ and
- $\ell_y^{[b-1]} \overset{k}{\neq} \ell_y^{[b]}$,

then either $L_x^{[a]} \xrightarrow{\text{obs}} L_y^{[b]}$ or $L_y^{[b]} \xrightarrow{\text{obs}} L_x^{[a]}$.

I4. 1. If $\ell_x^{[a]}[k-1] \in \{2,3,4,5\}$, $k > 2$, then there are at least $k-1$ labels $L_y^{[b]} \in max\_obs(L_x^{[a]})$ such that $\ell_y^{[b]} \overset{k+1}{=} \ell_x^{[a]}$;

2. if there exists an $L_x^{[a1]} \in inside(\ell_x^{[a]}[n..k])$, $\ell_x^{[a1]}[k-1] \in \{4,5\}$ (possibly $a = a1$), then there are exactly $k-1$ labels $L_y^{[b]} \in max\_obs(L_x^{[a]})$ such that $\ell_y^{[b]} \overset{k+1}{=} \ell_x^{[a]}$ and $\ell_y^{[b]}[k-1] \in \{3,4,5\}$; and

3. if $\ell_x^{[a]} \not\preccurlyeq \ell_x^{[a-1]}$ ($\ell_x^{[a]}[k-1]$, $\ell_x^{[a-1]}[k-1] \in \{3,4,5\}$), then for any $L_y^{[b]} \in max\_obs(L_x^{[a]})$, where $\ell_y^{[b]} \overset{k+1}{\neq} \ell_x^{[a-1]}$ and $\ell_y^{[b]}[k-1] \in \{3,4,5\}$, it is the case that $\ell_x^{[a-1]} \not\preccurlyeq \ell_y^{[b]} \not\preccurlyeq \ell_x^{[a]}$.

The induction hypothesis includes four main parts. I1 and I2 are simply Lemma 5.2 and Claim 5.3.2, which are to be proven. However, the proof of these properties is based on several "structural" properties of the labeling operation executions, and these are added in order to strengthen the induction hypothesis.

Property I3 is a weak formulation for the case of any T^k subgraph, $k \geq 2$, of a powerful property that holds in the case of a T^2 subgraph. For $k = 2$, that is, two labels in the cycle of a T^2 subgraph, it is the case that

> among any two labeling operation executions in the cycle, there must be one that observed the other.

Unfortunately, this is not true for any pair of labeling operation executions in a cycle on level $k > 2$. For example, the reader can verify that it is possible that while one process x moves among supernodes 3 and 4 on level k, another process y can concurrently move many times inside supernode 3 (that is, on a level lower than k) with neither x nor y observing a labeling operation execution of the other. However, the property that does hold is that the process x must have observed at least one labeling operation execution by y among those that y executed since it last started choosing labels in supernode 3. (Thus the first move into 3 was definitely observed.) The generalization of this example is formalized by property I3 of the inductive hypothesis.

Property I4 is a collection of three properties that were informally mentioned in section 4.3:

- I4.1 is based on the fact that supernode 1 in any T^k subgraph is a sink in which at least $k-1$ labels must accumulate before a label may be placed on the bridge supernode 2. Because of this accumulation property, any process that performs a

labeling operation execution on supernodes $\{2,3,4,5\}$ must have maximally observed at least $k-1$ other labels in the subgraph with him. The maximally observed set of operations of a labeling operation execution $L_x^{[a]}$ ($max\_obs(L_x^{[a]})$) is actually the set of labeling operation executions whose labels, in a sequential execution, could have existed together with $L_x^{[a]}$ at some point in time. Thus I4.1 can be thought of as establishing that if a process completes a labeling operation execution on one of the supernodes $\{2,3,4,5\}$, there are at that point in time at least $k-1$ other labels in the subgraph with him.

• I4.2 is a continuation of the behavior described in I4.1. Again, given that the maximally observed set of operations of a labeling operation execution $L_x^{[a]}$ represents the set of labeling operation executions whose labels, in a sequential execution, could have existed together with $L_x^{[a]}$ at some point in time, I4.2 formalizes the "invariant" that

> at any given time, there cannot be more than k labels in a cycle of a T^k structure.

In addition, not only is it true that there are not more than k, but if any one of these k labels moves inside the cycle, it must maximally observe exactly $k-1$ other labels in the cycle with it.

• Finally, I4.3 strengthens I1 for the particular case in which the new label chosen is dominated by the older label (such as a move from supernode 3 to 5 in the cycle). Based on I1, it could still be that some of the labels maximally observed by the process, though dominated by the new label, are on node 5 together with it. I4.3 establishes that this cannot be the case, that is, all other labels maximally observed in the cycle must be on supernode 4. Property I1 together with I4.3 capture the the "invariant" that

> at any given time, there are never labels on three different nodes of a cycle of a T^k subgraph.

In the next two sections, the induction base and the inductive step are presented.

5.9.4. The induction base.

LEMMA 5.5. *The hypothesis* $\mathrm{I1} \wedge \mathrm{I2} \wedge \mathrm{I3} \wedge \mathrm{I4}$ *holds for* $m(\varphi)^{L'} = \{L_1^{[0]}, \ldots, L_n^{[0]}\}$.

Proof. By definition, initially $max\_obs(L_x^{[a]}) = \emptyset$, and I1 and I4 hold vacuously. Since for any $L_x^{[a]}$, $a = 0$, there does not by definition exist an $L_x^{[a-1]}$, I3 holds vacuously. Also, by definition, for any two labels $\ell_x^{[0]}$ and $\ell_y^{[0]}$, $\ell_x^{[0]} \overset{k1}{\neq} \ell_y^{[0]}$, where $k1 = 1$, and $L_x^{[0]} \overset{obs}{\not\longleftrightarrow} L_y^{[0]}$. Since $\overset{k1}{\prec}$ is a total order for level $k1 = 1$, it follows that $\Longrightarrow$ is transitive in $m(\varphi)^{L'}$. □

5.9.5. The induction step.

LEMMA 5.6. *Given that the induction hypothesis* $\mathrm{I1} \wedge \mathrm{I2} \wedge \mathrm{I3} \wedge \mathrm{I4}$ *holds for the system execution* $\langle m(\varphi)^{L'}, \overset{*}{\longrightarrow}, \overset{*}{\dashrightarrow} \rangle$, $m(\varphi)^{L'} \subseteq m(\varphi)^{L}$, *it holds also for* $\langle m(\varphi)^{L'} \cup \{L_x^{[a]}\}, \overset{*}{\longrightarrow}, \overset{*}{\dashrightarrow} \rangle$, *where* $L_i^{[a]} \in m(\varphi)^{L} - m(\varphi)^{L'}$ *is such that for any* $L_j^{[b]} \in m(\varphi)^{L} - m(\varphi)^{L'}$, *either*

• $L_i^{[a]} \overset{obs}{\dashrightarrow} L_j^{[b]}$ *or*

• $L_j^{[b]} \overset{obs}{\not\longleftrightarrow} L_i^{[a]}$ *and for* $\ell_i^{[a-1]} \overset{k}{\prec} \ell_i^{[a]}$ *and* $\ell_j^{[b-1]} \overset{k'}{\prec} \ell_j^{[b]}$, *it is the case that* $k' > k$ *or that* $k' = k$ *and* $j > i$.

The proof of Lemma 5.6 will be separated into several sections. In the following section, several lemmas that will become useful in later sections of the proof are

presented and proven. The proof will then proceed by showing that the *maximally observed* set of labeling operation executions by a process x is a good representation of the possible label values that other processes can have given the location of x. In other words, later unobserved labeling operation executions cannot be "far away" from the maximally observed labels, and could definitely not have "cycled around" the current location of x. Based on these established properties, I1, I4, I3, and finally I2 will be proven for the inductive case. The order of presentation of the different lemmas will follow the order of dependency among them.

We make a final important comment: Throughout the proof, unless specifically stated otherwise, $L_x^{[a]}$ will denote the labeling operation execution added in the induction step to form $\langle m(\varphi)^{L'} \cup \{L_x^{[a]}\}, \xrightarrow{*}, \dashrightarrow^{*} \rangle$,

5.9.6. At most k labels in the cycle of a T^k subgraph. In this section, several lemmas are presented, proving a lemma that captures the informal invariant that at any point in time, there can be at most k different labels in the cycle of a T^k subgraph (supernodes $\{3,4,5\}$. The following lemma formalizes the notion that "before it can choose a label in the cycle of any T^k subgraph, a process must first raise a flag, that is, choose a label on supernodes 1 or 2 on level k in T^k."

LEMMA 5.7. *For any labeling operation execution $L_x^{[a]}$, if $\ell_x^{[a]}[k-1] \in \{3,4,5\}$, $k \geq 2$, then there exists an $L_x^{[a1]} \in inside(\ell_x^{[a]}[n..k])$ such that $L_x^{[a1]}[k-1] \in \{1,2\}$.*

Proof. Assume by way of contradiction that the claim does not hold. It must thus be that for $L_x^{[a2]} = min(inside(\ell_x^{[a]}[n..k]))$, $\ell_x^{[a2]}[k-1] \in \{3,4,5\}$. This implies that there is a labeling operation execution $\ell_x^{[a2-1]} \overset{k+1}{\neq} \ell_x^{[a2]}$. By the definition of $\mathcal{L}$, in order for $\ell_x^{[a2]}[k-1]$ to be in $\{3,4,5\}$, it must be that for $\ell_{\max}$, the maximal label in the dominating set read by $L_x^{[a2]}$, we have the following:

- $\ell_{\max} \overset{k+1}{=} \ell_x^{[a2]}$ (as a reminder, this means $\ell_{\max}[n..k] = \ell_x^{[a2]}[n..k]$),
- $\ell_{\max}[k-1] \in \{2,3,4,5\}$, and
- $\mathcal{L}^k(G)$ (the kth level of the recursion in $\mathcal{L}$) was executed for $G = \ell_{\max}[n..k]$ and returned the value $\ell_x^{[a2]}[k-1] = 3$ (as in line 3) or $\ell_x^{[a2]}[k-1] = dom(\ell_{\max}[k-1])$ (as in line 4 or 5).

But this implies that when executing $\mathcal{L}^{k+1}$, it must have been line 4 that was executed because from the above the conditions of lines 1–3 are not met and because

- $\ell_{\max}[k-1] \in \{2,3,4,5\}$, $k \geq 2$, and
- $\ell_{\max}[n..k] = \ell_x^{[a2]}[n..k] \neq \ell_x^{[a2-1]}[n..k]$ ($\ell_x^{[a2-1]}$ is ℓ_i in line 4).

But this implies that $\ell_x^{[a2]}[k] = dom(\ell_{\max}[k])$, that is, x would not execute $\mathcal{L}^k(G)$ for $G = \ell_{\max}[n..k]$ in the first place, a contradiction. □

The following lemma establishes that if in an earlier labeling operation execution a label $\ell_y^{[b]}$ was observed, the current labeling operation execution must read that label for y or a label later than it.

LEMMA 5.8. *If $L_y^{[b]} \overset{\text{obs}}{\dashrightarrow} L_x^{[a-1]}$, it cannot be that $r_y(L_x^{[a]}) = \ell_y^{[b1]}$, where $b1 < b$.*

Proof. By Corollary 5.1, it follows that if $L_y^{[b]} \overset{\text{obs}}{\dashrightarrow} L_x^{[a-1]}$, then there exists a read $r_y(L_\alpha^{[\beta]}) = \ell_y^{[b]}$ such that

$$w(L_y^{[b]}) \dashrightarrow r_y(L_\alpha^{[\beta]}) \longrightarrow w(L_x^{[a-1]}),$$

where possibly $\alpha = x$ and $\beta = a-1$. Since $w(L_x^{[a-1]}) \longrightarrow r_y(L_x^{[a]})$, it follows that $r_y(L_\alpha^{[\beta]}) \longrightarrow r_y(L_x^{[a]})$. Since, in addition, $w(L_y^{[b1]}) \longrightarrow w(L_y^{[b]})$, by register axiom B5, it cannot be that $r_y(L_x^{[a]}) = \ell_y^{[b]}$. □

The following lemma states that there cannot be a label read by $L_x^{[a]}$ that dominates $\ell_x^{[a-1]}$ on level $k1 > k$, where k is the level such that $\ell_x^{[a-1]} \overset{k}{\neq} \ell_x^{[a]}$.

LEMMA 5.9. *For $\ell_x^{[a]} \overset{k}{\neq} \ell_x^{[a-1]}$, it cannot be that there is an $L_y^{[b]}$ such that*

- $r_y(L_x^{[a]}) = \ell_y^{[b]}$ *and*
- $\ell_x^{[a-1]} \overset{k1}{\prec} \ell_y^{[b]}$, *where* $k1 > k$.

Proof. By the definition of $\overset{k}{\prec}$, it must be that $\ell_x^{[a]} \overset{k1}{\prec} \ell_y^{[b]}$, where $k1 \geq k$. By the definition of $\mathcal{L}$, either $\ell_{\max} \overset{k}{\prec} \ell_x^{[a]}$ or $\ell_{\max}$ is equal to $\ell_x^{[a]}$ (in which case by definition $\ell_{\max}$ is just $\ell_x^{[a-1]}$). The reason is that when executing $\mathcal{L}^{k3}$ for some level $k3$, $\ell_{\max}[k3] = \ell_x^{[a]}[k3]$ or $\ell_{\max}[k3] = dom(\ell_x^{[a]}[k3])$. It thus must be that $max \neq y$. It can either be the case that $\ell_y^{[b]} \overset{k}{\prec} \ell_{\max}$ or not.

If indeed $\ell_y^{[b]} \overset{k}{\prec} \ell_{\max}$, by the definition of $\overset{k}{\prec}$, in order for $\ell_x^{[a]} \overset{k}{\prec} \ell_y^{[b]}$, $\ell_y^{[b]} \overset{k}{\prec} \ell_{\max}$, and either $\ell_{\max} \overset{k}{\prec} \ell_x^{[a]}$ or $\ell_{\max} = \ell_x^{[a]}$, it must be that

$$\ell_x^{[a]} \overset{k}{\prec} \ell_y^{[b]} \overset{k}{\prec} \ell_{\max} \overset{k}{\prec} \ell_x^{[a]},$$

that is, the three labels are also on a cycle. Since by the definition of

$$max(dominating\_set(\ell, \ell_x^{[a-1]})),$$

either $\ell_x^{[a-1]} \overset{k}{\prec} \ell_{\max}$ or $\ell_{\max} = \ell_x^{[a-1]}$, it follows that $k \geq k1$, a contradiction.

However, if $\ell_{\max} \overset{k}{\prec} \ell_y^{[b]}$, by the definition of $max(dominating\_set(\ell, \ell_x^{[a-1]}))$, it could be only if the labels of y and max were on a cycle on a level $k2$, where $2 \leq k2 \leq k1$ ($k1$ is the level such that $\ell_x^{[a-1]} \overset{k1}{\prec} \ell_y^{[b]}$).[15] In order for $\ell_{\max} \overset{k}{\prec} \ell_x^{[a]}$ or $\ell_{\max} = \ell_x^{[a]}$, together with $\ell_x^{[a]} \overset{k}{\prec} \ell_y^{[b]}$, it must be that $\ell_x^{[a]}$ is in the cycle with these two labels. However, this implies $k \geq k1$, a contradiction. □

The following lemma captures the informal invariant that at any point in time, there can be at most k different labels in the cycle of any T^k subgraph. More precisely, it states that for any set of more than k labeling operation executions whose labels are in the cycle of the same T^k subgraph, all could not have been there at the same point in time since at least one of them must have already been observed by the others in a later location outside the cycle.

LEMMA 5.10. *Let $\mathcal{S}^k = \{L_{i_1}^{[a_1]}, L_{i_2}^{[a_2]}, \ldots, L_{i_m}^{[a_m]}\}$, $i_1, \ldots, i_m \in \{1..n\}$, and $i_\alpha \neq i_\beta$ for any $\alpha, \beta \in \{1..m\}$ be the set of labeling operation executions such that for any $L_i^{[a]}, L_j^{[b]} \in \mathcal{S}^k$,*

- $\ell_i^{[a]} \overset{k+1}{=} \ell_j^{[b]}$ *and* $\ell_i^{[a]}[k-1], \ell_j^{[b]}[k-1] \in \{3,4,5\}$, *and*
- *for $L_j^{[b1]} \in max\_obs(L_i^{[a]})$ and $L_i^{[a1]} \in max\_obs(L_j^{[b]})$, it is the case that $b1 \leq b$ and $a1 \leq a$.*

It must be that $||\mathcal{S}^k|| \leq k$.

Proof. Assume by way of contradiction that $||\mathcal{S}^k|| > k$. By Lemma 5.7, for each $L_i^{[a]} \in \mathcal{S}^k$, $||inside(L_i^{[a]}[n..k])|| \geq 2$, that is, it is included at least two labeling operation executions inside the T^k subgraph that $L_i^{[a]}$ is in. Let us define the relation *not_read_by* between labeling operation executions $L_i^{[a]}, L_j^{[b]} \in \mathcal{S}^k$ to be as follows.

DEFINITION 5.3. *$L_i^{[a]}$ not_read_by $L_j^{[b]}$ if $r_i(L_j^{[b]}) \neq \ell_i^{[a1]}$, $a1 \in \{a-1, a\}$.*

[15] The reason for this is that if the two labels are on different supernodes of a cycle, there could be a third label on the other supernode of the cycle, and any one of them could be selected as $\ell_{\max}$.

That is, $L_j^{[b]}$ did not read a label of a labeling operation execution $L_i^{[a]}$ or its preceding operation execution in the T^k that it is in. The contradiction will be derived by showing that there must be at least one labeling operation execution $L_i^{[a]} \in \mathcal{S}^k$ that read at least $k+1$ labels (including its own) in the T^k subgraph that $L_i^{[a]}$ and $L_i^{[a-1]}$ are in. This is the *flag principal* mentioned in section 4.3. Since for each $L_i^{[a]} \in \mathcal{S}^k$, $\ell_i^{[a]_k} = \ell_i^{[a-1]}$, that is, a move at level k, it must be that when executing $\mathcal{L}^{k+1}$ in $L_i^{[a]}$, line 5 was executed and that $num\_labels < (k+1) - 1$ (at most $k-1$ labels not including its own, or k including it, were read in the T^k subgraph $\ell_i^{[a]}[n..k]$), a contradiction.

Since it was assumed by way of contradiction that there are more than k labeling operation executions in $\mathcal{S}^k$, it must be that each labeling operation execution did not read (*not_read_by*) at least one of the others. Let it first be shown that the relation *not_read_by* is antisymmetric.

CLAIM 5.10.1. *For any $L_i^{[a]}$ and $L_j^{[b]}$ in $\mathcal{S}^k$, if $L_i^{[a]}$ not_read_by $L_j^{[b]}$, then it cannot be that $L_j^{[b]}$ not_read_by $L_i^{[a]}$.*

Proof. For any two labeling operation executions $L_i^{[a]}$ and $L_j^{[b]}$ in $\mathcal{S}^k$, by definition (for $L_j^{[b1]} \in max\_obs(L_i^{[a]})$ and $L_i^{[a1]} \in max\_obs(L_j^{[b]})$, it is the case that $b1 \leq b$ and $a1 \leq a$,) neither $r_i(L_j^{[b]}) = \ell_i^{[a1]}$, $a1 > a$, nor $r_j(L_i^{[a]}) = \ell_j^{[b1]}$, $b1 > b$. Given $L_i^{[a]}$ *not_read_by* $L_j^{[b]}$, it thus follows by atomic register axiom B5 that $r_i(L_j^{[b]}) \dashrightarrow w(L_i^{[a-1]})$. However, this implies

$$w(L_j^{[b-1]}) \longrightarrow r_i(L_j^{[b]}) \dashrightarrow w(L_i^{[a-1]}) \longrightarrow r_j(L_i^{[a]}).$$

By axiom A4, it follows that $w(L_j^{[b-1]}) \longrightarrow r_j(L_i^{[a]})$, implying that it cannot be that $L_j^{[b]}$ *not_read_by* $L_i^{[a]}$. □

Think of the relation *not_read_by* as the set of edges of a directed graph whose nodes are labeling operation executions, where an edge is directed from $L_i^{[a]}$ to $L_j^{[b]}$ if $L_i^{[a]}$ *not_read_by* $L_j^{[b]}$. Each labeling operation execution in $\mathcal{S} \cup \{L_x^{[a]}\}$ did not read at least one of the others; each node has at least one incoming edge. By known graph-theoretic arguments, this implies that

- there are two nodes that have edges directed one at the other or
- there is at least one node $L_s^{[c]}$ that has a directed path leading from it to every other node in the graph.

By Claim 5.10.1 (antisymmetry of *not_read_by*), the former is impossible.[16] The following claim establishes that the labeling operation execution associated with the node $L_s^{[c]}$ from which all other nodes are reachable (note that by assumption, this node has at least one incoming edge and is not a "root") must have read all of them.

CLAIM 5.10.2. *For any subset $\{L_{i_1}^{[a_1]}, L_{i_2}^{[a_2]}, \ldots, L_{i_m}^{[a_m]}\}$ of m labeling operation executions in $\mathcal{S}^k$, where*

$$L_{i_1}^{[a_1]} \text{ not\_read\_by } L_{i_2}^{[a_2]} \text{ not\_read\_by} \cdots \text{ not\_read\_by } L_{i_m}^{[a_m]},$$

it is the case that $r_{i_m}(L_{i_1}^{[a_1]}) = L_{i_m}^{[a_m]}$.

[16] Note that if the former does not hold, there is a cycle in the graph. If the relation *not_read_by* were transitive, a cycle would be impossible, and the proof would be complete. However, the reader can verify that this is not the case.

Proof. For any two labeling operation executions $L_i^{[a]}$, $L_j^{[b]} \in \mathcal{S}^k$, it follows by definition that neither $r_i(L_j^{[b]}) = \ell_i^{[a1]}$, $a1 > a$, nor $r_j(L_i^{[a]}) = \ell_j^{[b1]}$, $b1 > b$.

Let it be proven by induction that $r_{i_{m-1}}(L_{i_m}^{[a_m]}) \dashrightarrow w(L_{i_1}^{[a_1-1]})$. This will imply

$$w(L_{i_m}^{[a_m-1]}) \longrightarrow r_{i_{m-1}}(L_{i_m}^{[a_m]}) \dashrightarrow w(L_{i_1}^{[a_1-1]}) \longrightarrow r_{i_m}(L_{i_1}^{[a_1]}),$$

from which by axiom A4 follows $w(L_{i_m}^{[a_m-1]}) \longrightarrow r_{i_m}(L_{i_1}^{[a_1]})$, implying, $r_{i_m}(L_{i_1}^{[a_1]}) = L_{i_m}^{[a_m]}$, as desired.

The proof that $r_{i_{m-1}}(L_{i_m}^{[a_m]}) \dashrightarrow w(L_{i_1}^{[a_1-1]})$ is by induction on m, the size of the subset of labeling operation executions. For $m = 2$, it follows by definition. Assume it holds for sequences of length $m-1$, that is,

$$r_{i_{m-2}}(L_{i_{m-1}}^{[a_{m-1}]}) \dashrightarrow w(L_{i_1}^{[a_1-1]}).$$

Since $r_{i_{m-1}}(L_{i_m}^{[a_m]}) \neq \ell_{i_{m-1}}^{[a_{m-1}]}$, it follows that

$$r_{i_{m-1}}(L_{i_m}^{[a_m]}) \dashrightarrow w(L_{i_{m-1}}^{[a_{m-1}-1]}) \longrightarrow r_{i_{m-2}}(L_{i_{m-1}}^{[a_{m-1}]}) \dashrightarrow w(L_{i_1}^{[a_1-1]}).$$

By axiom A4*, it follows that $r_{i_{m-1}}(L_{i_m}^{[a_m]}) \dashrightarrow w(L_{i_1}^{[a_1-1]})$, implying the claim. □

Thus the node $L_s^{[c]}$ read at least k labels apart from its own in the T^k subgraph $L_s^{[c]}[n..k]$, providing the desired contradiction. □

Based on the above, the following lemma, which is part of the proof of I4.2 for the inductive case, can be shown. As mentioned before, the maximally observed set *max_obs*$(L_x^{[a]})$ is actually the set of labeling operation executions whose labels, in a sequential execution, could have existed together with $L_x^{[a]}$ at some point in time. The lemma thus captures the informal notion that if one could look at the cycle of a T^k subgraph at a given point in time in which x had a label $\ell_x^{[a]}$ in it, there would be at most $k-1$ other labels in the cycle together with it.

LEMMA 5.11. *For $\ell_x^{[a]}[k] \in \{3,4,5\}$, there are at most $k-1$ labels $L_y^{[b]}$, where $L_y^{[b]} \in$ max_obs$(L_x^{[a]})$, such that $\ell_y^{[b]} \overset{k+1}{=} \ell_x^{[a]}$ and $\ell_y^{[b]}[k-1] \in \{3,4,5\}$.*

Proof. For any two labeling operation executions $L_y^{[b]}$, $L_z^{[c]} \in$ *max_obs*$(L_x^{[a]})$, by the definition of *max_obs*$(L_x^{[a]})$, neither $r_y(L_z^{[c]}) = \ell_y^{[b1]}$, $b1 > b$, nor $r_z(L_y^{[b]}) = \ell_z^{[c1]}$, $c1 > c$. Also, by definition, for $L_y^{[b]} \in$ *max_obs*$(L_x^{[a]})$, neither $r_y(L_x^{[a]}) = \ell_y^{[b1]}$, $b1 > b$, nor $r_x(L_y^{[b]}) = \ell_x^{[a1]}$, $a1 \geq a$. The claim follows from Lemma 5.10 by defining $\mathcal{S}^k$ to be the set of labeling operation executions maximally observed by $L_x^{[a]}$ together with $L_x^{[a]}$ itself. □

The completion of the inductive argument involves a proof of properties I1–I4 through rather tedious case analysis. It is ommitted from this manuscript and can be found in [Sha90].

6. Discussion. There are three main types of problems defined in the shared-memory model:

- *waiting problems*, whose solution allows a process to take an infinite number of steps to complete an operation—that is, it could "busy-wait" for some other processes indefinitely;
- *wait-free problems*, whose solution is such that each process is *guaranteed* to complete an operation within a finite number of steps, independently of the pace of other processes; and

• *expected-wait-free problems*, whose solution is such that each process is *expected* (rather than guaranteed) to complete an operation within a finite number of steps, independently of the pace of other processes.

These classes of problems are fundamentally different from one another. However, they have the unifying theme that

> if the requirement that memory size be bounded is dropped, the problems have elegant and simple unbounded solutions based on the use of a CTSS.

The main implication of bounded concurrent time-stamping is that this unifying theme, true under the assumption that memory size can be unbounded, holds true for the bounded-memory case as well.

Based on the use of a bounded CTSS implementation, simple unbounded solutions can be given for what are considered to be core problems in each category and then directly transformed into bounded ones. Examples of problems and algorithms in the first category are the famous first-come first-served mutual-exclusion problem of Lamport [Lam74] (see [Lam86b, Ray86] for complete details) and the *fifo*-ℓ-exclusion problem of [AD*94, FLBB79, FLBB89]. As mentioned earlier, a CTSS-based solution due to Afek et al. can be found in [AD*94].

In the second category, we have Li and Vitanyi's simple version [LV87] of the elegant unbounded Vitanyi–Awerbuch algorithm [VA86] for solving the problem of providing a wait-free construction of an MRMW atomic register from SWMR atomic registers (see also [PB87, IL93, Sch88, **?**]). This algorithm can be immediately transformed into a bounded solution (see [G92]).

In the third category, a version (see [Sha90]) of the algorithm of Abrahamson [Abr88] based on the use of a CTSS can be modularly transformed into a bounded solution to the randomized consensus problem of [CIL87].

6.1. Further related research. The introduction of the concurrent time-stamping paradigm in the conference version of this paper [DS89] has led researchers to devising a series of alternative bounded CTSS algorithms. Israeli and Pinchasov [IP91] have provided a linear-complexity version of our algorithm by dropping the requirement that scan operations do not perform writes. In [DW92], Dwork and Waarts present the most efficient read/write-register-based CTSS construction to date, taking only $O(n)$ time for either a scan or update. They model their bounded construction after a new type of unbounded CTSS construction, where processes choose from "local pools" of label values instead of the simple "global-pool"-based CTSS as in the bakery algorithm [Lam74]. In order to bound the number of possible label values in the local pool of the bounded implementation, they introduce a form of garbage collection on "old" labels. They then prove that the linear-time bounded implementation meets the CTSS axioms of section 2. In [DPHW92], Dwork, Herlihy, Plotkin, and Waarts introduce an alternative linear-complexity bounded CTSS construction that combines a time-lapse snapshot with our bounded CTSS algorithm. The proof of their algorithm leverages the axiomatic proof in this paper by arguing that the executions of their algorithm are a subset of the executions of our algorithm. In [GLS92], Gawlick, Lynch, and Shavit introduce a streamlined version of our CTSS algorithm based on the use of an atomic snapshot primitive [AAD*89, And89a]. A snapshot primitive allows a process P_i to *update* the ith memory location, or *snap* the memory, that is, collect an "instantaneous" view of all n shared-memory locations. By using a snapshot primitive, they limit the number of interleavings that can occur and are able to introduce a considerably simplified version of our labeling algorithm (though

a logarithmic factor less efficient) that is tailored to allow a forward-simulation proof [LT87]. An advantage of their algorithm over other solutions is that it is no longer limited to read/write memory, providing a CTSS construction in any computation model whose basic operations suffice to provide a wait-free snapshot implementation, be it single-writer multireader registers [A93], multireader multiwriter registers [ICMT94], consensus objects [CD93], or memory with hardware supported *compare-and-swap* and *fetch-and-add* primitives.

Acknowledgments. We would like to thank Yehuda Afek, Hagit Attiya, Eli Gafni, Rainer Gawlick, Maurice Herlihy, Nancy Lynch, and Mike Merritt for many important conversations and comments. It was a subtle observation of Mike's regarding pairwise consistency among scans that led us to the current CTSS definitions. A subsequent observation by Rainer led us to add property P4 to the CTSS specification.

Finally, the second author would like to thank Nancy Lynch, Baruch Awerbuch, and the members of MIT's Theory of Distributed Systems group for their warm hospitality throughout the writing of this paper.

REFERENCES

[AAD*89] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, *Atomic snapshots of shared memory*, J. Assoc. Comput. Mach., 40 (1993), pp. 873–890.

[AB87] U. Abraham and S. Ben-David, *Informal and formal correctness proofs for programs (for the critical section problem)*, unpublished manuscript, Technion, Haifa, Israel, 1987.

[Abr88] K. Abrahamson, *On achieving consensus using a shared memory*, in Proc. 7th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1988, pp. 291–302.

[AD*94] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, *A bounded first-in, first-enabled solution to the ℓ-exclusion problem*, ACM Trans. Programming Lang. Systems, 16 (1994), pp. 939–953.

[A93] H. Attiya and O. Rachman, *Atomic snapshots in $O(n \log n)$ operations*, in Proc. 12th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1993, pp. 29–40.

[AG90] J. Anderson and M. Gouda, *The virtue of patience: Concurrent programming with and without waiting*, Technical Report TR-90-23, Department of Computer Science, University of Texas at Austin, Austin, TX, 1990.

[And89a] J. H. Anderson, *Multi-writer composite registers*, Distrib. Comput., 7 (1994), pp. 175–195.

[Ben88] S. Ben-David, *The global time assumption and semantics for concurrent systems*, in Proc. 7th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1988, pp. 223–231.

[Blo88] B. Bloom, *Constructing two-writer atomic registers*, in Proc. 6th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1987; revised version, IEEE Trans. Commun., 37 (1988), pp. 1506–1514.

[BP87] J. Burns and G. Peterson, *Constructing multi-reader atomic values from non-atomic values*, in Proc. 6th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1987, pp. 221–231.

[CD93] T. D. Chandra and C. Dwork, *Using consensus to solve atomic snapshots*, manuscript, 1993.

[CIL87] B. Chor, A. Israeli, and M. Li, *Wait-free consensus using asynchronous hardware*, SIAM J. Comput., 23 (1994), pp. 701–712.

[CS93] R. Cori and E. Sopena, *Some combinatorial aspects of timestamp systems*, European J. Combin., 14 (1993), pp. 95–102.

[DS89] D. Dolev and N. Shavit, *Bounded concurrent time-stamp systems are constructible*, in Proc. 21st ACM Symposium on Theory of Computing, ACM, New York, 1989, pp. 454–465.

[DW92] C. Dwork and O. Waarts, *Simple and efficient bounded concurrent timestamping, or, bounded concurrent timestamp systems are comprehensible*!, in Proc. 24th

ACM Symposium on Theory of Computing, ACM, New York, 1992, pp. 655–666.

[DPHW92] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts, *Time lapse snapshots*, in Proc. Israel Symposium on the Theory of Computing and Systems, D. Dolev, Z. Galil, and M. Rodeh, eds., Lecture Notes in Comput. Sci. 601, Springer-Verlag, Berlin, 1992, pp. 154–170.

[DGS88] D. Dolev, E. Gafni, and N. Shavit, *Towards a non-atomic era: ℓ-exclusion as a test case*, in Proc. 20th ACM Symposium on Theory of Computing, ACM SIGACT, ACM, New York, 1988, pp. 78–92.

[Dij65] E. W. Dijkstra, *Solution of a problem in concurrent programming control*, Comm. Assoc. Comput. Mach., 8 (1965), p. 569.

[FLBB79] M. Fischer, N. Lynch, J. Burns, and A. Borodin, *Resource allocation with immunity to limited process failure*, in Proc. 20th Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1979, pp. 234–254.

[FLBB89] M. Fischer, N. Lynch, J. Burns, and A. Borodin, *Distributed fifo allocation of identical resources using small shared space*, ACM Trans. Programming Lang. Systems, 11 (1989), pp. 90–114.

[G92] R. Gawlick, *Concurrent timestamping made simple*, Masters thesis, Technical Report MIT/LCS/TR-556, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1992.

[GLS92] R. Gawlick, N. Lynch, and N. Shavit, *Concurrent time-stamping made simple*, in Proc. Annual Israel Symposium on Theory of Computing and Systems, D. Dolev, Z. Galil, and M. Rodeh, eds., Lecture Notes in Comput. Sci. 601, Springer-Verlag, Berlin, 1992, pp. 171–185.

[Her91] M. P. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Lang. Systems, 13 (1991), pp. 124–149.

[ICMT94] M. Inoue, W. Chen, T. Masuzawa and N. Tokura, *Linear-time snapshot using multi-writer multi-reader registers*, in Workshop on Distributed Algorithms, Springer-Verlag, Berlin, 1994, pp. 130–140.

[HW88] M. P. Herlihy and J. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Trans. Programming Lang. Systems, 12 (1990), pp. 463–492.

[IL93] A. Israeli and M. Li, *Bounded time stamps*, Distrib. Comput., 6 (1993), pp. 205–209.

[IP91] A. Israeli and M. Pinchasov, *A linear time bounded concurrent timestamp scheme*, Technical Report, Technion, Haifa, Israel, 1991.

[Kat78] H. Katseff, *A new solution to the critical section problem*, in Proc. 10th ACM Symposium on Theory of Computing, ACM, New York, 1978, pp. 86–88.

[Lam74] L. Lamport, *A new solution of Dijkstra's concurrent programming problem*, Comm. Assoc. Comput. Mach., 17 (1974), pp. 453–455.

[Lam77] L. Lamport, *Concurrent reading and writing*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 806–811.

[Lam86a] L. Lamport, *The mutual exclusion problem part* i: *A theory of interprocess communication*, J. Assoc. Comput. Mach., 33 (1986), pp. 313–326.

[Lam86b] L. Lamport, *The mutual exclusion problem part* ii: *Statement and solutions*, J. Assoc. Comput Mach., 33 (1986), pp. 327–348.

[Lam86c] L. Lamport, *On interprocess communication part* i: *Basic formalism*, Distrib. Comput., 1 (1986), pp. 77–85.

[Lam86d] L. Lamport, *On interprocess communication part* ii: *Algorithms*, Distrib. Computing, 1 (1986), pp. 86–101.

[LH89] E. A. Lycklama and V. Hadzilacos, *A first-come-first-served mutual exclusion algorithm with small communication variables*, ACM Trans. Programming Lang. Systems, 13 (1991), pp. 558–576.

[LT87] N. Lynch and M. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, Tecnical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[LTV96] M. Li, J. Tromp, and P. Vitanyi, *How to share concurrent waitfree variables*, J. Assoc. Comput. Mach., 43 (1996), pp. 723–746 (journal version of [LV87]).

[LV87] M. Li and P. Vitanyi, *A very simple construction for atomic multiwriter registers*, Report, Aiken Computation Laboratory, Harvard University, Cambridge, MA, 1987.

[New87] R. Newman-Wolfe, *A protocol for waitfree atomic multi-reader shared variables*, in Proc. 6th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1987, pp. 232–248.

[PB87] G. L. Peterson and J. Burns, *Concurrent reading while writing* ii: *The multi-writer case*, in Proc. 28th Symposium on Foundations of Computer Science, IEEE

Computer Society Press, Los Alamitos, CA, 1987, pp. 383–392.

[Pet81] G. L. Peterson, *Myths about the mutual exclusion problem*, Inform. Process. Lett., 12 (1981), pp. 115–116.

[Pet83] G. Peterson, *Concurrent reading while writing*, ACM Trans. Programming Lang. Systems, 1 (1983), pp. 46–55.

[Pet88] G. Peterson, personal communication, 1988.

[Ray86] M. Raynal, *Algorithms for Mutual Exclusion*, North Oxford Academic Publishing, Oxford, UK and MIT Press, Cambridge, MA, 1986; originally published as *Algorithmique du Parallélisme*, Dunod Informatique, Paris, 1984 (in French; translated by D. Beeson).

[SAG94] A. Singh, J. Anderson, and M. Gouda, *The elusive atomic register*, J. Assoc. Comput. Mach., 41 (1994), pp. 311–339; original version in Proc. 6th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1987, pp. 206–221.

[Sch88] R. Schaffer, *On the correctness of atomic multi-writer registers*, Bachelor's thesis, Technical Memo MIT/LCS/TM-364, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[Sha90] N. Shavit, *Concurrent time-stamping*, Ph.D. thesis, School of Mathematics and Computer Science, Hebrew University, Jerusalem, 1990.

[SZ91] M. Saks and F. Zaharoglou, *Optimal space distributed move-to-front lists*, in Proc. 10th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1991, pp. 65–73.

[VA86] P. Vitanyi and B. Awerbuch, *Shared register access by asynchronous hardware*, in Proc. 27th Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1986, pp. 233–243.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 456–483, March 1997

008

PROBE ORDER BACKTRACKING*

PAUL WALTON PURDOM, JR.† AND G. NEIL HAVEN‡

Abstract. The algorithm for constraint-satisfaction problems, Probe Order Backtracking, has an average running time much faster than any previously analyzed algorithm under conditions where solutions are common. The algorithm uses a probing assignment (a preselected test assignment to unset variables) to help guide the search for a solution. If the problem is not satisfied when the unset variables are temporarily set to the probing assignment, the algorithm selects one of the relations which is not satisfied by the probing assignment and selects an unset variable which affects the value of that relation. It then does a backtracking (splitting) step, where it generates subproblems by setting the selected variable each possible way. Each subproblem is simplified and then solved recursively. For random problems with v variables, t clauses, and probability p that a literal appears in a clause, the average time for Probe Order Backtracking is no more than v^n when $p \geq (\ln t)/v$ plus lower-order terms. The best previous result was $p \geq \sqrt{(\ln t)/v}$. When the algorithm is combined with an algorithm of Franco that makes selective use of resolution, the average time for solving random problems is no more than v^n for all values of p when $t \leq O(n^{1/3}(v/\ln v)^{2/3})$. The best previous result was $t \leq O(n^{1/3}(v/\ln v)^{1/6})$. Probe Order Backtracking also runs in polynomial average time when $p \leq 1/v$, compared with the best previous result of $p \leq 1/(2v)$. With Probe Order Backtracking, the range of p that leads to more than polynomial time is much smaller than that for previously analyzed algorithms.

Key words. average time, backtracking, combinatorial search, NP-complete, satisfiability, searching

AMS subject classifications. 03B05, 05A16, 68Q20, 68Q25, 68T15

PII. S0097539793256053

1. Backtracking.

The constraint-satisfaction problem is to determine whether a set of constraints over discrete variables can be satisfied. Each constraint must have a form that is easy to evaluate, so any difficulty in solving such a problem comes from the interaction between the constraints and the need to find a setting for the variables that simultaneously satisfies all of the constraints.

Constraint-satisfaction problems are extremely common. Indeed, the proof that a problem is NP-complete implies an efficient way to transform the problem into a constraint-satisfaction problem. Most NP-complete problems are initially stated as constraint-satisfaction problems. A few special forms of constraint-satisfaction problems have known algorithms that solve problem instances in polynomial worst-case time. However, for the general constraint-satisfaction problem, no known algorithm is fast for the worst case. Nonetheless, many instances of the problem can be solved rapidly.

When no polynomial-time algorithm is known for a particular form of constraint-satisfaction problem, it is common practice to solve problem instances with a search algorithm. The basic idea of searching is to choose a variable and generate subproblems by assigning each possible value to the variable. In each subproblem, the relations are simplified by plugging in the value of the selected variable. This step of generating simplified subproblems is called *splitting*. If any subproblem has a solution, then the original problem has a solution. Otherwise, the original problem has no

* Received by the editors September 22, 1993; accepted for publication (in revised form) May 24, 1995. This research was partially supported by NSF grants CCR 92-03942 and CCR 94-02780.
http://www.siam.org/journals/sicomp/26-2/25605.html

† Computer Science Department, Indiana University, Lindley Hall, Bloominton, IN 47405-4101 (pwp@cs.indiana.edu).

‡ Autospect Inc., 4750 Venture Drive, Ann Arbor, MI 48108-9559.

solution. Subproblems that are simple enough (such as those with no unset variables) are solved directly. More complex subproblems are solved by applying the technique recursively.

If a problem contains the always *false* relation, then the problem has no solution. *Simple Backtracking* improves over plain search by immediately reporting no solution for such problems. Backtracking often saves a huge amount of time.

2. Probe Order Backtracking. This research is concerned with an algorithm that is an improvement over Simple Backtracking. A key idea in the algorithm is *probing*: if a fixed assignment to the unset variables solves the problem, no additional investigation is needed. An algorithm probes by setting each unset variable to some preselected value and testing to see whether all relations simplify to *true*. For random problems, one may as well use the value *false* for all variables. For practical problems, one could use a randomly selected probe sequence.

The algorithm *Backtracking with Probing* uses backtracking, probing, and no additional techniques. In particular, during splitting, it always picks the first unset variable from a fixed ordering on the variables. This algorithm is only a slight improvement over Simple Backtracking, and so it is not discussed further in this paper. The reader is referred to [19] for a detailed analysis.

A better algorithm, *Probe Order Backtracking*, is more sophisticated in its variable selection. It has a fixed ordering on the variables and a fixed ordering on the relations. First, it checks that there are no always *false* relations. If an always *false* relation is encountered, the problem is not satisfiable and the algorithm backtracks. Next, it checks to see if there is a currently selected relation. If there is no currently selected relation, it selects the first relation that evaluates to *false* under the probing assignment. (If all clauses evaluate to *true*, then the probing assignment solves the problem.) Finally, the algorithm does splitting using the first unset variable of the selected relation. The algorithm always finds a solution if there is one, it may find several solutions, but it normally does not find all solutions.

This paper shows that Probe Order Backtracking is fast for random problems drawn from the region of parameter space where solutions are common. It is also fast for random problems drawn from the region where solutions are uncommon. The algorithm probably uses exponential average time in a narrow region between those two regions, but this paper has only an upper-bound analysis, so it does not address that question.

3. Probability model. To measure the quality of a search algorithm, we use the average number of nodes in the backtrack tree that is generated when the algorithm solves a randomly generated problem. Our random problems are formed by the conjunction of independently generated random *clauses* (the logical *or* of literals, where a literal is a binary variable or the negation of a binary variable). A random clause is generated by independently selecting each literal with a fixed probability p. We use v for the number of variables and t for the number of clauses. (Some of the clauses may happen to be tautological.) For the asymptotic analysis, both p and t are functions of v.

Many algorithms have been analyzed with this random-clause-length model [3, 4, 7, 8, 9, 13, 15, 17, 18]. Most of these analyses and a few unpublished ones are summarized in [14].

A second probability model that is in common use is the fixed-clause model, where the significant difference is that all clauses have the same length. Much research has been done with the fixed-clause model, but only a few authors have done average

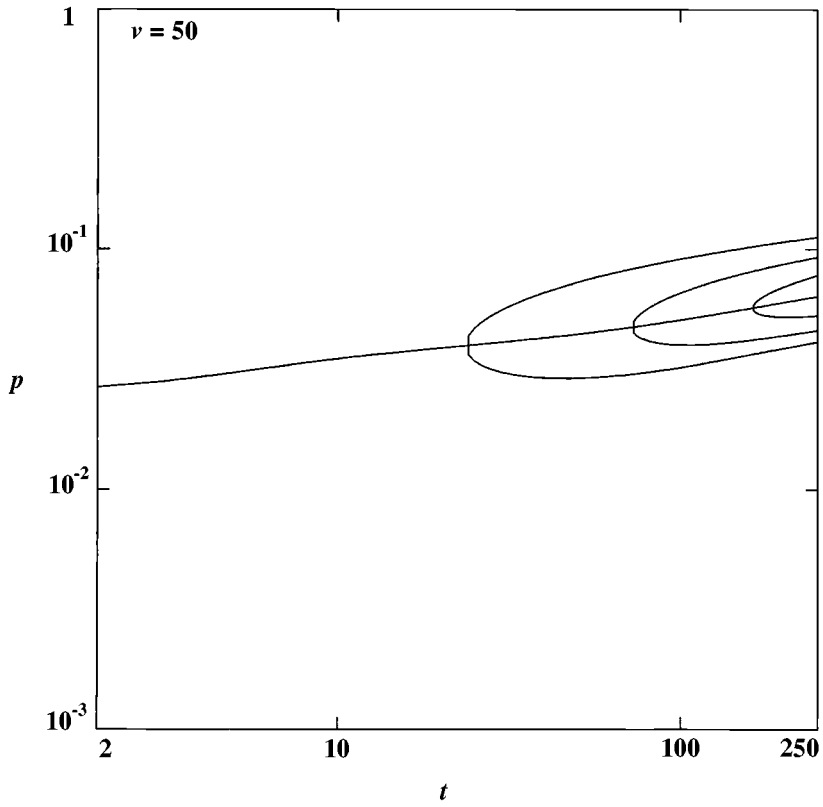


FIG. 1. *Probe Order Backtracking.*

time analyses for algorithms using this model [1, 2, 11, 16]. This second model generates problems that are more difficult to satisfy but perhaps more like the problems encountered in practice. It also leads to much more difficult analyses.

4. Summary of results. This section summarizes the performance of Probe Order Backtracking and gives some intuition as to why Probe Order Backtracking is fast.

Figure 1 is a contour plot showing the performance of Probe Order Backtracking for random problems with 50 variables. The vertical axis shows p, the probability that a given literal appears in a clause, running from 0.001 to 1 with ticks at 0.01 and 0.1. At $p = 0.01$, the average clause length for problems is 1. At $p = 0.1$, the average clause length is 10 literals. The horizontal axis shows t, the number of clauses, running from 2 to 250 with ticks at 10 and 100. When p is near 0 or 1, most problems are trivial. When p is low, most problems are easy because they contain an empty clause; empty clauses are trivially unsatisfiable. When p is high, most problems are easy because any assignment of values to variables is a solution to most problems. The region of hard problems lies in the middle.

The contours are shaped like elongated horseshoes. The area within a horseshoe contour represents problems that are more difficult than the problems outside the contour. The outermost contour shows where the average number of nodes required to solve a problem is 50, the next inner one 50^2, and finally 50^3. Running near the

centerline of the horseshoes is a line that shows for each t that value of p that results in the hardest problems (those with the largest number of nodes).

Except for the small t region, these contours are much better than those of any other algorithm for which such contours have been published. The improvement is particularly noticeable along the upper contour.

The asymptotic analysis of Probe Order Backtracking shows that the average number of nodes is no more than v^n, for large v and $n > 1$, when any of the following conditions hold:

$$(1) \quad p \geq \frac{\ln t + 2\ln\ln t - \ln\ln v - \ln(n-1) - \ln 2}{v} + \Theta\left(\frac{\ln\ln t}{v \ln t}\right),$$

$$(2) \quad p \leq \left\lceil \frac{\ln(t/v) + \ln\ln(t/v) + 1 - \ln 2 - \ln\ln 2}{2v} \right\rceil \times \left\lceil 1 + \frac{2(n-1)\ln v - (\ln 2)[\ln(t/v) + \ln\ln(t/v) + 3 - \ln 2 - \ln\ln 2]}{4v \ln 2} - \Theta\left(\frac{(\ln v)\ln(t/v)}{v^2}\right) - \Theta\left(\frac{\ln\ln(t/v)}{v\ln(t/v)}\right) \right\rceil,$$

$$(3) \quad p \leq \frac{1}{v} + \frac{e[(n-1)\ln v - \ln 2]}{tv},$$

$$(4) \quad t \leq \frac{(n-1)\ln v - \ln 2}{\ln\{1 + [(pv)^2 - 1]e^{-pv}/2\}}.$$

The Θ term in bound (1) is valid only when t increases more rapidly than $\ln v$. Bound (4) is valid when the limit of pv is greater than 1 and finite.

By setting p to minimize the right side of (4), we see that for all p and large v, the average number of nodes is no more than v^n when

$$(5) \quad t \leq 5.1150(n-1)\ln v - \Theta(1)$$

because in bound (4), the minimum value of the right side (when considered as a function of pv) occurs at $pv = 1 + \sqrt{2}$.

The bound for large p, bound (1), is much better than that for any previously analyzed algorithm. The best previous result was

$$(6) \quad p \geq \sqrt{\frac{\ln t - \ln n}{v}}$$

for Iwama's inclusion–exclusion algorithm [9]. In the region between bounds (1) and (6), Probe Order Backtracking is the fastest algorithm with proven results on its running time (as long as t is not small compared to v). Algorithms that repeatedly adjust variable settings to satisfy as many clauses as possible [20] are even faster on many problems. Those algorithms, however, have difficulty with problems which have no solution. They have been difficult to correctly analyze, and it is not clear at this time what their average running time is.

For the best previously analyzed algorithms, there was a large range of p where the algorithms apparently required more than polynomial time. (The word "apparently" is used because the analyses are all upper-bound analyses.) The ratio of the large p boundary to the small p boundary was $v^{1/2}$ times logarithmic factors. For Probe Order Backtracking, only the logarithmic factors are left. In some cases, even the logarithmic factors are gone and the ratio is constant (in the limit of large v). Bound (2) for small p results from the fact that the average number of nodes for Probe Order Backtracking is no larger than the average for Simple Backtracking. When $t = v^\alpha$ with $\alpha > 1$, the ratio of the upper boundary (1) to the lower boundary (2) is $2\alpha/(\alpha - 1)$ plus lower-order terms. Thus, for large α only a very limited range of p leads to problems with a large average time.

Perhaps the region of greatest interest is the one where t is proportional to v. When t is below $3.22135v$, bound (3) is better than (2). When $t = \beta v$, the ratio of the upper bound (1) to the first lower bound (2) is $(2 \ln v)/(\ln \beta + \ln \ln \beta + 1 - \ln 2 - \ln \ln 2)$ plus lower-order terms. When $t = \beta v$, the ratio of the upper bound to the second lower bound (3) is $\ln v$ plus lower-order terms.

Previously, for small t, the best algorithm was a combination of Franco's limited-resolution algorithm [8] for small p and Iwama's inclusion–exclusion algorithm [9] for large p. When p is unknown, the two algorithms can be run in parallel and stopped as soon as an answer is found. This combined algorithm generates no more than than v^n nodes (regardless of p) when $t = O(n^{1/3}(v/\ln v)^{1/6})$. Combining Franco's algorithm with Probe Order Backtracking improves the bound to $O(n^{1/3}(v/\ln v)^{2/3})$. The techniques of Franco's algorithm can be combined with Probe Order Backtracking, so there is no need to have two algorithms running in parallel. This is helpful when designing practical algorithms.

4.1. Informal discussion. The basic idea behind probing is old. The idea resembles that used by Newell and Simon in GPS [12]. Just as their program concentrates on differences between its current state and its goal state, Probe Order Backtracking focuses on a set of troublesome relations that are standing in the way of finding a solution. It is our impression that people who are good at solving puzzles use related ideas all the time.

Franco observed that two extremely simple algorithms could quickly solve most problems outside of a small range of p [6]. His algorithm for the region of high p does a single probe and gives up if no solution is found. His algorithm for the region of low p looks for an empty clause and gives up if there is none. Since Franco's algorithms sometimes give up, their average time is not well defined.

At the time of Franco's work, it was already known that Simple Backtracking was fast along the lower boundary (2), but it was not clear how to obtain an algorithm with a fast average time along the upper boundary (1). Simple uses of probing did not seem to lead to a good average time. Probe Order Backtracking was discovered while considering Franco's results [6] and considering the measurements of Sosič and Gu [20] for algorithms that concentrate on adjusting values until a solution is found. Both of those algorithms have difficulty with problems that have no solution.

Simple Backtracking improves over plain search by noticing when a problem has no solution due to the presence of an empty clause. However, Simple Backtracking is unfocused in its variable selection. As long as a problem does not have an empty clause, Simple Backtracking always proceeds by selecting the next splitting variable from a fixed ordering. The Clause Order Backtracking algorithm [3] improves over Simple Backtracking by focusing on the variables in one clause of the problem at a

time. This method of searching has the advantage that it performs splitting on just those variables that actually appear in a problem.

The Clause Order Backtracking algorithm provides a framework for the construction of a probing algorithm that has good performance for a wide range of problems, including those with no solution. Probe Order Backtracking, like Clause Order Backtracking, focuses on the variables in one clause at a time. However, Probe Order Backtracking improves over Clause Order Backtracking by only selecting variables from clauses which are not satisfied by the probing assignment. These are the clauses standing in the way of finding a solution: the *troublesome* clauses. Because Probe Order Backtracking probes by setting all unset variables to *false*, the troublesome clauses for Probe Order Backtracking are the clauses containing no negated variables.

When solutions are common, the informal explanation of why the algorithm is fast is that there will be only a few relations that are not satisfied by the probing assignment. Indeed, for $pv > \ln t$, the expected number of troublesome clauses for any given probing sequence is less than $1 - 1/t$. (See section 7.1.1 below.) By setting a variable so as to satisfy a troublesome clause, you definitely eliminate that clause. If luck is with you, you do not create any new troublesome clauses. The analysis at the end of the paper shows that, when solutions are common, setting variables so as to satisfy troublesome clauses creates new troublesome clauses at a rate slow enough that the algorithm is fast.

When solutions are uncommon, the algorithm is fast because it usually works on clauses that are shorter than average. First, the algorithm keeps working on the selected clause until it is satisfied—once the first variable has been set, the average length of the selected clause is less than that of a random clause. Second, troublesome clauses are smaller than average. The first reason is the same reason that Clause Order Backtracking is fast when solutions are uncommon [3]. (That algorithm is essentially Probe Order Backtracking without the probing.)

In the worst case, Probe Order Backtracking may need to try almost every combination of values for the variables. Thus the average-case performance of the algorithm is extremely good, but its worst-case performance is probably not a significant improvement over previous algorithms.

The analysis of the simpler Backtracking with Probing algorithm, which uses probing to test for a solution but not to select the variables to set [19], shows that a naïve application of probing does not lead to fast average time for the region of high p or for the region of low t. For good performance, it appears to be essential that an algorithm use probing both to notice when there is a solution and to focus its effort on the clauses which are interfering with solving the problem.

5. Practical algorithms. Probe Order Backtracking was studied in part because it is simple enough to analyze. In practice, one wants an algorithm that is fast whether or not it is possible to analyze its running time. There are several improvements that would clearly improve Probe Order Backtracking's average speed even though it is difficult to analyze their precise effectiveness:

1. Stop the search as soon as one solution is found. The analysis suggests that this would greatly improve the speed near the upper boundary (1), but stopping at the first solution leads to statistical dependencies that are difficult to analyze.

2. Carefully choose the probing sequence instead of just setting all variables to a fixed value. Various greedy approaches where variables are set to satisfy as many clauses as possible should be considered (see [10, 20]). This is particularly important near the upper boundary (1).

3. Probe with several sequences at one time. See [5, p. 151] for an algorithm that used two sequences. This is helpful along the upper boundary.

4. Carefully select which variable to set. The analysis suggests that this is particularly important along the lower boundary. Variables in hard-to-satisfy relations (short clauses) are more important than those in easy-to-satisfy relations. Variables that appear in lots of relations are more important than those that appear in a few relations. Apparently, when the relations are clauses, it is helpful to consider the number of clauses containing a particular variable positively and the number containing it negatively [5]. It appears that variable selection was a major factor in determining the order of placement of winning entries in a recent SAT competition [5].

5. Use resolution when it does not increase the problem size [8].

6. Algorithm statement. The precise form of Probe Order Backtracking that is analyzed and the rules for charging time are given below. The algorithm is specialized to work on satisfiability problems presented in conjunctive normal form (CNF). This version of the algorithm is specially tailored to accord with the mathematics of the equations in section 7 below.

A literal is *positive* if it is not in the scope of a *not* sign. It is *negative* if it is in the scope of a *not* sign. In the following algorithm, a variable can have the value *true*, *false*, or *unset*. The *positively augmented* current assignment is the current assignment of values to variables with the *unset* values changed to *true*. The *negatively augmented* current assignment is the current assignment of values to variables with the *unset* values changed to *false*.

The algorithm simplifies clauses by plugging in the values of the set variables so that (except when simplifying) it is concerned with only those variables that have the value *unset*. In this algorithm, the set of solutions is a global variable that is initially the empty set. Any solutions that are found are added to the set. If the problem has any solutions, at least one solution will be added to the set before the algorithm terminates. The algorithm may find more than one solution, but it does not in general find all solutions. If the problem has no solution, then the algorithm will terminate with an empty set of solutions.

Probe Order Backtracking for CNF Problems. Given a CNF predicate, perform a preprocessing step by determining whether the predicate has an empty clause. If it does, charge one time unit and return with an empty solution set. The following algorithm is never called on a predicate with an empty clause. Also, remove all tautological clauses from the problem.

1. (probe) If there are no all-positive clauses (that is, every clause has at least one negative literal), then return with the negatively augmented current assignment added to the set of solutions and charge one time unit.

2. (partial probe) If every clause of the CNF problem has only positive literals, then return with the positively augmented current assignment added to the set of solutions and charge one unit of time.

3. (select) Choose the first clause that is all-positive. Step 1 ensures that there is at least one such clause.

4. (splitting) Let k be the number of variables in the selected clause. (The lack of empty clauses ensures that $k \geq 1$, and step 3 ensures that each variable occurs in at most one literal of the clause.) For j starting at 1 and increasing to at most $k+1$, set the first $j-1$ variables of the clause so that their literals are *false* and charge one time unit. Use the assigned values to simplify the problem (remove each false

literal from its clause and remove from the problem each clause with a true literal). If setting the first $j-1$ literals of the selected clause to *false* results in some clause being empty, then stop the loop. If setting the first $j-1$ literals of the selected clause to *false* results in all clauses of the CNF problem having only positive literals, then add the positively augmented current assignment to the set of solutions and stop the loop. Generate a new subproblem by taking the simplified predicate obtained by setting the first $j-1$ literals of the selected clause to *false* and also setting the jth variable to *true* and simplifying. (This assignment satisfies the selected clause.) If this subproblem has an empty clause, charge one additional time unit. Otherwise, apply the algorithm recursively to the subproblem.

The cost in time units has been defined to be the same as the number of nodes in the backtrack tree generated by the algorithm. The actual running time of the algorithm depends on how cleverly it is implemented, but a good implementation will result in a time that is proportional to the number of nodes multiplied by a factor that is between 1 and tv, where v is the number of variables and t is the number of clauses.

The backtrack tree includes nodes for determining that the selected clause is empty. The computation associated with those nodes can be done quickly, so one might wish to have an upper limit of k on the time units for step 4. This would lead to small, unimportant changes in the analysis.

7. Exact analysis. The remainder of this paper consists of the analysis of the Probe Order Backtracking algorithm. The same analysis is presented in greater detail in [19]. An exact analysis of the Backtracking with Probing algorithm is also contained in [19].

We now derive recurrence equations which give exact values of the average number of nodes generated by Probe Order Backtracking.

7.1. Basic probabilities. For the analysis of probing algorithms it is useful to divide clauses into the following categories: *empty* (no literals), *all-positive* (1 or more positive literals and no negative ones), *tautological* (a positive and negative literal for the same variable, possibly with additional literals), and *mixed* (any clause that does not fall into one of the preceding categories). The mixed clauses have at least one negative literal and zero or more positive literals. Assigning values to some variables and then simplifying the clause may change the category of a clause, or it may result in the clause becoming *satisfied*. (Note that empty clauses remain empty and all-positive clauses never become mixed clauses.)

The probability that a random clause formed from v variables is nontautological, contains j positive literals, and contains k negative literals is

$$P(v,j,k) = \binom{v}{j,k,v-j-k} p^{j+k}(1-p)^{2v-j-k}. \tag{7}$$

The probability that a random clause has no literals is

$$P(v,0,0) = (1-p)^{2v}. \tag{8}$$

Note that

$$\sum_{j,k} P(v,j,k) = (1-p^2)^v \le 1 \tag{9}$$

because tautological clauses are not counted in the double sum.

Suppose a random clause is formed from v variables and then one of the variables is selected at random. The probability that the clause has a particular value of j and k (implying that it is not a tautology) and that the selected variable appears in the indicated way is

$$\text{positive: } \frac{j}{v}P(v,j,k), \quad \text{negative: } \frac{k}{v}P(v,j,k), \quad \text{neither: } \frac{v-j-k}{v}P(v,j,k). \tag{10}$$

7.1.1. All-positive clauses. The probability that a random clause is all-positive is

$$\sum_{j\geq 1} P(v,j,0) = (1-p)^v[1-(1-p)^v]. \tag{11}$$

As p becomes large, all-positive clauses become rare. In particular, for $pv > \ln t$, the average number of all-positive clauses in a predicate is bounded by

$$t(1-p)^v[1-(1-p)^v] < t\left(1-\frac{\ln t}{v}\right)^v\left[1-\left(1-\frac{\ln t}{v}\right)^v\right] < 1-\frac{1}{t}.$$

Suppose clauses are generated at random until an all-positive clause is produced. The probability that the all-positive clause contains j literals is

$$A(v,j) = \frac{P(v,j,0)}{\sum_{j\geq 1}P(v,j,0)} = \binom{v}{j}\frac{p^j(1-p)^{v-j}}{1-(1-p)^v}. \tag{12}$$

If a random variable is assigned the value *true*, then an all-positive clause will either become satisfied or remain all-positive. The probability that the clause becomes satisfied is

$$\sum_j \frac{j}{v}A(v,j) = \frac{p}{1-(1-p)^v}. \tag{13}$$

The probability that the clause has length j and that it remains all-positive is

$$\frac{v-j}{v}A(v,j) = \binom{v-1}{j}\frac{p^j(1-p)^{v-j}}{1-(1-p)^v} = \frac{P(v-1,j,0)}{(1-p)^{v-2}[1-(1-p)^v]}. \tag{14}$$

If a random variable is assigned the value *false*, then an all-positive clause will either become empty or remain all-positive. The probability that the resulting clause becomes empty is

$$\frac{1}{v}A(v,1) = \frac{p(1-p)^{v-1}}{1-(1-p)^v}. \tag{15}$$

The probability that the resulting clause remains all-positive and that it has length $j \geq 1$ is

$$\frac{j+1}{v}A(v,j+1) + \frac{v-j}{v}A(v,j) = \frac{P(v-1,j,0)}{(1-p)^{v-1}[1-(1-p)^v]}. \tag{16}$$

The average length of a random all-positive clause is

$$\sum_j jA(v,j) = \frac{pv}{1-(1-p)^v}. \tag{17}$$

7.1.2. Mixed clauses. The probability that a random clause is a mixed clause is

$$\sum_{\substack{j\geq 0\\ k\geq 1}} P(v,j,k) = (1-p)^v[(1+p)^v - 1]. \tag{18}$$

Suppose clauses are generated at random until a mixed clause is produced. The probability that the mixed clause contains j positive literals and $k \geq 1$ negative literals is

$$M(v,j,k) = \frac{P(v,j,k)}{\sum_j \sum_{k\geq 1} P(v,j,k)} = \binom{v}{j,k,v-j-k} \frac{p^{j+k}(1-p)^{v-j-k}}{(1+p)^v - 1}. \tag{19}$$

If a random variable is assigned the value *true*, a mixed clause may become empty, become all-positive, become satisfied, or remain a mixed clause. The probability that a mixed clause becomes an empty clause is

$$\frac{1}{v} M(v,0,1) = \frac{p(1-p)^{v-1}}{(1+p)^v - 1}. \tag{20}$$

The probability that it becomes an all-positive clause with length j is

$$\frac{1}{v} M(v,j,1) = \binom{v-1}{j} \frac{p^{j+1}(1-p)^{v-j-1}}{(1+p)^v - 1} = \frac{pP(v-1,j,0)}{(1-p)^{v-1}[(1+p)^v - 1]}. \tag{21}$$

The probability that it is satisfied is

$$\sum_j \sum_{k\geq 1} \frac{j}{v} M(v,j,k) = \frac{p[(1+p)^{v-1} - 1]}{(1+p)^v - 1}. \tag{22}$$

The probability that the mixed clause remains a mixed clause and that it has j positive literals and $k \geq 1$ negative literals is

$$\frac{k+1}{v} M(v,j,k+1) + \frac{v-j-k}{v} M(v,j,k) = \frac{P(v-1,j,k)}{(1-p)^{v-1}[(1+p)^v - 1]}. \tag{23}$$

If a random variable is assigned the value *false*, a mixed clause may become satisfied or remain a mixed clause. The probability that a mixed clause is satisfied is

$$\sum_j \sum_{k\geq 1} \frac{k}{v} M(v,j,k) = \frac{p(1+p)^{v-1}}{(1+p)^v - 1}. \tag{24}$$

The probability that the mixed clause becomes a mixed clause with j positive literals and $k \geq 1$ negative literals is

$$\frac{j+1}{v} M(v,j+1,k) + \frac{v-j-k}{v} M(v,j,k) = \frac{P(v-1,j,k)}{(1-p)^{v-1}[(1+p)^v - 1]}. \tag{25}$$

Equations (14), (16), (21), (23), and (25) show that in all cases where a nonempty clause results from setting a variable associated with a random nonempty clause generated from v variables, the resulting clause either is satisfied or has the same length distribution as random clauses generated from $v-1$ variables. Thus it is possible to base an analysis on the number of all-positive clauses, the number of mixed clauses, and the number of variables without having to contend with statistical dependencies.

7.1.3. Total number of nodes. Equation (8) implies that a random predicate with t clauses contains an empty clause (and is therefore solved with one node) with probability

$$1-[1-(1-p)^{2v}]^t. \tag{26}$$

Equations (8), (11), and (18) imply that the probability that a random predicate contains zero empty clauses, m all-positive clauses, n mixed clauses, and $t-m-n$ tautological clauses is

$$\binom{t}{m,n,t-m-n}(1-p)^{(n+m)v}[1-(1-p)^v]^m[(1+p)^v-1]^n[1-(1-p)^v(1+p)^v]^{t-m-n}. \tag{27}$$

If we let $T(v,m,n)$ be the average time required to solve a random problem with v variables, m all-positive clauses, n mixed clauses, and no empty clauses then by summing all of the cases we see that the expected number of nodes is

$$\begin{aligned} 1\ &-[1-(1-p)^{2v}]^t+\sum_{m,n}\binom{t}{m,n,t-m-n}(1-p)^{(n+m)v} \\ &\times[1-(1-p)^v]^m[(1+p)^v-1]^n[1-(1-p)^v(1+p)^v]^{t-m-n}T(v,m,n). \end{aligned} \tag{28}$$

7.1.4. Heuristic analysis. Before continuing with the exact analysis, we give a brief heuristic analysis, for the average time used by Probe Order Backtracking.

Ignore the fact that setting variables has an effect on clauses other than the selected clause. In particular, ignore the fact that the nonselected clauses can become empty or satisfied and ignore the fact that once the variables of one clause are set, there could be fewer variables waiting to be set in the remaining clauses. Under this radical assumption, the number of subproblems produced by splitting on the variables of the selected clause is the same as the length of the selected clause. If each of m clauses contains w variables, the total number of nonroot nodes in the implied search tree satisfies the recurrence

$$N(m)=wN(m-1)+2w.$$

"$2w$" is the number of nodes arising from setting each variable in the clause (one node for *true* and one for *false*) as specified in the Probe Order Backtracking algorithm. There are w subproblems produced by splitting. Each subproblem has $m-1$ clauses. The solution to this recurrence is

$$2\left[\frac{w^{m+1}-1}{w-1}\right]-2.$$

Add 1 for the root node and use equation (17) for w. Under these assumptions, $T(v,m,n)$ is given by

$$2\left\{\left[\left(\frac{pv}{1-(1-p)^v}\right)^{m+1}-1\right]\Big/\left[\left(\frac{pv}{1-(1-p)^v}\right)-1\right]\right\}-1. \tag{29}$$

Plugging into (28) and summing over m and n gives an average number of nodes of

$$1+\left(\frac{2pv}{pv-1+(1-p)^v}\right)\{[1+(pv-1)(1-p)^v]^t-[1-(1-p)^{2v}]^t\}. \tag{30}$$

The heuristic analysis ignores three aspects of setting variables: 1. a clause other than the selected clause can become empty, 2. a clause other than the selected clause can be satisfied, and 3. a mixed clause can become an all-positive clause. The first leads to a predicted answer that is too large—particularly for small p. The last two effects come close to canceling, but for large p they can lead to the predicted value being either too large or too small depending on the exact value of the parameters. Curves for the heuristic analysis are given in [19].

This rough and not quite correct analysis is useful for obtaining a general understanding of the performance of the algorithm before completing a correct analysis. In particular, the heuristic analysis given here is useful in guessing the form of the dependence of the exact analysis on m, the number of all-positive clauses. See the value selected for $x(v)$ in section 8.3.

7.1.5. Transition probabilities. Suppose a predicate is produced by repeatedly generating random clauses from v variables. Suppose the resulting predicate contains m all-positive clauses, n mixed clauses, and no empty clauses.

Let $G(v,n)$ be the probability that setting a random variable to *true* results in the predicate having one or more empty clauses. When a variable is set to *true*, mixed clauses become empty with the probability given in equation (20) while all-positive clauses do not become empty. Therefore,

$$G(v,n) = 1 - \left(1 - \frac{p(1-p)^{v-1}}{(1+p)^v - 1}\right)^n. \tag{31}$$

Let $F(v,m)$ be the probability that setting a random variable to *false* results in a predicate with one or more empty clauses. Equation (15) implies

$$F(v,m) = 1 - \left(1 - \frac{p(1-p)^{v-1}}{1-(1-p)^v}\right)^m. \tag{32}$$

Let $D(v,i,k,m,n)$ be the probability that setting i random variables to *false* results in no clauses becoming empty and k mixed clauses becoming satisfied. If $i = 0$, nothing happens, so

$$D(v,0,k,m,n) = \delta_{k0}. \tag{33}$$

For $i = 1$, equations (15) and (24) imply

$$D(v,1,k,m,n) = \binom{n}{k}\left(1 - \frac{p(1-p)^{v-1}}{1-(1-p)^v}\right)^m \left(\frac{p(1+p)^{v-1}}{(1+p)^v - 1}\right)^k \left(1 - \frac{p(1+p)^{v-1}}{(1+p)^v - 1}\right)^{n-k}. \tag{34}$$

For $i > 1$, some of the mixed clauses (x of them) must be satisfied when the first $i-1$ variables are set and then the rest ($k-x$) must be satisfied when the last variable is set, so D can be calculated from

$$D(v,i,k,m,n) = \sum_x D(v,i-1,x,m,n) D(v-i+1,1,k-x,m,n-x). \tag{35}$$

Since the m index is constant in this recurrence and

$$\prod_{0 \le j < i}\left(1 - \frac{p(1-p)^{v-j-1}}{1-(1-p)^{v-j}}\right) = \frac{1-(1-p)^{v-i}}{1-(1-p)^v}, \tag{36}$$

we have

$$D(v,i,k,m,n) = \left(\frac{1-(1-p)^{v-i}}{1-(1-p)^v}\right)^m D(v,i,k,n), \tag{37}$$

where

$$D(v,1,k,n) = \binom{n}{k}\left(\frac{p(1+p)^{v-1}}{(1+p)^v-1}\right)^k\left(1-\frac{p(1+p)^{v-1}}{(1+p)^v-1}\right)^{n-k} \tag{38}$$

and

$$D(v,i,k,n) = \sum_x D(v,i-1,x,n)D(v-i+1,1,k-x,n-x). \tag{39}$$

The solution to this recurrence is

$$D(v,i,k,n) = \binom{n}{k}\frac{[(1+p)^v-(1+p)^{v-i}]^k[(1+p)^{v-i}-1]^{n-k}}{[(1+p)^v-1]^n}. \tag{40}$$

Let $E(v,j,k,l,m,n)$ be the probability that setting one random variable to *true* results in no clauses becoming empty, j mixed clauses becoming all-positive clauses, k mixed clauses becoming satisfied, and l all-positive clauses becoming satisfied. (Notice that no other changes of clause category can occur.) From equation (21), the probability that a mixed clause will become all-positive is

$$\sum_{j\geq 1}\binom{v-1}{j}\frac{p^{j+1}(1-p)^{v-j-1}}{(1+p)^v-1} = \frac{p[1-(1-p)^{v-1}]}{(1+p)^v-1}. \tag{41}$$

Using this with Equations (13), (20), and (22) gives

(42)

$$\begin{aligned}
&E(v,j,k,l,m,n)\\
&= \binom{m}{l}\binom{n}{j,k,n-j-k}\left(\frac{p}{1-(1-p)^v}\right)^l\left(1-\frac{p}{1-(1-p)^v}\right)^{m-l}\\
&\quad\times\left(\frac{p[1-(1-p)^{v-1}]}{(1+p)^v-1}\right)^j\left(\frac{p[(1+p)^{v-1}-1]}{(1+p)^v-1}\right)^k\\
&\quad\times\left(1-\frac{p(1-p)^{v-1}}{(1+p)^v-1}-\frac{p[1-(1-p)^{v-1}]}{(1+p)^v-1}-\frac{p[(1+p)^{v-1}-1]}{(1+p)^v-1}\right)^{n-j-k}\\
&= \binom{m}{l}\binom{n}{j,k,n-j-k}\left(\frac{p}{1-(1-p)^v}\right)^l\left(1-\frac{p}{1-(1-p)^v}\right)^{m-l}\\
&\quad\times\left(\frac{p[1-(1-p)^{v-1}]}{(1+p)^v-1}\right)^j\left(\frac{p[(1+p)^{v-1}-1]}{(1+p)^v-1}\right)^k\left(1-\frac{p(1+p)^{v-1}}{(1+p)^v-1}\right)^{n-j-k}.
\end{aligned}$$

7.2. Recurrence. Probe Order Backtracking selects a clause and then sets the variables that occur in the clause. If the selected clause has h variables, then there is a root, a node from setting the first variable to *false*, a potential node from setting the first two variables to *false*, and so on. This gives a root plus up to h additional

nodes. In addition, there is a subtree for setting the first variable to *true*, potentially a subtree for setting the first variable to *false* and the second to *true*, and so on. When setting the first few variables, some of the mixed clauses may evaluate to *false*. Also, setting the first few variables may result in the number of mixed clauses dropping to zero. Either of these effects may prevent a potential node from occurring in the tree.

Define $a(v,i)$ as the probability that the selected clause contains i or more literals (thus potentially contributing an ith node to the backtrack tree). Then from equation (12), we obtain

$$a(v,i) = \sum_{j \geq i} A(v,j) = \sum_{j \geq i} \binom{v}{j} \frac{p^j (1-p)^{v-j}}{1-(1-p)^v}. \tag{43}$$

Let $T(v,m,n)$ be the average number of nodes for a problem that has v variables, m all-positive clauses, n mixed clauses, and no empty clauses. Then

(44)

$$\begin{aligned} T(v,m,n) &= 1 \\ &\quad + \sum_{1 \leq i \leq v} a(v,i) \sum_{x<n} D(v,i-1,x,m-1,n) \Big[1 + G(v-i+1,n-x) \\ &\quad + \sum_{j,k,l} E(v-i+1,j,k,l,m-1,n-x) T(v-i,m+j-l-1,n-j-k-x) \Big]. \end{aligned}$$

The initial 1 is for the root of the tree. The i index is for those nodes that occur as a result of setting the first i variables from the clause. The factor $a(v,i)$ gives the probability that the selected clause has at least i variables. The index x is for the number of mixed clauses that are satisfied when setting the first $i-1$ variables to *false*. The sum does not include $x = n$ because no subproblems are generated when the number of mixed clauses is reduced to zero. The factor $D(v,i-1,x,m-1,n)$ is the probability that x of the n mixed clauses become satisfied and no clauses become empty as a result of setting the first $i-1$ variables. The D factor multiplies the sum of terms that relate to the various kinds of nodes that can result when the ith variable is set. The 1 following the square bracket is for the node that results from setting the ith variable to *false*. The $G(v-i+1,n-x)$ term gives the probability that setting the ith variable to *true* produces an empty clause. When setting the ith variable to *true*, the j index counts the number of mixed clauses that become all-positive, the k index counts the number of mixed clauses that become satisfied, and the l index counts the number of all-positive clauses that are satisfied. (The selected clause is not included in this count.) The factor $E(v-i+1,j,k,l,m-1,n-x)$ is the probability that setting the ith variable results in the values j, k, and l. The factor $T(v-i,m+j-l-1,n-j-k-x)$ is the expected number of nodes in the subtree that results from setting the first $i-1$ variables to *false* and the ith variable to *true*.

If m or n is zero, then the algorithm stops immediately, so there is only one node. Thus

$$T(v,0,n) = T(v,m,0) = 1. \tag{45}$$

We now do a number of transformations to convert equation (44) into a more convenient form. Define

$$Y(v,i) = \frac{1-(1-p)^{v-i}}{1-(1-p)^v}. \tag{46}$$

Then equations (37) and (44) imply

$$\begin{aligned}T(v,m,n) = 1 + \sum_{1\le i\le v} a(v,i) \sum_{x<n} [Y(v,i-1)]^{m-1} D(v,i-1,x,n) \\ \times \Bigg[1 + G(v-i+1,1,n-x) \\ + \sum_{j,k,l} E(v-i+1,j,k,l,m-1,n-x) \\ \times T(v-i,m+j-l-1,n-j-k-x) \Bigg].\end{aligned}$$

Change indices with $j' = m+j-l-1$ and $k' = n-j-k-x$ ($j = l+j'-m+1$, $k = n+m-l-x-j'-k'-1$) and drop primes to obtain

(47)
$$\begin{aligned}T(v,m,n) = 1 + \sum_{1\le i\le v} a(v,i) \sum_{x<n} [Y(v,i-1)]^{m-1} D(v,i-1,x,n) \\ \times \Bigg[1 + G(v-i+1,1,n-x) \\ + \sum_{j,k,l} E(v-i+1,l+j-m+1,n+m-l-x-j-k-1,l,m-1,n-x) T(v-i,j,k) \Bigg].\end{aligned}$$

Define

(48)
$$\begin{aligned}Z(v,i,m,n) &= \sum_{x<n} [Y(v,i-1)]^{m-1} D(v,i-1,x,n)[1+G(v-i+1,1,n-x)] \\ &= [Y(v,i-1)]^{m-1} \\ &\quad \times \left[2 - \left(\frac{(1+p)^v - (1+p)^{v-i+1}}{(1+p)^v - 1} \right)^n - \left(\frac{(1+p)^v - 1 - p(1-p)^{v-i}}{(1+p)^v - 1} \right)^n \right]\end{aligned}$$

and

$$\begin{aligned}&H(v,i,j,k,m,n) \\ &\quad = \sum_{x<n} [Y(v,i-1)]^{m-1} D(v,i-1,x,n) \\ &\qquad \times \sum_{l} E(v-i+1,l+j-m+1,n+m-l-x-j-k-1,l,m-1,n-x).\end{aligned}$$

After doing the sum over x and canceling some factors (one page of algebra), one has

(49)
$$\begin{aligned}H(v,i,j,k,m,n) = \left(\frac{(1+p)^{v-i}-1}{(1+p)^v - 1} \right)^k \sum_l \binom{m-1}{l} \binom{n}{l+j-m+1} \\ \times \binom{n+m-l-j-1}{k} \left(\frac{p}{1-(1-p)^v} \right)^l \\ \times \left(\frac{1-(1-p)^{v-i+1}-p}{1-(1-p)^v} \right)^{m-l-1} \left(\frac{p[1-(1-p)^{v-i}]}{(1+p)^v - 1} \right)^{l+j-m+1}\end{aligned}$$

$$\times \left(\frac{(1+p)^v - (1+p)^{v-i} - p}{(1+p)^v - 1} \right)^{n+m-l-j-k-1}$$
$$- \delta_{k0} \left(\frac{(1+p)^v - (1+p)^{v-i+1}}{(1+p)^v - 1} \right)^n$$
$$\times \binom{m-1}{j} \left(\frac{1-(1-p)^{v-i+1} - p}{1-(1-p)^v} \right)^j \left(\frac{p}{1-(1-p)^v} \right)^{m-1-j}.$$

With these definitions, equation (47) can be written as

$$T(v,m,n) = 1 + \sum_{1 \le i \le v} a(v,i) \left(Z(v,i,m,n) + \sum_{j,k} H(v,i,j,k,m,n) T(v-i,j,k) \right). \tag{50}$$

Equation (50) is suitable for the derivation of the asymptotic number of nodes, but it requires time $O(v^2t^4)$ to evaluate. In [19], an alternate four-index recurrence is given which can be evaluated in time $O(vt^4 + v^2t^3)$.

7.3. Verification of the recurrences. Aside from being careful with the mathematics, we performed measurements to help insure the correctness of the analysis of Probe Order Backtracking.

For t and v in the range $1 \le v \le 6$, $1 \le t \le 6$, $1 \le tv \le 12$, we generated each of the 2^{2tv} SAT problems and counted the number of nodes produced. A problem with i literals has probability $p^i(1-p)^{2tv-i}$. Multiplying the node counts for each i by this probability gives a polynomial in p with integer coefficients [3]. We used Maple V to solve (50) algebraically and verified that the polynomials from the recurrence were identical with the polynomials generated from the corresponding node counts.

8. Bounds. We are interested in values for t and p for which the running time is polynomial in v. Thus we now compute a simple upper bound on the number of nodes and compare this bound with v^n as v becomes large.

Our approach is to eliminate indices from the recurrence until we obtain a linear first-order recurrence. To eliminate an index, we assume that the unknown function (T) has a particular dependence on the index being eliminated times a new unknown function of the remaining indices. By plugging the assumed form into the initial recurrence (and performing one or two summations), we obtain a bounding recurrence for the new function.

To simplify the algebra, we now drop the term that starts with δ_{k0} from the definition of H (in equation (49)) and drop the first negative term from the definition of Z (in equation (48)). These changes lead to a new $T(v,m,n)$, which is an upper bound on the number of nodes. The terms have no significant effect when v is large. Dropping them now saves a lot of ink.

It is convenient to first shift the recurrence by using $T'(v,m,n) = T(v,m,n) - 1$. From equation (50), we obtain

(51)
$$\begin{aligned} & T'(v,m,n) \\ &= \sum_{1 \le i \le v} a(v,i) \left(Z(v,i,m,n) + \sum_j \sum_k H(v,i,j,k,m,n)[T'(v-i,j,k)+1] \right) \\ &= \sum_{1 \le i \le v} a(v,i) \left[Z(v,i,m,n) + \left(\frac{1-(1-p)^{v-i+1}}{1-(1-p)^v} \right)^{m-1} \left(\frac{(1+p)^v - 1 - p(1+p)^{v-i}}{(1+p)^v - 1} \right)^n \right. \\ &\qquad \left. + \sum_j \sum_k H(v,i,j,k,m,n) T'(v-i,j,k) \right], \end{aligned}$$

which can be written as

(52)
$$T'(v,m,n) = \sum_{1\le i\le v} a(v,i)\biggl(Z'(v,i,m,n) + \sum_j \sum_k H(v,i,j,k,m,n)T'(v-i,j,k)\biggr),$$

where

$$Z'(v,i,m,n) = 2\left(\frac{1-(1-p)^{v-i+1}}{1-(1-p)^v}\right)^{m-1}. \tag{53}$$

The boundary conditions for the shifted recurrence are

$$T'(v,0,n) = T'(v,m,0) = 0, \tag{54}$$

and the average number of nodes is

$$\begin{aligned} &1 + \sum_{m,n} \binom{t}{m,n,t-m-n}(1-p)^{(n+m)v} \\ &\quad\times [1-(1-p)^v]^m[(1+p)^v-1]^n[1-(1-p)^v(1+p)^v]^{t-m-n}T'(v,m,n). \end{aligned} \tag{55}$$

8.1. Two-index recurrence. For any $x(v)$, define $T(v,n)$ so that $T'(v,m,n) \le x(v)^m T(v,n)$ for all m. (We could include an n dependence in x, but that does not appear to be useful.) Rearrange the binomials to obtain

$$\begin{aligned} H(v,i,j,k,m,n) = \binom{n}{k}&\left(\frac{(1+p)^{v-i}-1}{(1+p)^v-1}\right)^k \sum_l \binom{m-1}{l}\binom{n-k}{l+j-m+1} \\ &\times \left(\frac{p}{1-(1-p)^v}\right)^l \left(\frac{1-(1-p)^{v-i+1}-p}{1-(1-p)^v}\right)^{m-l-1} \\ &\times \left(\frac{p[1-(1-p)^{v-i}]}{(1+p)^v-1}\right)^{l+j-m+1} \\ &\times \left(\frac{(1+p)^v-(1+p)^{v-i}-p}{(1+p)^v-1}\right)^{n+m-l-j-k-1}. \end{aligned} \tag{56}$$

Equation (52) implies that we can obtain our bound by requiring

$$\begin{aligned} &x(v)^m T(v,n) \\ &\quad\ge \sum_{1\le i\le v} a(v,i)\biggl(Z'(v,i,m,n) + \sum_j \sum_k x(v-i)^j H(v,i,j,k,m,n)T(v-i,k)\biggr), \end{aligned}$$

$$T(v,0) = 0, \qquad T(v,n) \ge 0,$$

where the bounds must hold for all m of interest. Thus

$$\begin{aligned} T(v,n) = \max_m \biggl\{ \frac{1}{x(v)^m}\biggl[\sum_{1\le i\le v} a(v,i)\biggl(&Z'(v,i,m,n) \\ &+ \sum_j \sum_k x(v-i)^j H(v,i,j,k,m,n)T(v-i,k)\biggr)\biggr]\biggr\}, \end{aligned} \tag{57}$$

$$T(v,0)=0.$$

Summing x^jH over j gives

$$\begin{aligned}\sum_j x^jH(v,i,j,k,m,n)&\\ =\left(\frac{x[1-(1-p)^{v-i+1}]+(1-x)p}{1-(1-p)^v}\right)^{m-1}&\binom{n}{k}\left(\frac{(1+p)^{v-i}-1}{(1+p)^v-1}\right)^k\\ \times\left(\frac{(1+p)^v-(1+p)^{v-i}-(1-x)p-xp(1-p)^{v-i}}{(1+p)^v-1}\right)^{n-k}&.\end{aligned}\tag{58}$$

Using this sum and the definition of Z' in equation (57) gives

$$\begin{aligned}T(v,n)&\\ =\frac{1}{x(v)}\max_m\Bigg\{\sum_{1\le i\le v}a(v,i)\Bigg[&2\left(\frac{1-(1-p)^{v-i+1}}{x(v)[1-(1-p)^v]}\right)^{m-1}\\ &+\left(\frac{x(v-i)[1-(1-p)^{v-i+1}]+[1-x(v-i)]p}{x(v)[1-(1-p)^v]}\right)^{m-1}\\ &\times\sum_k\binom{n}{k}\left(\frac{(1+p)^{v-i}-1}{(1+p)^v-1}\right)^k\\ &\times\left(\frac{(1+p)^v-(1+p)^{v-i}-[1-x(v-i)]p-x(v-i)p(1-p)^{v-i}}{(1+p)^v-1}\right)^{n-k}\\ &\times T(v-i,k)\Bigg]\Bigg\},\end{aligned}\tag{59}$$

$$T(v,0)=0,\qquad T(v,n)\ge 0.\tag{60}$$

So that this recurrence will be favorable, we wish to avoid raising quantities that are above 1 to the mth power. Thus we require

$$[1-(1-p)^v]x(v)\ge 1-(1-p)^{v-i+1}\tag{61}$$

and

$$[1-(1-p)^v]x(v)\ge[1-(1-p)^{v-i+1}]x(v-i)+[1-x(v-i)]p.\tag{62}$$

As long as $x(v)$ is above 1, any increasing function of v can be chosen for $[1-(1-p)^v]x(v)$.

If $x(v)$ obeys the bounds (61) and (62), then we have

$$\begin{aligned}T(v,n)=\frac{1}{x(v)}\Bigg[\sum_{1\le i\le v}a(v,i)\Bigg(&2+\sum_k\binom{n}{k}\left(\frac{(1+p)^{v-i}-1}{(1+p)^v-1}\right)^k\\ \times\left(\frac{(1+p)^v-(1+p)^{v-i}-[1-x(v-i)]p-x(v-i)p(1-p)^{v-i}}{(1+p)^v-1}\right)^{n-k}&T(v-i,k)\Bigg)\Bigg].\end{aligned}\tag{63}$$

(55) implies that the average number of nodes is bounded by

(64)
$$1+\sum_{m,n}\binom{t}{m,n,t-m-n}(1-p)^{(n+m)v}x(v)^m[1-(1-p)^v]^m \times[(1+p)^v-1]^n[1-(1-p)^v(1+p)^v]^{t-m-n}T(v,n)$$
$$=1+\sum_n\binom{t}{n}(1-p)^{nv}[(1+p)^v-1]^n \times\{1-(1-p)^v(1+p)^v+x(v)(1-p)^v[1-(1-p)^v]\}^{t-n}T(v,n).$$

Equation (64) gives a good bound when $x(v)$ is set to the average length of an all-positive clause, equation (17). (See Figure 5 in [19].)

Note the division by $x(v)$ in equation (63). This is critical to obtaining an analytical understanding of why Probe Order Backtracking is fast. We are free to set $x(v)$ large enough to cancel out the effect of summing over i (which is where the growth in $T(v,n)$ comes from) as long as the factor in equation (64) which is raised to the power $t-n$ is not above 1. This division by $x(v)$ is related to the fact that selecting an all-positive clause results in a reduction of one in the number of all-positive clauses. (The setting of variables can augment or counteract this reduction.)

8.2. One-index recurrence. For any $x(v)$ and $y(v)$, define $T(v)$ so that $T'(v,m,n)\le x(v)^m y(v)^n T(v)$ for all m and n. Summing x^jy^kH over j and k gives

$$\begin{aligned}&\sum_{j,k}x^jy^kH(v,i,j,k,m,n)\\ &\quad=\sum_k y^k\left(\frac{x[1-(1-p)^{v-i+1}]+(1-x)p}{1-(1-p)^v}\right)^{m-1}\binom{n}{k}\left(\frac{(1+p)^{v-i}-1}{(1+p)^v-1}\right)^k\\ &\qquad\times\left(\frac{(1+p)^v-(1+p)^{v-i}-(1-x)p-xp(1-p)^{v-i}}{(1+p)^v-1}\right)^{n-k}\\ &\quad=\left(\frac{x[1-(1-p)^{v-i+1}]+(1-x)p}{1-(1-p)^v}\right)^{m-1}\\ &\qquad\times\left(\frac{(1+p)^v+(y-1)(1+p)^{v-i}-y-(1-x)p-xp(1-p)^{v-i}}{(1+p)^v-1}\right)^n.\end{aligned}\tag{65}$$

Using this result in equation (59) implies that a suitable $T(v)$ is any function at least as large as the solution to

(66)
$$\begin{aligned}T(v)=\frac{1}{x(v)}\max_{m,n}\Bigg\{\sum_{1\le i\le v}a(v,i)\Bigg[&\frac{2}{y(v)^n}\left(\frac{1-(1-p)^{v-i+1}}{x(v)[1-(1-p)^v]}\right)^{m-1}\\ &+\left(\frac{x(v-i)[1-(1-p)^{v-i+1}]+[1-x(v-i)]p}{x(v)[1-(1-p)^v]}\right)^{m-1}\\ &\times\Bigg(\frac{(1+p)^v+[y(v-i)-1](1+p)^{v-i}}{y(v)[(1+p)^v-1]}\\ &-\frac{y(v-i)-[1-x(v-i)]p-x(v-i)p(1-p)^{v-i}}{y(v)[(1+p)^v-1]}\Bigg)^n T(v-i)\Bigg]\Bigg\}.\end{aligned}$$

Again, we wish to avoid raising quantities above 1 to high powers. Thus we still have the bounds (61) and (62) for $x(v)$. In addition, we have

$$y(v) \geq 1 \tag{67}$$

and

$$[y(v)-1][(1+p)^v-1]-[y(v-i)-1][(1+p)^{v-i}-1] \geq p\{x(v-i)[1-(1-p)^{v-i}]-1\}. \tag{68}$$

These bounds for y are satisfied by

$$[y(v)-1][(1+p)^v-1] = p \sum_{1\leq j\leq v-1} \max\{0, \{x(j)[1-(1-p)^j]-1\}\}. \tag{69}$$

If $x(v)$ and $y(v)$ obey the bounds, we have

$$T(v) = \frac{1}{x(v)} \sum_{1\leq i\leq v} a(v,i)[2+T(v-i)]. \tag{70}$$

Equation (64) implies that the average number of nodes is bounded by

$$\begin{aligned} &1+\sum_n \binom{t}{n} y(v)^n (1-p)^{nv}[(1+p)^v-1]^n \\ &\quad \times \{1-(1-p)^v(1+p)^v + x(v)(1-p)^v[1-(1-p)^v]\}^{t-n} T(v) \\ &= 1 + \\ &\{1-(1-p)^v(1+p)^v + y(v)(1-p)^v[(1+p)^v-1] + x(v)(1-p)^v[1-(1-p)^v]\}^t T(v). \end{aligned} \tag{71}$$

If the value of $y(v)$ is set by equation (69), then the number of nodes is bounded by

$$1+\left[1+(1-p)^v\left(x(v)[1-(1-p)^v]-1+p\sum_{1\leq j\leq v-1}\max\{0,(x(j)[1-(1-p)^j]-1)\}\right)\right]^t T(v). \tag{72}$$

(72) gives a good bound when $x(v)$ is set to the average length of an all-positive clause, equation (17). (See Figure 2 in [19].)

If one sets $x(v)$ to the average clause size and $y(v) = 1$ (ignoring the requirement that $y(v)$ satisfy bounds (67, 68)), one obtains a result that is essentially the same as that given by the heuristic analysis, eq. (30).

8.3. A simplification. Equation (70) has only one index, but it is still rather complex due to the summation on the right side. Define $U(0) = T(0)$ and

$$U(v) = \frac{2+U(v-1)}{x(v)} \sum_{1\leq i\leq v} a(v,i). \tag{73}$$

When $U(v)$ is nondecreasing, $T(v) \leq U(v)$. (In equation (70), we can obtain an upper bound by replacing $T(v-i)$ with an upper bound, i.e., with $U(v-1)$.) A good choice for $x(v)$ is one that cancels the effect of the summation. For

$$x(v) = \sum_{1\leq i\leq v} a(v,i) = \frac{pv}{1-(1-p)^v}, \tag{74}$$

we have

$$U(v) = 2 + U(v-1), \qquad U(0) = 0. \tag{75}$$

Thus

$$T(v) \le 2v. \tag{76}$$

When $x(v)$ is given by equation (74), we have

$$\begin{aligned}\sum_{1\le j\le v-1} \max\{x(j)[1-(1-p)^j]-1,0\} &= \sum_{1/p\le j\le v-1} (pj-1)\\ &\le \frac{pv(v-1)}{2} - \frac{(1/p-1)}{2} - \left(v-\frac{1}{p}\right).\end{aligned} \tag{77}$$

(The less than or equal comes from the fact that $1/p$ may be a noninteger.) Thus equations (72) and (77) imply that the number of nodes is bounded by

$$1 + 2v\left[1 + (1-p)^v\left(\frac{p^2v(v-1)}{2} - \frac{1-p}{2}\right)\right]^t. \tag{78}$$

Figure 2 shows the bounds that result from (78). The contour for v^1 nodes has two branches, the upper one along the $p = 1$ axis and the lower one shown on the figure. The remaining contours have about the same shape as the contours from the exact analysis (Figure 1), but they are fatter. Also, the v^4 contour is shown in Figure 2, whereas it occurs at too large of a t value to be shown in Figure 1.

In [19], we show that

$$x(v) = \frac{apv}{1-(1-p)^v} \tag{79}$$

with

$$a = \frac{2 + (1-p)^v[-1+p+2(t-1)pv + p^2tv(v-1)]}{p(t-1)v(1-p)^v[p(v-1)+2]} \tag{80}$$

gives a better bound, but we do not use that result here.

9. Asymptotics. For the asymptotic analysis, we require that the bound on the number of nodes be no more than v^n. That is,

$$v^n \ge 1 + 2v\left[1 + (1-p)^v\left(\frac{p^2v(v-1)}{2} - \frac{1-p}{2}\right)\right]^t \tag{81}$$

or

$$\frac{v^{n-1}}{2}\left(1 - \frac{1}{v^n}\right) \ge \left[1 + (1-p)^v\left(\frac{p^2v(v-1)}{2} - \frac{1-p}{2}\right)\right]^t. \tag{82}$$

9.1. Small *t*. From (82), we have

$$\ln\left[\frac{v^{n-1}}{2}\left(1 - \frac{1}{v^n}\right)\right] \ge t\ln\left[1 + (1-p)^v\left(\frac{p^2v(v-1)}{2} - \frac{1-p}{2}\right)\right]. \tag{83}$$

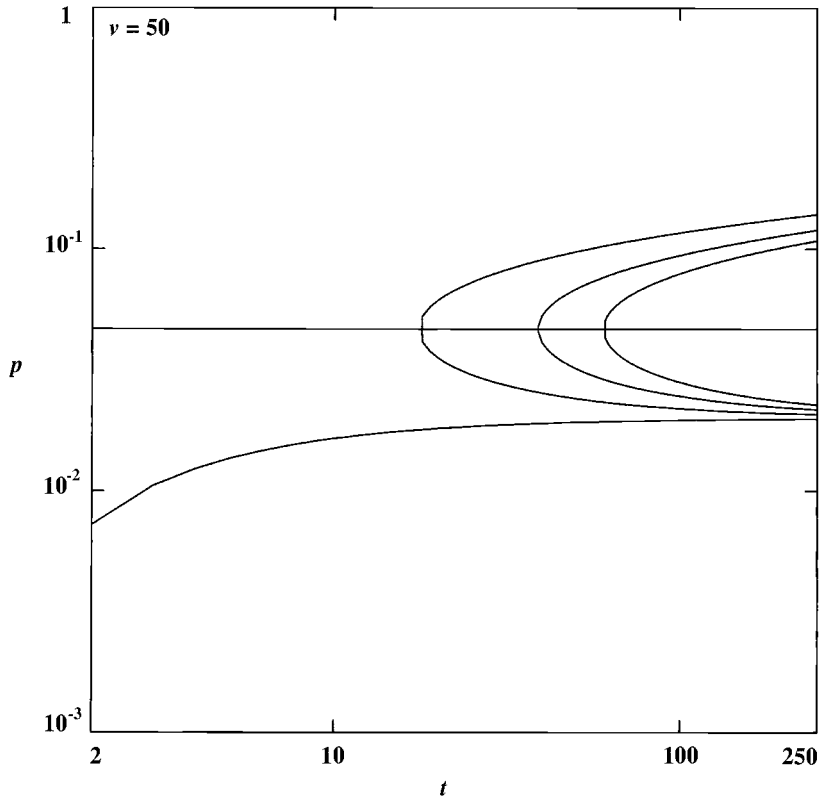


FIG. 2. *Upper limit.*

Simplifying and solving for t gives

$$t \le \frac{(n-1)\ln v - \ln 2 - \Theta(1/v^n)}{\ln\{1 + [p^2 v(v-1) - 1 + p](1-p)^v/2\}}. \tag{84}$$

For $n > 1$ and $1 < a \le pv \le b < \infty$, (84) can be simplified to

$$t \le \frac{(n-1)\ln v - \ln 2}{\ln\{1 + [(pv)^2 - 1]e^{-pv}/2\}}, \tag{85}$$

which is bound (4). (Since this is an upper bound, positive Θ terms were dropped.)

9.2. Small *p*. When p is small, it is useful to simplify (82) by taking the tth root and using $x = e^{\ln x}$:

$$1 + (1-p)^v \left(\frac{p^2 v(v-1)}{2} - \frac{1-p}{2}\right) \le \frac{v^{(n-1)/t}}{2^{1/t}} \left[1 - \Theta\left(\frac{1}{v^n t}\right)\right]. \tag{86}$$

Expanding in power series and multiplying by 2 gives

$$(1-p)^v [p^2 v(v-1) - 1 + p] \le \frac{2[(n-1)\ln v - \ln 2]}{t} \left[1 + \Theta\left(\frac{\ln v}{t}\right)\right]. \tag{87}$$

When $pv > 1$ and p^2v is bounded, bound (87) can be written as

$$[(pv)^2 - 1]e^{-pv} \leq \frac{2[(n-1)\ln v - \ln 2]}{t}\left[1 + \Theta\left(\frac{1}{v}\right) + \Theta(p^2 v) + \Theta\left(\frac{\ln v}{t}\right)\right]. \tag{88}$$

This can be written as

$$p^2v^2 \leq 1 + 2e^{pv}\frac{(n-1)\ln v - \ln 2}{t}\left[1 + \Theta\left(\frac{1}{v}\right) + \Theta\left(\frac{\ln v}{t}\right)\right]. \tag{89}$$

To find the solution near $pv = 1$, let $pv = 1 + y$, giving

$$2y + y^2 \leq 2e[1 + y + \Theta(y^2)]\frac{(n-1)\ln v - \ln 2}{t}\left[1 + \Theta\left(\frac{1}{v}\right) + \Theta\left(\frac{\ln v}{t}\right)\right]. \tag{90}$$

Thus

$$y \leq \frac{e[(n-1)\ln v - \ln 2]}{t}\left[1 + \Theta\left(\frac{1}{v}\right) + \Theta\left(\frac{\ln v}{t}\right)\right]. \tag{91}$$

Drop positive Θ terms to obtain

$$pv \leq 1 + \frac{e[(n-1)\ln v - \ln 2]}{t}, \tag{92}$$

which is equivalent to bound (3).

9.3. Large *p*. In the previous section, we found a solution to bound (82) that has pv near 1. For large pv, $(1-p)^v$ decreases much more rapidly than $(pv)^2$ increases. Bound (88) has the form $(x^2-1)e^{-x} \leq y$ with small y. Define z by $x = -\ln y + 2\ln(-\ln y) + z\ln(-\ln y)/(-\ln y)$. Plugging into $(x^2-1)e^{-x} \leq y$ gives

$$\frac{[-\ln y + 2\ln(-\ln y) + z\ln(-\ln y)/(-\ln y)]^2 - 1}{(-\ln y)^{2+z/(-\ln y)}}y = y. \tag{93}$$

Dividing both sides by y and clearing fractions gives

$$\left(1 + \frac{2\ln(-\ln y)}{-\ln y} + \frac{z\ln(-\ln y)}{(-\ln y)^2}\right)^2 - \frac{1}{(-\ln y)^2} = (-\ln y)^{z/(-\ln y)}.$$

Taking logarithms, expanding the logarithms, and retaining the important terms gives

$$\frac{4\ln(-\ln y)}{-\ln y}\left[1 + \Theta\left(\frac{\ln(-\ln y)}{-\ln y}\right)\right] = \frac{z\ln(-\ln y)}{-\ln y}.$$

In the limit, the right side is bigger for $z > 4$ and the left side is bigger for $z \leq 4$. Thus

$$x = -\ln y + 2\ln(-\ln y) + \Theta\left(\frac{\ln(-\ln y)}{-\ln y}\right)$$

is a solution to the bound $(x^2-1)e^{-x} \leq y$. (Also, there is no solution with larger x.)

When t increases more rapidly than $\ln v$, the solution to (88) is

$$pv \geq \ln t + 2\ln\ln t - \ln\ln v - \ln(n-1) - \ln 2 + \Theta\left(\frac{\ln\ln t}{\ln t}\right). \tag{94}$$

This is bound (1).

9.4. Comparison with Simple Backtracking. When t/v is large, the results of the small-p analysis are not very good. A better result can be obtained by observing that the average running time for Probe Order Backtracking is no larger than that of Simple Backtracking. (The proof that the average running time of Clause Order Backtracking is no larger than that of Simple Backtracking [3, Theorem 1] also applies to Probe Order Backtracking.)

We require that A.18 from [18], the bound for Simple Backtracking, be no more than v^n. For this bound, we use $M(v) = v$ and $\delta = 0$ and let $q = -\ln(1-p)$ to obtain

$$v^n \geq 1 + v \exp\left[2(\ln 2)v + \ln 2 - \frac{\ln 2}{q} \ln\left(1 + \frac{qt}{\ln 2}\right) + t \ln\left(1 - \frac{\ln 2}{\ln 2 + qt}\right)\right]. \tag{95}$$

This can be written as

$$qv - \frac{1}{2}\ln(qv) \leq \frac{1}{2}\ln\left(\frac{t}{v}\right) + \frac{1 - \ln\ln 2}{2} + \frac{q}{2\ln 2}[(n-1)\ln v - \ln 2] - \Theta\left(\frac{q}{v^n}\right) + \Theta\left(\frac{1}{qt}\right). \tag{96}$$

When $t/v > e$, the solution to bound (96) is

$$\begin{aligned} qv \leq \frac{1}{2}\ln\left(\frac{t}{v}\right) + \frac{1}{2}\ln\ln\left(\frac{t}{v}\right) + \frac{1 - \ln 2 - \ln\ln 2}{2} + \frac{q}{2\ln 2}[(n-1)\ln v - \ln 2] \\ + \Theta\left(\frac{\ln\ln(t/v)}{\ln(t/v)}\right) - \Theta\left(\frac{q}{v^n}\right). \end{aligned} \tag{97}$$

Replacing q with its value in terms of p and solving for p in bound (97) gives

$$\begin{aligned} p \leq \left[\frac{\ln(t/v) + \ln\ln(t/v) + 1 - \ln 2 - \ln\ln 2}{2v}\right] \\ \times\left[1 + \frac{2(n-1)\ln v - (\ln 2)[\ln(t/v) + \ln\ln(t/v) + 3 - \ln 2 - \ln\ln 2]}{4v\ln 2}\right. \\ \left. - \Theta\left(\frac{(\ln v)\ln(t/v)}{v^2}\right) - \Theta\left(\frac{\ln\ln(t/v)}{v\ln(t/v)}\right)\right], \end{aligned} \tag{98}$$

which is bound (2). For large v, this is an improvement over the small-p analysis when $t = \beta v$ and $\beta > 3.22136$. Note that $\beta > e$.

9.5. Comparison with solution boundary. The average number of solutions to a random CNF satisfiability problem is

$$2^v[1 - (1-p)^v]^t \tag{99}$$

[18, equation A.1]. This is v^n when

$$p = \frac{1}{v}\left[-\ln\left(1 - \frac{1 + \Theta(n(\ln v)/t)}{e^{(\ln 2)v/t}}\right)\right][1 - \Theta(p)]. \tag{100}$$

(Note that $n = 0$ corresponds to an average of one solution per problem.) When t/v is large, this can be simplified to

$$\begin{aligned} pv = (\ln t - \ln v - \ln\ln 2)\left[1 - \Theta\left(\frac{\ln t - \ln v}{v}\right)\right] \\ + \Theta\left(\frac{v}{t}\right) + \Theta\left(\frac{n\ln v}{v}\right). \end{aligned} \tag{101}$$

Note that for large t/v, the large-p boundary for Probe Order Backtracking being fast (based on the upper bound analysis) is only slightly above the boundary for the number of solutions per problem being above 1. That is, the leading terms in (94) are bigger than those in (101) by only the amount $\ln v$. When t/v is not large, the relative distance between the two curves increases.

9.6. Intersection with Franco's analysis. Franco gives an algorithm [8] which makes selective use of resolution. This algorithm has the fastest proven average time for small t as long as p is not too large. Combining Franco's algorithm with Iwama's algorithm [9] gives an algorithm that is fast for all p when

$$t \le O(n^{1/3}(v/\ln v)^{1/6}). \tag{102}$$

The number of nodes for Franco's algorithm is no more than

$$3 + v + e^{-te^{-2p(1+p)v} + (\ln 2)[8(pt)^3 v + 1]} \tag{103}$$

[8, pp. 1123–1124]. To find the intersection of this bound with the bound for Probe Order Backtracking, it is useful to write (94) as

$$pv \ge \ln t + 2\ln\ln t - \ln\ln v - \Theta(\ln(n-1)) = \ln\left[\frac{t(\ln t)^2}{\Theta(n-1)\ln v}\right].$$

When pv is equal to this bound,

$$e^{-2p(1+p)v} = e^{-(2+2p)pv} = \left[\frac{(n-1)^{\Theta(1)}\ln v}{t(\ln t)^2}\right]^{2+\Theta(p)}.$$

When v and t are polynomially related ($\ln t = \Theta(\ln v)$), we may write this as

$$e^{-2p(1+p)v} = \frac{1}{t^{2+\Theta(p)}}\left[\frac{(n-1)^{\Theta(1)}}{\Theta(1)\ln t}\right]^{2+\Theta(p)} = \frac{1}{t^{2+\Theta(1)}}\left(\frac{n-1}{\ln t}\right)^{\Theta(1)}.$$

Plugging these results into (103) gives a number of nodes (at the intersection of the curves for the two algorithms) of

$$\begin{aligned} &3 + v + \exp\Bigg\{-\left(\frac{n-1}{t\ln t}\right)^{\Theta(1)} \\ &\quad + \frac{(8\ln 2)t^3(\ln t)^3}{v^2}\left[1 + \frac{2\ln\ln t}{\ln t} - \frac{\ln\ln v}{\ln t} - \Theta\left(\frac{\ln(n-1)}{\ln t}\right)\right]^3 + \ln 2\Bigg\}. \end{aligned} \tag{104}$$

If we set this equal to v^n and take logarithms, we have

$$\begin{aligned} &n\ln v - \Theta\left(\frac{1}{v^{n-1}}\right) \\ &= -\left(\frac{n-1}{t\ln t}\right)^{\Theta(1)} + \frac{(8\ln 2)t^3(\ln t)^3}{v^2}\left[1 + \frac{2\ln\ln t}{\ln t} - \frac{\ln\ln v}{\ln t} - \Theta\left(\frac{\ln(n-1)}{\ln t}\right)\right]^3 + \ln 2, \\ &\frac{8(\ln 2)t^3(\ln t)^3}{v^2}\left[1 + \frac{2\ln\ln t}{\ln t} - \frac{\ln\ln v}{\ln t} - \Theta\left(\frac{\ln(n-1)}{\ln t}\right)\right]^3 = n\ln v - \Theta(1), \end{aligned}$$

$$t^3(\ln t)^3 = \frac{nv^2 \ln v}{8 \ln 2} \left[1 - \Theta\left(\frac{1}{\ln v}\right)\right] \left[1 + \frac{2 \ln \ln t}{\ln t} - \frac{\ln \ln v}{\ln t} - \Theta\left(\frac{\ln(n-1)}{\ln t}\right)\right]^{-3}.$$

We may write

$$\begin{aligned} t \ln t &= \frac{(n \ln v)^{1/3} v^{2/3}}{2(\ln 2)^{1/3}} \left[1 - \Theta\left(\frac{1}{\ln v}\right)\right] \left[1 + \frac{2 \ln \ln t}{\ln t} - \frac{\ln \ln v}{\ln t} - \Theta\left(\frac{1}{\ln t}\right)\right]^{-1} \\ &= \frac{(n \ln v)^{1/3} v^{2/3}}{2(\ln 2)^{1/3}} \left[1 - \Theta\left(\frac{1}{\ln v}\right)\right] \left[1 - \frac{2 \ln \ln t}{\ln t} + \frac{\ln \ln v}{\ln t} + \Theta\left(\frac{1}{\ln t}\right)\right] \\ &= \frac{(n \ln v)^{1/3} v^{2/3}}{2(\ln 2)^{1/3}} \left[1 - \frac{2 \ln \ln t}{\ln t} + \frac{\ln \ln v}{\ln t} + \Theta\left(\frac{1}{\ln t}\right) - \Theta\left(\frac{1}{\ln v}\right)\right]. \end{aligned}$$

Define

$$y = \frac{(n \ln v)^{1/3} v^{2/3}}{2(\ln 2)^{1/3}} \left[1 - \frac{2 \ln \ln t}{\ln t} + \frac{\ln \ln v}{\ln t} + \Theta\left(\frac{1}{\ln t}\right) - \Theta\left(\frac{1}{\ln v}\right)\right]$$

and solve $y = t \ln t$ for t. Consider the test solution

$$t = \frac{y}{\ln y} \left[1 + \frac{\ln \ln y}{\ln y} + a \left(\frac{\ln \ln y}{\ln y}\right)^2\right].$$

Plugging this in gives

$$\begin{aligned} y &= \frac{y}{\ln y} \left[1 + \frac{\ln \ln y}{\ln y} + a \left(\frac{\ln \ln y}{\ln y}\right)^2\right] \ln \left\{\frac{y}{\ln y} \left[1 + \frac{\ln \ln y}{\ln y} + a \left(\frac{\ln \ln y}{\ln y}\right)^2\right]\right\} \\ &= y \left[1 + \frac{\ln \ln y}{\ln y} + a \left(\frac{\ln \ln y}{\ln y}\right)^2\right] \\ &\quad \times \left[1 - \frac{\ln \ln y}{\ln y} + \frac{\ln \ln y}{(\ln y)^2} + \Theta\left(\frac{(a - 1/2)(\ln \ln y)^2}{(\ln y)^3}\right)\right] \\ &= y \left[1 + (a-1) \left(\frac{\ln \ln y}{\ln y}\right)^2 + \Theta\left(\frac{\ln \ln y}{(\ln y)^2}\right)\right]. \end{aligned}$$

In the limit, the left side is larger for $a < 1$ and the right side is larger for $a > 1$, so a solution is

$$t = \frac{y}{\ln y} \left[1 + \frac{\ln \ln y}{\ln y} + \Theta\left(\frac{(\ln \ln y)^2}{(\ln y)^2}\right)\right].$$

We have

$$\ln y = \frac{2 \ln v}{3} + \frac{\ln \ln v}{3} + \Theta(\ln n) = \frac{2 \ln v}{3} \left[1 + \frac{\ln \ln v}{2 \ln v} + \Theta\left(\frac{\ln n}{\ln v}\right)\right]$$

and

$$\ln \ln y = \ln \ln v + \Theta(1),$$

so

$$t = \frac{3(n\ln v)^{1/3}v^{2/3}}{4(\ln 2)^{1/3}\ln v}\,\frac{1-(2\ln\ln t)/\ln t+(\ln\ln v)/\ln t+\Theta(1/\ln t)-\Theta(1/\ln v)}{1+(\ln\ln v)/(2\ln v)+\Theta((\ln n)/\ln v)}$$
$$\times\left[1+\frac{3\ln\ln v+\Theta(1)}{2\ln v+\Theta(\ln\ln v)}+\Theta\left(\frac{(\ln\ln v)^2}{(\ln v)^2}\right)\right]$$
$$= \frac{3(n\ln v)^{1/3}v^{2/3}}{4(\ln 2)^{1/3}\ln v}\left[1-\frac{2\ln\ln t}{\ln t}+\frac{\ln\ln v}{2\ln t}+\Theta\left(\frac{1}{\ln t}\right)-\Theta\left(\frac{1+\ln n}{\ln v}\right)\right]$$
$$\times\left[1+\frac{3\ln\ln v}{2\ln v}+\Theta\left(\frac{1}{\ln v}\right)\right]$$
$$= \frac{3(n\ln v)^{1/3}v^{2/3}}{4(\ln 2)^{1/3}\ln v}\left[1-\frac{2\ln\ln t}{\ln t}+\frac{\ln\ln v}{2\ln t}+\frac{3\ln\ln v}{2\ln v}+\Theta\left(\frac{1}{\ln t}\right)-\Theta\left(\frac{1+\ln n}{\ln v}\right)\right].$$

Since t is approximately $v^{2/3}$,

$$\frac{-2\ln\ln t}{\ln t}+\frac{\ln\ln v}{2\ln t}+\frac{3\ln\ln v}{2\ln v}$$

is approximately $-(3/4)(\ln\ln v)/\ln v$. Thus bound (104) is no more than v^n when

$$t = \frac{3n^{1/3}v^{2/3}}{4(\ln 2)^{1/3}(\ln v)^{2/3}}\left[1-\Theta\left(\frac{\ln\ln v}{\ln v}\right)\right]. \tag{105}$$

Note added in proof. The idea of selecting from all positive clauses has been used with good effect on quasigroup problems; see p. 546 of [H. Zhang, M. P. Bonacina, and J. Hsiang, "PSATO: A distributed propositional prover and its application to quasigroup problems," *J. Symbolic Comput.*, 21 (1996), pp. 543–560].

REFERENCES

[1] C. A. Brown and P. W. Purdom, *An average time analysis of backtracking*, SIAM J. Comput., 10 (1981), pp. 583–593.

[2] K. M. Bugrara and C. A. Brown, *The average case analysis of some satisfiability model problems*, Inform. Sci., 40 (1986), pp. 21–38.

[3] K. M. Bugrara and P. W. Purdom, *Average time analysis of clause order backtracking*, SIAM J. Comput., 23 (1993), pp. 303–317.

[4] K. M. Bugrara, Y. F. Pan, and P. W. Purdom, *Exponential average time for the pure literal rule*, SIAM J. Comput., 18 (1988), pp. 409–418.

[5] M. Buro and H. K. Büning, *Report on a SAT competition*, Bull. European Assoc. Theoret. Comput. Sci., 49 (1993), pp. 143–151.

[6] J. V. Franco, *On the probabilistic performance of algorithms for the satisfiability problem*, Inform. Process. Lett., 18 (1986), pp 103–106.

[7] J. V. Franco, *On the occurrence of null clauses in random instances of satisfiability*, Discrete Appl. Math., 41 (1993), pp. 203–210.

[8] J. V. Franco, *Elimination of infrequent variables improves average case performance of satisfiability algorithms*, SIAM J. Comput., 20 (1991), pp. 1119–1127.

[9] K. Iwama, *CNF satisfiability test by counting and polynomial average time*, SIAM J. Comput., 18 (1989), pp. 385–391.

[10] K. J. Lieberherr and E. Specker, *Complexity of partial satisfaction*, J. Assoc. Comput. Mach., 28 (1981), pp. 411–421.

[11] H. M. Méjean, H. Morel, G. Reynaud, *A variational method for analysing unit clause search*, SIAM J. Comput., 24 (1995), pp. 621–649.

[12] A. Newell and H. A. Simon, *GPS, a program that simulates human thought*, in Computers and Thought, E. A. Feigenbaum and J. Feldman, eds., McGraw–Hill, New York, 1963, pp. 279–296.

[13] P. W. Purdom, *Search rearrangement backtracking and polynomial average time*, Artif. Intell., 21 (1983), pp. 117–133.
[14] P. W. Purdom, *A survey of average time analyses of satisfiability algorithms*, J. Inform. Process., 13 (1990), pp. 449–455; an earlier version appeared as *Random satisfiability problems*, in Proc. International Workshop on Discrete Algorithms and Complexity, Institute of Electronics, Information and Communication Engineers, Tokyo, 1989, pp. 253–259.
[15] P. W. Purdom, *Average time for the full pure literal rule*, Inform. Sci., 78 (1994), pp 269–291.
[16] P. W. Purdom and C. A. Brown, *An analysis of backtracking with search rearrangement*, SIAM J. Comput., 12 (1983), pp. 717–733.
[17] P. W. Purdom and C. A. Brown, *The pure literal rule and polynomial average time*, SIAM J. Comput., 14 (1985), pp. 943–953.
[18] P. W. Purdom and C. A. Brown, *Polynomial-average-time satisfiability problems*, Inform. Sci., 41 (1987), pp. 23–42.
[19] P. W. Purdom and G. N. Haven, *Backtracking and probing*, Technical Report 387, Department of Computer Science, Indiana University, Bloomington, IN, 1993.
[20] R. Sosič and J. Gu, *Fast search algorithms for the n-queens problem*, IEEE Trans. Systems Man Cybernet., 21 (1991), pp. 1572–1576.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 484–538, April 1997

009

AMBIVALENT DATA STRUCTURES FOR DYNAMIC 2-EDGE-CONNECTIVITY AND k SMALLEST SPANNING TREES*

GREG N. FREDERICKSON†

Abstract. Ambivalent data structures are presented for several problems on undirected graphs. These data structures are used in finding the k smallest spanning trees of a weighted undirected graph in $O(m \log \beta(m,n) + \min\{k^{3/2}, km^{1/2}\})$ time, where m is the number of edges and n the number of vertices in the graph. The techniques are extended to find the k smallest spanning trees in an embedded planar graph in $O(n + k(\log n)^3)$ time. Ambivalent data structures are also used to dynamically maintain 2-edge-connectivity information. Edges and vertices can be inserted or deleted in $O(m^{1/2})$ time, and a query as to whether two vertices are in the same 2-edge-connected component can be answered in $O(\log n)$ time, where m and n are understood to be the current number of edges and vertices, respectively.

Key words. analysis of algorithms, data structures, embedded planar graph, fully persistent data structures, k smallest spanning trees, minimum spanning tree, on-line updating, topology tree, 2-edge-connectivity

AMS subject classification. 68Q

PII. S0097539792226825

1. Introduction.

Efficient handling of on-line requests requires that data be stored flexibly. At each location in a data structure, it can be advantageous to keep track of a small number of alternatives, only one of which can in fact be valid. An example of such alternatives might be whether a path between vertices x and y in a spanning tree of a graph goes through a vertex w or through a vertex w'. We say that a data structure possesses *ambivalence* if at each of many locations in the structure it keeps track of several alternatives, even when a global examination of the data structure would identify for each location the alternative (or valence) that is in fact valid. (A more formal definition of this property is given at the beginning of section 7.) The structure necessarily organizes the data in such a way that the correct alternative is known for some crucial case. We apply this technique in the design of data structures for several graph problems related to connectivity. Our data structures are ambivalent with regard to the structure of a spanning tree as that spanning tree is being updated, and they yield algorithms faster than any previously known.

Our first problem is that of finding the k smallest spanning trees of a weighted undirected graph. Using data structures that are both ambivalent and fully persistent [DSST], we give an algorithm that uses $O(m \log \beta(m,n) + \min\{k^{3/2}, km^{1/2}\})$ time, where m is the number of edges and n is the number of vertices. Here $\beta(\cdot,\cdot)$ is a very slowly growing function, as defined by Fredman and Tarjan [FT], and the first term in our running time represents the best known time to find a minimum spanning tree [GGST]. Where appropriate, we shall substitute this time when quoting previous results. The amount of space used by our algorithm is $O(m + \min\{k^{3/2}, km^{1/2}\})$. For

* Received by the editors February 11, 1992; accepted for publication (in revised form) June 1, 1995. A preliminary version of this paper appeared in *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 632–641. This research was supported in part by National Science Foundation grants CCR-9001241 and CCR-9322501 and Office of Naval Research contract N00014-86-K-0689.

http://www.siam.org/journals/sicomp/26-2/22682.html

† Department of Computer Sciences, Purdue University, West Lafayette, IN 47907 (gnf@cs.purdue.edu).

the case of a planar graph, we give fully persistent data structures that are then used in an algorithm that takes $O(n + k(\log n)^3)$ time and $O(n + k(\log n)^2)$ space.

Our results compare with previous results on this problem as follows. The problem of enumerating the k smallest combinatorial objects of some particular type has been studied in a number of contexts, including the assignment problem [M], the shortest-path problem [Y], [L1], and the minimum-spanning-tree problem [BH], [CFM], [G], [KIM], [F1], [E]. Early algorithms for finding the k smallest minimum spanning trees can be found in [BH] and [CFM]. Gabow has given an $O(m \log m + km\alpha(m, n))$-time algorithm [G], Katoh, Ibaraki, and Mine have given an $O(m \log \beta(m, n) + km)$-time algorithm [KIM], Frederickson gave an $O(m \log \beta(m, n)+k^2 m^{1/2})$-time algorithm [F1], and Harel claimed an $O(m \log n + kn(\log n)^2)$-time algorithm [Hl2]. Most recently, in [E], Eppstein has given an elegant preprocessing step that allows him to achieve, in conjunction with the algorithm of [KIM], a running time of $O(m \log \beta(m, n) + \min\{k^2, km\})$, using $O(m + k)$ space. Our algorithm matches the running time of Eppstein's for $k \leq (m \log \beta(m, n))^{1/2}$ and is faster for larger values of k. While our algorithm uses more space than Eppstein's for sufficiently large k, the space used by our algorithm is $O(k + m)$ whenever $k \leq m^{2/3}$.

For the case of a planar graph, there are two previous results. In [F1], Frederickson gave an $O(n+k^2(\log n)^3)$-time algorithm, and in [E], Eppstein has given an $O(n+k^2)$-time algorithm. Thus the time for our algorithm is never worse than that in [E], and it is strictly better whenever $k > n^{1/2}$. The space of our algorithm is $O(n)$ whenever $k \leq n/(\log n)^3$.

Our second problem is that of maintaining a data structure for an undirected graph under the operations of inserting and deleting edges and vertices, so as to be able to answer queries about whether two given vertices are in the same 2-edge-connected component. Using ambivalent data structures, we achieve an update time of $O(m^{1/2})$ and a query time of $O(\log n)$, where m and n are understood to be the current number of edges and vertices, respectively.

The question of whether there are data structures with sublinear-time algorithms for maintaining 2-edge-connectivity information under the operations of both insertion and deletion of edges was posed by Westbrook and Tarjan [WT]. Galil and Italiano [GI] describe a data structure that achieves $O(m^{2/3})$ update and query times.

Recently, Eppstein, Galil, Italiano, and Nissenzweig [EGIN] and Eppstein, Galil, and Italiano [EGI] have introduced a sparsification technique that creates a data structure using multiple copies of other data structures. Their technique allows them to use the data structures in this paper and thus replace the m by an n in the times for updating 2-edge-connectivity information and for finding the k smallest spanning trees in a general graph.

Our approach is based on variants of the topology tree and the 2-dimensional topology tree structures presented in [F1]. To make the approach work, we present a different, and in some sense simpler, multilevel partition of the vertices, on which the topology tree and 2-dimensional topology tree are based. A version of our variant of the topology tree that is designed for rooted trees [F3] is appropriate for implementing dynamic trees. In addition to ambivalence, we introduce other novel ideas in our solutions. These include an encoding scheme for vertex names, with the encoded names changing as the topology of a spanning tree changes, and also a partition of the spanning tree into paths based on the multilevel partition.

Our paper is organized as follows. In section 2, we describe the new multilevel partition, and in section 3, we describe the data structures, including fully persistent

data structures, used for updating spanning trees. In section 4, we characterize the adjacency of clusters in embedded planar graphs. In section 5, we describe the data structures, including fully persistent data structures, that are use for updating spanning trees in embedded planar graphs. In section 6, we describe the basic algorithm for finding the k smallest spanning trees, omitting the description of the key data structure. In section 7, we define ambivalence formally and then describe this key data structure for general graphs. In section 8, we describe this key data structure for planar graphs. In section 9, we give a data structure to maintain 2-edge-connectivity in general graphs.

2. Clustering vertices in spanning trees. In this section, we define basic data structures similar to but simpler than those in [F1]. The main contribution of the section is a new way to partition the vertices based on the topology of a spanning tree. We first describe a graph transformation that we use throughout. We then define vertex clusters and our new partition and discuss how the clusters change when an edge is inserted or deleted. We then show that the partition can be applied recursively for only $\Theta(\log n)$ levels.

Throughout this paper, we shall wish to deal with graphs that have maximum vertex degree 3. We first describe how to transform our graph into a graph in which every vertex has degree no greater than three. A well-known transformation in graph theory [Hy, p. 132] is used. By ∞ we designate a sufficiently large value, say equal to the largest value that can be represented in a single word of memory. For each vertex v of degree $d > 3$ and neighbors $w_0, w_1, \ldots, w_{d-1}$, replace v with new vertices $v_0, v_1, \ldots, v_{d-1}$. Add edges $\{(v_i, v_{i+1}) | i = 0, \ldots, d-2\}$, each of cost $-\infty$, and edge (v_{d-1}, v_0) of cost ∞, and replace edges $\{(w_i, v) | i = 0, 1, \ldots, d-1\}$ with $\{(w_i, v_i) | i = 0, \ldots, d-1\}$, of corresponding costs. Note that a minimum spanning tree for the transformed graph will be a minimum spanning tree for the original graph with every edge of cost $-\infty$ added. (The value $-\infty$ is used to ensure that these edges are not swapped out when identifying a best swap in section 6. For the purpose of identifying the cost of the minimum spanning tree in the original graph, treat the $-\infty$ as 0.)

We next define some terms that serve as the foundation for data structures from [F1] that we wish to use. Let $G = (V, E)$ be a connected undirected graph with maximum vertex degree at most 3, and let T be a subgraph of G that is a tree. A *vertex cluster* with respect to T is a set of vertices such that the subgraph of T induced on the cluster is connected. A *boundary vertex* of a cluster is a vertex that is adjacent in T to some vertex not in the cluster. The *tree degree* of a vertex cluster is the number of tree edges with precisely one endpoint in the cluster. Two disjoint vertex clusters are *adjacent* if there is a tree edge that contains one endpoint in each of the clusters. We illustrate the above definitions using Fig. 1, in which a spanning tree is shown with bold edges. The set of vertices $\{6, 7, 13\}$ is not a cluster, since the subset of edges of T incident on it is just $\{(6, 7)\}$ so that the subgraph of T induced on it is not connected. The set of vertices $\{4, 10\}$ is a cluster since the subset of edges of T incident on it is $\{(4, 10)\}$, which connects the vertices. The tree degree of cluster $\{4, 10\}$ is 4. Clusters $\{8, 9, 10, 11\}$ and $\{12, 13, 14\}$ are adjacent because there is a tree edge $(11, 12)$.

We define a partition of a set of vertices so that the resulting vertex clusters possess certain nice properties. Let z be a positive integer. A *restricted partition of order* z with respect to T is a partition of V such that:

1. Each set in the partition is a vertex cluster of tree degree at most 3.
2. Each cluster of tree degree 3 is of cardinality 1.

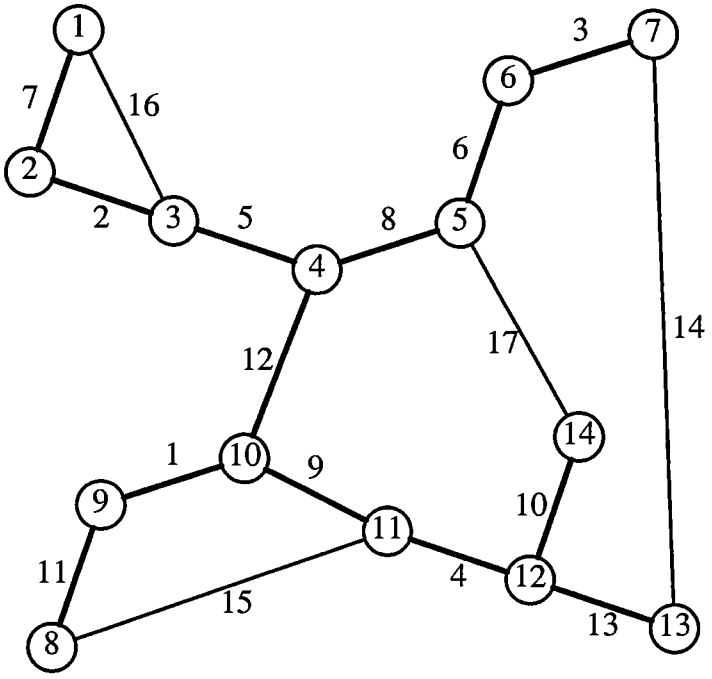


FIG. 1. *A weighted undirected graph with its minimum spanning tree in bold.*

3. Each cluster of tree degree less than 3 is of cardinality at most z.
4. No two adjacent clusters can be combined and still satisfy the above.

As an example, consider the spanning tree from the graph in Fig. 1. A restricted partition of order 2 is shown for this tree in Fig. 2. Note that vertices 10 and 11 cannot be clustered together due to the constraint on the cardinality of a cluster of tree degree 3. Also note that vertex 2 could have been clustered with vertex 1 rather than with vertex 3. In general, there are many different restricted partitions for a given tree and parameter z.

It is not hard to show that the number of clusters in a restricted partition of order z is $\Theta(m/z)$. We do this after the proof of the upcoming Lemma 2.2.

Given a tree described by adjacency lists, a restricted partition can be found as follows. Root the tree at a vertex of tree degree 1. Call the procedure *cluster* with the root as argument. Procedure *cluster*(v), defined below, does the following. It finds all clusters of the subtree rooted at v and outputs all except the partial cluster containing v, which it leaves as $C(v)$. It also finds $tdeg(v)$, the tree degree of $C(v)$, and $size(v)$, the number of vertices in $C(v)$. Upon the return of the call to *cluster*(*root*), print out the set $C(root)$ as the final cluster. Function *cluster*(v) is defined as follows. It initializes $C(v)$ to $\{v\}$, $size(v)$ to 1, and $tdeg(v)$ to the tree degree of v. Then for each child w of v, it calls *cluster*(w) and then does the following. If $tdeg(v) + tdeg(w) - 2 \leq 2$ and $size(v) + size(w) \leq z$, then it resets $C(v)$ to be $C(v) \cup C(w)$, $tdeg(v)$ to be $tdeg(v) + tdeg(w) - 2$, and $size(v)$ to be $size(v) + size(w)$. Otherwise, $C(w)$ is output as a cluster. This completes the description of how each child is handled, and with it the description of procedure *cluster*. Note that the tree degree of any resulting partial cluster will be correctly computed since the tree degree of two unioned partial clusters will be 2 less than the sum of their tree degrees.

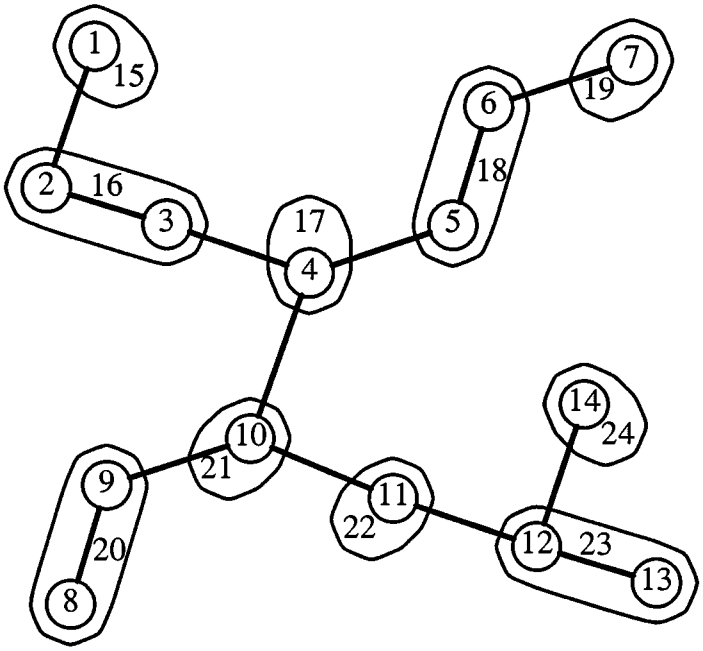


FIG. 2. *A restricted partition of the vertices of the spanning tree in Fig.* 1.

Furthermore, the resulting tree degree will always be less than 3.

The above procedure takes $O(n)$ time for a tree of n vertices. The procedure can be modified in a straightforward fashion to identify the boundary vertices for each cluster and for each vertex identify whether it is a boundary vertex and, if so, for which cluster. Using this representation, the neighboring clusters of any given cluster can be identified in constant time.

An operation that changes the structure of T may force a change in the clusters of a restricted partition. Consider the operation of removing an edge from T, which leaves two trees, T_1 and T_2. Given a restricted partition of order z with respect to T and given an edge in T that is deleted, we discuss how to efficiently generate restricted partitions with respect to the resulting trees T_1 and T_2. Trees T_1 and T_2 inherit the clusters of T, with the following exceptions. Either the edge to be removed has both endpoints contained in one cluster C, or it has one endpoint in a cluster C' and the other in a cluster C''. If the edge to be removed has both endpoints contained in one cluster C, then split C into two clusters, calling them C' and C'', so that the edge has one endpoint in C' and the other in C''. Deleting the edge now causes the tree degree of C' and C'' to change. We need to check if either cluster needs to be combined with other clusters. We discuss how to handle C', with handling C'' being completely analogous. Initialize vertex cluster A to be C'. Repeat the following until no change occurs to A on an iteration. If A has tree degree 1, then check to see if the combined size of it and its neighboring cluster is no greater than z and, if so, combine the neighboring cluster into A. If A has tree degree 2, check to see if there is a neighboring cluster of tree degree no greater than 2 such that the combined size of A and that neighboring cluster is no greater than z and, if so, combine that neighboring cluster into A. This completes the description of how to handle C'.

Next consider the inverse operation of combining two vertex disjoint trees T_1 and

T_2 into one tree T by adding an edge with one endpoint in each of the trees. (Here we assume that the edge was already in the graph, but just not in T.) Given restricted partitions of order z with respect to T_1 and T_2 and given an edge to be inserted that links the trees, we discuss how to efficiently generate a restricted partition with respect to the resulting tree T. Tree T will inherit the clusters of the trees T_1 and T_2, with the following exceptions. If the addition of the edge causes the tree degree of a cluster containing more than 1 vertex to increase from 2 to 3, we must do the following. Consider the three tree edges with exactly one endpoint in the cluster, and let w, w', and w'' denote these endpoints, which are boundary vertices. (Note that two or three of w, w', and w'' may be identical if two or all three of the edges share an endpoint.) Identify the common vertex x on paths in the tree between w and w', between w' and w'', and between w'' and w. Split the cluster by making the vertex x into a cluster by itself and taking the remaining parts of the cluster as clusters. For each cluster so formed, check if another cluster that is adjacent to it is of tree degree at most 2 and if these two clusters together have at most z vertices. If so, combine these clusters. This completes the description of how to handle a cluster when its tree degree increases to 3. Note that these operations can be performed in time proportional to the size of the cluster. If the addition of the edge causes the tree degree of a cluster to increase from 0 to 1 or from 1 to 2, we must do the following. Check to see if the cluster can be combined with the cluster newly adjacent to it. If so, combine these clusters. This completes the description of how to handle a cluster when its tree degree increases from 0 to 1 or from 1 to 2.

LEMMA 2.1. *At most a constant number of clusters are deleted or created when an edge insertion or deletion is performed with respect to restricted partitions of z. The time to perform the changes is $O(z)$.*

Proof. We first observe that only a constant number of clusters are deleted or created when an edge insertion is performed. In the case that a cluster has its tree degree increase from 2 to 3, each part of the split cluster except the part containing only vertex x has tree degree 2. It can be combined with a neighbor B_1 of tree degree 2 but not also combined with the other neighbor B_2 of B_1 since otherwise B_1 and B_2 would have already been combined. In the case of adding an edge that causes the tree degree of a cluster to increase from 0 to 1 or from 1 to 2, a similar argument ensures that only the two clusters containing the endpoints of the edge need to be considered for merging.

We next consider how many clusters are deleted or created when an edge deletion is performed. We perform a case analysis below for how C' can be combined with other clusters. The analysis for C'' is essentially the same. We consider size constraints only when they definitely rule out a case. Subcases are meant to inherit the conditions satisfied by parent cases. *Case* 1: (C' is of tree degree 1.) Then C' can be combined with a neighboring cluster B_1 of tree degree 1, 2, or 3. Let the resulting cluster be B_2. *Case* 1.1: (B_1 is of tree degree 1.) Then B_2 has tree degree 0 and we are done. *Case* 1.2: (B_1 is of tree degree 2.) Then B_2 is of tree degree 1. Let the other neighbor of B_1 be B_3. *Case* 1.2.1: (B_3 is of tree degree 1 or 2.) Then B_2 cannot be combined with B_3 since otherwise B_1 and B_3 would have already been combined. *Case* 1.2.2: (B_3 is of tree degree 3.) Then B_2 and B_3 may be combined, and the resulting cluster B_4 is of tree degree 2. Let the neighbors of B_3, besides B_1, be B_5 and B_6. *Case* 1.2.2.1: (B_5 is of tree degree 1.) Then B_5 cannot be combined with B_4 since it would already have been combined with B_3. *Case* 1.2.2.2: (B_5 is of tree degree 3.) Then B_5 cannot be combined with B_4 because the resulting cluster would have tree degree 3. *Case*

1.2.2.3: (B_5 is of tree degree 2.) Then B_5 can be combined with B_4, but the resulting cluster B_7 cannot be combined with the other neighbor B_8 of B_5 since otherwise B_5 would already have been combined with B_8. A similar discussion holds for cluster B_6, allowing that B_7 could be used in place of B_4 in the arguments. *Case* 1.3: (B_1 is of tree degree 3.) The argument mimics that in Case 1.2.2 and its subcases, but with B_1 in the role of B_3 and C' in the role of B_2. *Case* 2: (C' is of tree degree 2.) The argument mimics those in Cases 1.2.2.1, 1.2.2.2, and 1.2.2.3, with C' in the role of B_4 and with B_5 and B_6 being the neighbors of C'. This completes an analysis of the cases. It is clear that the above operation will examine just a constant number of clusters.

We next consider the time to perform an edge insertion. In the case that a cluster has its tree degree increase from 2 to 3, identifying w, w', and w'' takes constant time, and finding x takes time proportional to the size of the cluster, which is $O(z)$. Splitting the cluster will take $O(z)$ time. Checking neighboring clusters and merging as necessary will take constant time. In the case that a cluster has its tree degree increase to 1 or 2, the time to perform checking and merging is constant.

We next consider the time to perform an edge deletion. If the edge to be deleted has both endpoints contained in one cluster, then the time to split the list of vertices of the cluster into two lists is $O(z)$. All checking and combining will then take constant time. □

We next define our restricted multilevel partition. A *restricted multilevel partition* is a set of partitions of V that satisfy the following:

1. For each level $l = 0, 1, \ldots, q$, the vertex clusters at level l form a partition of V.
2. The clusters at level 0 form a restricted partition of order z.
3. The clusters at any level $l > 0$ constitute a restricted partition of order 2 with respect to the tree resulting from viewing each cluster at level $l-1$ as a vertex.
4. There is precisely one vertex cluster at level q, which contains all vertices.

As an example, consider the spanning tree from the graph in Fig. 1. A restricted multilevel partition for this tree is shown in Fig. 3. Here we assume that $z = 1$ so that each basic vertex cluster contains precisely one vertex. The second level corresponds to the restricted partition in Fig. 2. There are six levels in this multilevel partition.

We note that the restricted multilevel partition is somewhat similar to a structure that may be inferred from applying the rake-and-compress paradigm to a rooted binary tree [MR], [CV], [ADKP].

A vertex cluster at level 0 of a restricted multilevel partition is called a *basic vertex cluster*. Since any basic vertex cluster of tree degree 3 consists of a single vertex and any cluster resulting from the union of two clusters will have tree degree at most 2, any nonbasic cluster of tree degree 3 will also consist of a single vertex. All three of its incident edges will be tree edges. Note that there are no nontree edges with an endpoint in a cluster of tree degree 3.

We next show that the restricted multilevel partition has other nice properties. Consider any level $l > 0$ of a restricted multilevel partition. Call any vertex cluster of level $l-1$ *matched* if it is unioned with another another cluster to give a vertex cluster at level l. Call all other vertex clusters at level $l-1$ *unmatched*. Since a cluster of tree degree 1 can be matched with a cluster of tree degree 1, 2, or 3, the only reason that a cluster of tree degree 1 is not matched is that its adjacent cluster is already matched with some other cluster. Since a cluster of tree degree 2 can be matched

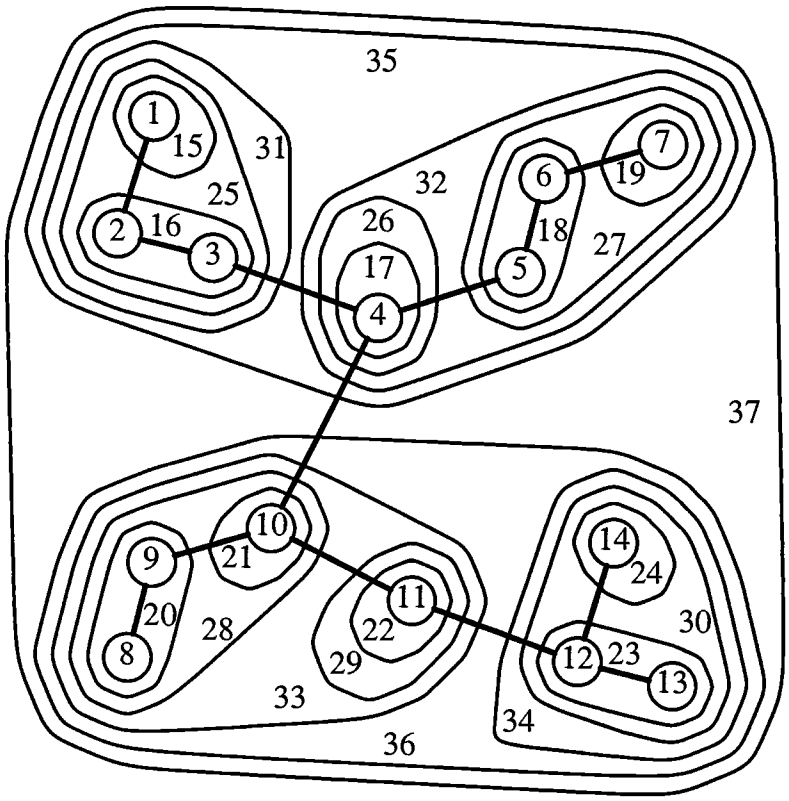


FIG. 3. *A restricted multilevel partition of the vertices of the spanning tree in Fig.* 1.

with an adjacent cluster of tree degree 2, the only reason that a cluster of tree degree 2 is not matched with an adjacent cluster of tree degree 2 is that that adjacent cluster is already matched with another cluster.

LEMMA 2.2. *For any level* $l > 0$ *of a restricted multilevel partition, the number of matched vertex clusters at level* $l-1$ *is at least* $1/3$ *of the total number of vertex clusters at level* $l-1$.

Proof. Consider any level $l > 0$ of a restricted multilevel partition. Contract the graph by contracting all tree edges both of whose endpoints are in the same cluster at level $l-1$. Let each vertex resulting from a matched cluster by such a contraction be called a *matched* vertex. Let the tree degree of a resulting vertex be the tree degree of the corresponding cluster. If all vertices are matched, then clearly the lemma follows. Otherwise, root the tree at an unmatched vertex of largest tree degree. We shall give 6 credits to each pair of vertices that have been matched together, and we will show that these credits can be spread around so that, in the end, each vertex will receive at least 1 credit. The lemma will then follow.

Consider any pair of vertices that have been matched together, and assume that the pair has been allocated 6 credits. Since neither is the root, and since the number of unmatched neighbors of the pair is at most 2, the higher of the two has a parent, which may be unmatched, and the second neighbor (if any) is a child, which may be unmatched. If the higher vertex of the pair has an unmatched parent, let the matched pair send 3 credits to this parent. If there is a second neighbor, let

the matched pair send 1 credit to this child. Let each matched vertex in the pair retain at least 1 credit. Call any unmatched vertex of tree degree 2 that is the root or has an unmatched parent of tree degree 3 *sheltered*. From the properties of unmatched clusters discussed prior to the statement of this lemma, every unmatched vertex of tree degree 1 and every unsheltered unmatched vertex of tree degree 2 must have a neighbor that is matched. In particular, the parent of every such vertex will be matched so that the vertex will receive from its parent 1 credit, which it will retain.

The above credit-sharing rule guarantees that if an unmatched vertex has not received a credit from either its children or its parent, then it must be either a vertex of tree degree 3 or a sheltered vertex. We add the following two rules to handle these cases. For any sheltered nonroot vertex, if it receives 3 credits from its child, it should pass 2 credits to its parent and retain the other 1. For any unmatched nonroot vertex of tree degree 3, if it receives at least 2 credits from each of its two children, it should pass 3 credits to its parent and retain the other at least 1 credit. By a simple induction, it can be shown that every sheltered nonroot vertex will receive 3 credits from its child and retain 1 of them, and every unmatched nonroot vertex of tree degree 3 will receive at least 4 credits from its children and retain at least 1 of them. It follows that at the end of all credit passing, each vertex will retain (at least) 1 credit. A root of tree degree 3 will receive at least 2 credits from each of its 3 children, and a root of tree degree 2 will receive 3 credits from each of its 2 children. Since an unmatched vertex of tree degree 1 must have a matched neighbor, an unmatched tree root will receive 3 credits from its child. □

We now show that the number of clusters in a restricted partition of order z is $\Theta(m/z)$. Consider a restricted multilevel partition of order z. Level 0 of the multilevel partition is a restricted partition of order z. Consider the vertex clusters at level 0 that are matched together to form vertex clusters at level 1. The total number of vertices in a pair of matched vertex clusters must be greater than z since otherwise the pair could have been merged in the partition at level 0. Thus there are fewer than m/z such pairs of matched vertex clusters. By Lemma 2.2, the total number of these pairs is at least 1/3 of the total number of vertex clusters at level 0. Thus there are fewer than $3m/z$ clusters at level 0, i.e., in the restricted partition.

There is an infinite family of examples that match the bound of Lemma 2.2 in the following way. Let n_{l-1} be the number of clusters at level $l-1$. For $n_{l-1} \geq 13$ and $n_{l-1}+5$ a multiple of 6, the number of matched vertex clusters is at least $(n_{l-1}+5)/3$. It has not escaped our attention that we could match more vertex clusters if rule 3 in the restricted multilevel partition allowed unions whose resulting vertex cluster had tree degree 3. However, it appears difficult and inefficient to update the corresponding topology tree structures when changes occur. (Indeed, the difficulty encountered when trying to make things work with tree degree 3 rather than tree degree 2 in rule 3 was the reason that the multilevel partition was defined as it was in [F1].)

THEOREM 2.3. *The number of levels in a restricted multilevel partition is* $\Theta(\log n)$.

Proof. The number of vertex clusters at level 0 is $O(n)$. By Lemma 2.2, for any level $l > 0$, the number of matched vertex clusters at level $l-1$ is at least 1/3 of the total number of vertex clusters at level $l-1$. Since each pair of matched vertex clusters at level $l-1$ that are paired together are replaced by the union at level l, the number of vertex clusters at level l is at most 5/6 the number of vertex clusters at level $l-1$. Since the number of vertex clusters at level l is at least 1/2 the number of vertex clusters at level $l-1$, it follows that the number of levels is $\Theta(\log n)$. □

3. Data structures for maintaining spanning trees.

In this section, we define basic data structures similar to but simpler than those in [F1]. Following [F1], we define a "topology tree" based on the partition and show how to update the topology tree when an edge not in the spanning tree is swapped for an edge in the spanning tree. We then define a "2-dimensional topology tree," again following [F1]. We show how to make 2-dimensional topology trees fully persistent. Finally, we show how to update a 2-dimensional topology tree when edges and vertices are inserted into or deleted from the underlying graph.

As in [F1],we define data structures that describe our partitions. Given a restricted multilevel partition for a spanning tree T, a *topology tree* for T is a tree in which each nonleaf node has at most two children and all leaves are at the same depth, such that:

1. A node at level l in the topology tree represents a vertex cluster at level l in the restricted multilevel partition.
2. A node at level $l > 0$ has children that represent the vertex clusters at level $l-1$ whose union is the vertex cluster it represents.

We label a node in the topology tree by the indexed name of the vertex cluster that the node represents.

A topology tree for the restricted multilevel partition of Fig. 3 is given in Fig. 4. Each node in the topology tree is labeled with the index of the vertex cluster that it represents. The children and parent pointers are represented by the straight, bold edges. The adjacency between clusters is represented by the thin, curved edges.

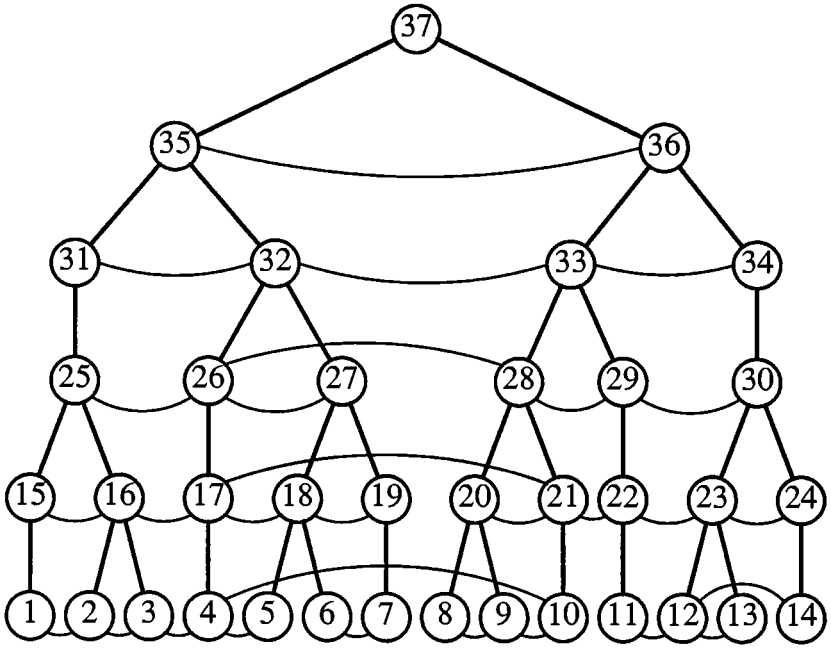


FIG. 4. *The topology tree corresponding to the restricted multilevel partition in Fig.* 3.

A topology tree based on a restricted multilevel partition has the same nice properties as a topology tree based on the multilevel partition of [F1]. In particular, it can be modified efficiently to show the result of inserting or deleting an edge or performing a swap. A *swap* (e, f) in a spanning tree T replaces a tree edge e by a nontree edge

f, yielding another spanning tree. We next discuss how to modify the topology tree when a swap is performed. First, reform the basic clusters to reflect the insertion and deletion of edges, as discussed previously. The number of basic vertex clusters that are changed, created, or deleted will be at most some constant. Then starting with the lists of basic vertex clusters that are changed, created, or deleted, we rebuild portions of the topology tree from the bottom up.

We describe this rebuilding carefully. We shall use three lists of nodes that need to be examined: Let L_D be a list of nodes that represent clusters that should be deleted, L_C a list of nodes that have parents and that represent clusters that have been changed, and L_A a list of nodes that represent clusters that have no parent, either because they are new or because their parent is on a list of nodes to be deleted. Note that we view a cluster as changing not only if its set of vertices changes, but also if its set of tree edges to other clusters changes. (Here we assume that a tree edge to another cluster has not changed if it is the same tree edge as before the swap, i.e., that it has the same vertices as endpoints in the underlying graph even though one of the endpoints may be in a different cluster than before.)

We initialize L_D, L_C, and L_A as follows. Insert into L_D each node representing a basic cluster that has been split or combined to form new basic clusters, and insert into L_A each node representing a new basic cluster. Let the adjacency information of neighboring nodes refer to the nodes inserted into L_A rather than L_D, and let the nodes on L_D retain their parent information but have their adjacency information set to null. For each node x representing a basic cluster whose set of vertices has not changed but whose set of edges incident on it has changed, update its adjacency information and insert it into L_C.

While we have not reached the root of the resulting topology tree, we will recluster the clusters represented by the nodes on these lists and reset the lists to contain the nodes representing the corresponding parents. We perform this activity in such a way that at any point in time, the total size of these lists does not exceed a fixed constant. At each level in the topology tree, we do the following. Assume that the adjacency information of the nodes on L_D, L_C, and L_A reflects the result of the swap but that the adjacency information of the parents of these nodes does not yet. We shall create lists L'_D, L'_C, and L'_A to hold the corresponding nodes at the next higher level. Initialize lists L'_D, L'_C, and L'_A to be empty.

First, we handle list L_D. For every node x in L_D, remove x from L_D (and return it to the available storage pool), remove x as a child from its parent y (if any), and if y then has no children, insert y into L'_D. If y had another child x' and this child is not already on L_C or L_D, then insert x' into L_C. Next, we scan L_C for nodes that have siblings. Let x be such a node on L_C, with parent y and sibling x'. If the cluster corresponding to node y remains a valid cluster, then remove x from L_C (and x' too if it resides on L_C) and insert y into L'_C. By a cluster remaining valid, we mean that there is actually an edge between the cluster representing x and the one representing x', and the tree degree of the cluster for y does not now exceed 2. If the cluster corresponding to y does not remain a valid cluster, then remove x and x' as children of y, remove x from L_C (and also x' if it resides on it), insert x' and x into L_A, and insert y into L'_D.

Finally, we handle nodes on L_A and the remaining nodes on L_C. Let x be such a node. Remove x from the appropriate list. We consider three cases. First, suppose that node x represents a cluster of tree degree 3. If x is adjacent to a node x' representing a cluster of tree degree 1, then do the following. Cluster x and x' together,

removing one or both from lists L_A and L_C as necessary. If neither had a parent, then create a parent and insert it into L'_A. If both had a parent, then use the parent y of x, inserting y into L'_C and inserting the old parent y' of x' into L'_D. If just one of x and x' had a parent, use it and insert it into L'_C. This completes the description of how to handle x when it is adjacent to such a node x'. If x is not adjacent to such a node x', then do the following. Cluster node x by itself. If x has a parent, then insert this parent onto L'_C. If x has no parent, then create a parent and insert it onto L'_A. This completes the description of the first case.

The second and third cases are similar. Second, suppose node x represents a cluster of tree degree 2. If x is adjacent to a node x' representing a cluster of tree degree 1 or 2 such that or x' has no sibling (either because it is an only child or because it has no parent), then cluster x and x' together as discussed in the case above in which x represents a cluster of tree degree 3. If node x is not adjacent to such a node x', then cluster x with itself and handle the parent (or lack of one) as discussed in the case above for tree degree 3. Third, suppose node x represents a cluster of tree degree 1. If x is adjacent to a node x' such that x' has no sibling (either because it is an only child or because it has no parent), then cluster x and x' together as discussed in the case above for tree degree 3. Note that if x' represents a cluster of tree degree 1, the resulting cluster must have tree degree 0 so that the corresponding node will be the root of the topology tree. If node x is not adjacent to such a node x', then cluster x with itself and handle the parent (or lack of one) as discussed in the case above for tree degree 3. Fourth, if x represents a node of tree degree 0, then it is the root of a topology tree, and a pointer to it should be saved. This completes the discussion of how to handle a node x on L_A or on L_C.

When all nodes have been removed from L_D, L_C, and L_A, determine and adjust the adjacency information for all nodes on L'_D, L'_C, and L'_A and then reset L_D to be L'_D, L_C to be L'_C, and L_A to be L'_A. This completes the description of how to handle the lists L_D, L_C, and L_A. When L_C and L_A together contain only one node, then this node corresponds to the root of the topology tree. Any additional nodes in L_D should be removed. (These nodes and their ancestors should be returned to the available storage pool.) This completes the description of the algorithm to handle a swap, which we call algorithm *basic_swap*.

LEMMA 3.1. *Consider a topology tree based on a restricted multilevel partition. Algorithm basic_swap performs a swap in* $O(z + \log n)$ *time.*

Proof. We first consider the correctness of *basic_swap*. The algorithm processes the topology tree level by level in rounds. Before each round, we claim that the only nodes that need to be considered are on L_D, L_A, and L_C and that the adjacency information is valid for the nodes on the level being processed. Furthermore, we claim that for any level after the first that is being processed, clusters for all nodes on that level except for the ones on L_D correspond to a valid restricted partition of the clusters on the next lower level. We prove the above claims by induction on the number of rounds, with the final result being that the structure created is a valid topology tree.

For the basis, note that before the first round, the lists L_D, L_A, and L_C contain precisely those nodes whose corresponding clusters are undergoing some change. Also, the adjacency information for the nodes corresponding to basic clusters has been changed to accurately reflect the changes in basic clusters caused by replacing one edge by another. For the induction step, we consider the point in the execution of the algorithm just before the rth round, $r > 1$. We assume that the claims are true at the

point in the execution of the algorithm just before the $(r-1)$st round. The algorithm deletes each node x from L_D and adjusts the information at its parent correctly. The algorithm also examines nodes on L_C that have siblings and splits any node from its sibling if they do not now form a valid cluster. Finally, any nodes that are only children and can be clustered together are clustered together. Any resulting node is put on L'_A if it is a new node and on L'_C if it represents a changed cluster. Thus at this point, the nodes at the next level minus those nodes on L'_D represent the clusters of a valid restricted partition. Nodes on L'_D are those nodes at the next level that should be deleted. Thus the only nodes that need to be considered at the next level are on L'_D, L'_A, and L'_C. Just before the end of the round, the adjacency information is adjusted for all nodes on L'_D, L'_A, and L'_C, and L'_D, L'_A, and L'_C are reassigned to be L_D, L_A, and L_C, respectively. Thus at the beginning of the rth round, all three claims hold.

We next consider the time complexity of *basic_swap*. Since a constant number of basic vertex clusters are changed, deleted, and created and each basic cluster that is altered in some way can be handled in $O(z)$ time, the total cost of handling the basic vertex clusters is $O(z)$. The time to perform this algorithm exclusive of changing the basic clusters will be proportional to the number of nodes in the topology tree that are deleted, examined, and created. We analyze how many nodes can be on the lists L_A, L_C, and L_D at the beginning of any round, and we show this to be bounded by a constant. For a *link*, there are no more than some constant number of nodes on L_A and L_C before the first round. A simple case analysis indicates that this number of nodes can be at most eight before the first round begins. This is realized when two vertices in clusters previously of tree degree 2 are linked together. Each of these may be split into at most four clusters. For a *cut*, there are also no more than some constant number of nodes on L_A and L_C before the first round.

Our analysis depends on the way in which the nodes on L_A and L_C relate to each other within the structure of the tree induced on the level corresponding to a round. If the operation is *link*, then the nodes on lists L_A and L_C form a subtree of the resulting tree induced on that level. If the operation is *cut*, then the nodes on lists L_A and L_C form a subtree of each of the two resulting trees induced on that level.

We first analyze the *link* operation. Before the first round, the subtree induced on nodes in L_A and L_C consists of no more than eight nodes and seven edges. Let a *border edge* be an edge in the tree but not in the subtree that is incident to a node of the subtree. We claim that at any point while the topology tree is being rebuilt, there are at most two border edges on each side of the linking edge in the tree. This is true initially since each of the linked clusters previously had tree degree at most 2. During a round, the subtree can be extended to include a larger portion of the tree in two ways. First, a cluster represented by node y can be recognized to be invalid, where the constituent clusters are represented by x and x', x is on L_C, and x' is not on any list. If the cluster is invalid because the tree degree would now be 3, then either of two cases holds. If x now has tree degree 3 and x' has tree degree 2, then including x' in the subtree does not increase the number of border edges. If x now has tree degree 2 and x' has tree degree 3, then including x' in the subtree increases by one the number of border edges. But in this case x must have previously had tree degree 1. It follows that the linking edge is incident on a vertex in the cluster represented by x and that this node was the only one in the subtree on its side of the linking edge. Thus there was just one border edge (the one incident on x) on its side of the linking edge, and we are now increasing the number to two.

If the cluster is invalid because the two constituent clusters are not adjacent, then before the previous round, the cluster corresponding to node x included a cluster adjacent to a constituent cluster of x'. If x' has tree degree 2, then including x' does not increase the number of border edges. If x' has tree degree 3, then including x' increases the number of border edges by one. But in this case, the old version of x previously had tree degree 1, and its cluster contained one endpoint of the linking edge and thus was the only node in the subtree on one side of the linking edge. So prior to including x' in the subtree, the portion of the subtree on that side of the linking edge had just one border edge. Thus that number is now increased to two.

The second way to extend the subtree is to union a cluster represented by a node x on L_A with a cluster represented by a node x' not on L_A or L_C. If x' has tree degree 2, then including x' does not increase the number of border edges. If x' has tree degree 3, then x must have tree degree 1. This means that the subtree consists only of x. On all subsequent rounds, the subtree will consist of only a single node, and the number of edges incident on it will be at most two. This completes our case analysis. In all cases, the number of border edges will never exceed four.

We next consider how many nodes will be in the subtree at the end of a round, when there are s nodes in the subtree at the beginning of the round. The only clusters that can be determined to be invalid must contain the endpoints of the original *link* operation. Thus at most two clusters will be determined to be invalid, yielding two more nodes for the subtree. Other clusters may enter the subtree by unioning a cluster in the subtree with one not in the subtree, but this results in no net gain in the number of nodes. In the worst case, each node in the subtree that has a border edge incident on it will not have its cluster unioned with one whose node is in the subtree. Of the remaining $s + 2 - 4$ nodes, Lemma 2.2 establishes that at most $5/6$ of that number, or $5(s-2)/6$, will remain. An upper bound on the largest value possible for s is thus determined by the inequality $s \leq 5(s-2)/6 + 4$. This implies that the total size of L_A and L_C will never exceed 14. (A more careful analysis of the proof of Lemma 2.2, considering the constant additive term, will reduce the bound substantially.)

To bound the number of nodes in L_D, consider the original two topology trees representing the two trees linked together. For each round, we consider the minimal subtrees of the induced trees that connect nodes on L_D. These subtrees are of the same form as the subtree induced on nodes of L_A and L_C. By similar arguments, it can be shown that the subtrees, and hence the length of L_D, are bounded by a constant. Thus the total size of lists L_A, L_C, and L_D is bounded by a constant on any round. Since the amount of work per list entry is constant, and by Theorem 2.3 the number of rounds is $O(\log n)$, the total time for a *link* is $O(\log n)$.

The analysis for a *cut* is similar. In each of the two subtrees, there will be at most two border edges. The subtrees can be extended in a fashion similar to that for a *link*. The analysis is essentially the same, yielding equivalent bounds for the lengths of lists L_A, L_C, and L_D. Thus the total time for a *cut* is also $O(\log n)$. □

We trace through an example to illustrate algorithm *basic_swap*. Consider the graph in Fig. 2 and the spanning tree and multilevel partition in Fig. 3. We assume that each node in Fig. 2 represents a basic cluster. Suppose that the edge between V_7 and V_{13} is swapped in to replace the edge between V_4 and V_{10}. For simplicity, we shall assume that no basic cluster is changed as far as the set of vertices it contains. (This would be true if clusters V_3, V_4, V_9, and V_{11} have size exactly z, so V_4 or V_{10} cannot be combined with their neighboring clusters, and the size of V_7 plus the size of V_{13} is greater than z and the size of V_6 plus the size of V_7 is greater than z, so that

neither V_7 nor V_{13} can be combined with a neighboring cluster.) For convenience, we use as the name of the node the index of its cluster. Initially, L_D and L_A are empty, and L_C comprises $4, 10, 7$, and 13. When L_C is examined on the first phase, node 13 and its sibling 12 no longer form a valid cluster. Thus node 12 is placed on L_C and node 23 is placed on L'_D. We then proceed to handling the remainder of L_C and L_A. Node 4 cannot be clustered with a neighbor, so its parent 17 is placed on L'_C. Node 10 is clustered with 11, and the resulting parent 21 is placed on L'_C, while the previous parent 22 of 11 is placed on L'_D. Nodes 7 and 13 get clustered, and the parent 19 is placed on L'_C. Node 12 can be clustered with 14, putting resulting parent 24 on L'_C. At this point, L_D, L_A, and L_C are empty, and the next phase begins with setting L_D, L_A, and L_C to L'_D, L'_A, and L'_C.

When L_D is handled on the second phase, nodes 22 and 23 are deleted. Since node 22 is an only child, its parent 29 is put on L'_D. Node 24, the sibling of 23, is already on L_C. Node 28, the parent of 21, remains a valid cluster and is put on L'_C. Similarly, node 27, the parent of 19, remains a valid cluster and is put on L'_C. We then proceed to handling the remainder of L_C and L_A. Node 17 cannot be clustered with a neighbor, so its parent 26 is placed on L'_C. Node 24 cannot be clustered with a neighbor, so its parent 30 is placed on L'_C.

At this point, the second phase ends, and the third phase begins with recopying the lists. Node 29 is removed from L_D, and its sibling 28 is already on L_C. Node 32, the parent of nodes 26 and 27, remains a valid cluster and is put on L'_C. We then proceed to handling the remainder of L_C and L_A. Node 28 is clustered with 30, and the resulting parent 33 is placed on L'_C, while the previous parent 34 of 30 is placed on L'_D. At this point, the third phase ends, and the fourth phase begins. Node 34 is removed from L_D, and its sibling 33 is already on L_C. Node 32 on L_C has a sibling, and its parent cluster 35 is still valid, so that 35 is placed on L'_C. Node 33 cannot be clustered with a neighbor, so its parent 36 is placed on L'_C. At this point, the fourth phase ends, and the fifth phase begins. List L_D is empty. Nodes 35 and 36 each have siblings (each other), and their parent 37 represents a valid cluster, so 37 is placed on L'_C. In the sixth phase, 37 is identified as having tree degree 0, so that it is the root of the new topology tree. The algorithm then terminates. The resulting topology tree, with the old vertices crossed out and the old edges dashed, is shown in Fig. 5. To minimize the clutter in Fig. 5, the edges representing adjacency between clusters are not shown.

A 2-*dimensional topology tree* for a given topology tree is a tree in which for every ordered pair of nodes labeled V_j and V_r at the same level in the topology tree, there is a node labeled $V_j \times V_r$, and there is a child of node $V_j \times V_r$, labeled $V_{j'} \times V_{r'}$, for each pair consisting of a child $V_{j'}$ of V_j and a child $V_{r'}$ of V_r in the topology tree.

A portion of the 2-dimensional topology tree for the topology tree of Fig. 4 is given in Fig. 6. Specifically, all nodes and children of nodes on the path from the root to the leaf 5×14 are shown. For any node that is shown in the figure but whose children are not shown, there is an edge for each child coming from the bottom of that node.

For the size bound z on the number of vertices in a basic cluster, we choose $z = \lceil m^{1/2} \rceil$. When one modifies a topology tree as the result of performing a swap, the 2-dimensional topology tree must be modified. This modification is essentially the same as that discussed in [F1].

LEMMA 3.2. *Consider a 2-dimensional topology tree for a topology tree that is based on a restricted multilevel partition. The space is $O(m)$, the time to set up the*

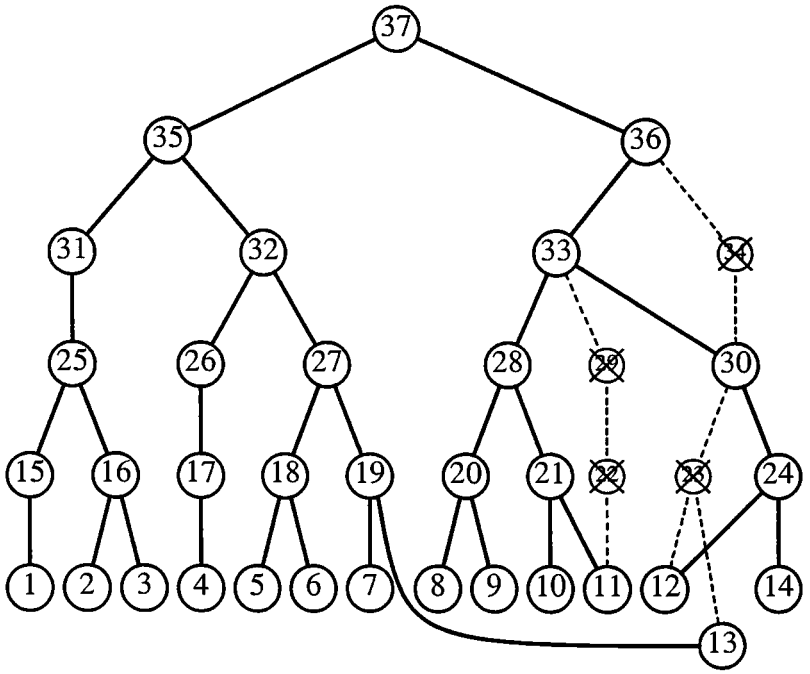


FIG. 5. *The changes to the topology tree in Fig.* 4 *after edge* $(7, 13)$ *replaces edge* $(4, 10)$ *in the spanning tree.*

2-*dimensional topology tree given its topology tree is* $O(m)$*, and the time required to modify the* 2-*dimensional topology tree to show the result of performing a swap is* $O(m^{1/2})$.

Proof. The space and times are derived in a fashion similar to that in [F1]. □

In section 7, we use a version of 2-dimensional topology trees that is fully persistent [DSST]. Thus when we perform a swap, we want to retain the old version of the 2-dimensional topology tree before the swap and also create a version of the 2-dimensional topology tree after the swap. This can be done efficiently by creating new nodes when they are needed and by allowing a new node in the 2-dimensional topology tree to have one or more children in the old 2-dimensional topology tree. Thus certain subtrees of the old 2-dimensional topology tree are shared by both old and new trees by virtue of having two pointers pointing at the root of any such subtree. (After creating a number of versions via swaps, a node could have any number of pointers less than or equal to the number of versions pointing to it.) To make such a scheme work, the version of 2-dimensional topology tree that is made fully persistent is not allowed to have any parent pointers in the nodes.

The discussion in [F1] regarding modifying a 2-dimensional topology tree implicitly supposes that there are parent pointers in the tree. We thus discuss how to perform a swap when the 2-dimensional topology tree does not have these pointers. We maintain a copy of the topology tree for each version of the 2-dimensional topology tree, and since the nodes of these topology trees will not be shared, we allow the nodes of the topology tree to have parent pointers. When we want to perform a swap (e, f) in tree T with topology tree $T1D(T)$ and a pointer to the 2-dimensional topology tree $T2D(T)$, we do the following. First, we make a copy of $T1D(T)$ and run *basic_swap*

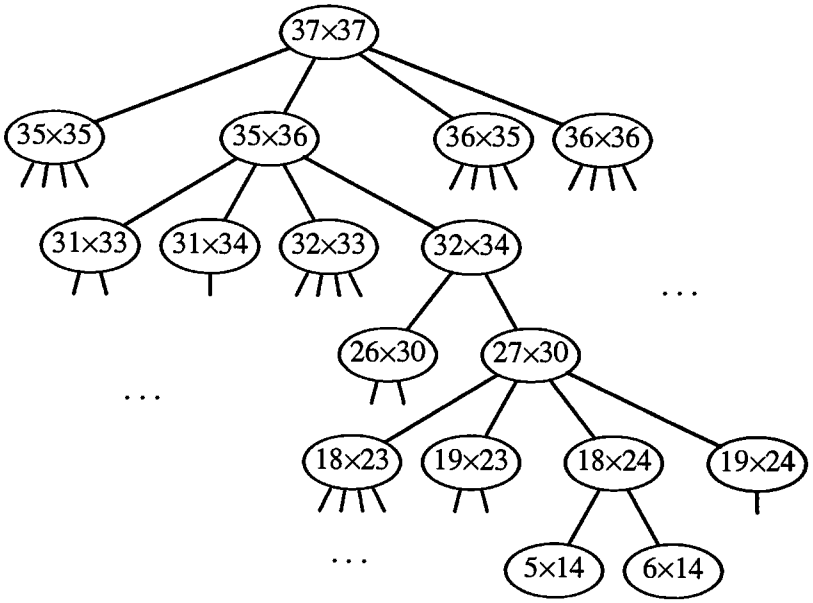


FIG. 6. *A portion of the* 2*-dimensional topology tree for the topology tree shown in Fig.* 4.

on this copy, generating $T1D(T')$. We actually use a modified version of *basic_swap* that marks each node in $T1D(T)$ that has been deleted or whose corresponding node in $T1D(T')$ represents a cluster that has changed. These nodes are all the nodes that have been inserted into L_D, L'_D, or L'_C.

For example, consider the graph and spanning tree in Fig. 1, whose topology tree is shown in Fig. 4, with edge $(7, 13)$ swapped in to replace edge $(4, 10)$, as shown shown in Fig. 5. The vertices in the topology tree in Fig. 4 that should be marked are $4, 7, 10, 13, 17, 19, 21, 22, 23, 24, 26, 27, 28, 29, 30, 32, 33, 34, 35,$ and 36. The subtrees shared by both the old 2-dimensional topology tree, as shown in Fig. 6, and the new 2-dimensional topology tree are those rooted at the nodes $V_{31} \times V_{31}$, $V_{18} \times V_{18}$, $V_{20} \times V_{20}$, $V_{15} \times V_{18}$, $V_{18} \times V_{15}$, $V_{15} \times V_{20}$, $V_{20} \times V_{15}$, $V_{16} \times V_{18}$, $V_{18} \times V_{16}$, $V_{16} \times V_{20}$, $V_{20} \times V_{16}$, $V_{18} \times V_{20}$, and $V_{20} \times V_{18}$ and leaves $V_j \times V_r$ and $V_r \times V_j$, for $j \in \{1, 2, 3, 5, 6, 8, 9\}$ and $r \in \{11, 12, 14\}$ or for $j, r \in \{11, 12, 14\}$.

Having marked those nodes in $T1D(T)$ whose corresponding nodes in the copy are involved in the restructuring that generates $T1D(T')$, we then set temporary parent pointers in a portion of the shared data structure that represents $T2D(T)$. The temporary pointers, as well as the marks in the nodes of $T1D(T)$, will be reset to a null value once $T2D(T')$ has been generated. The temporary pointers will be set for any node $V_j \times V_r$ such that either V_j or V_r or both are marked in $T1D(T)$. The procedure *set_tpp*$(p1, p2)$ will do this, where $p1$ points to a marked node in $T1D(T)$ representing some cluster V_j and $p2$ points to a node in $T2D(T)$ representing an ordered pair of clusters $V_j \times V_r$ or $V_r \times V_j$.

proc *set_tpp*$(p1, p2)$
    if $p1$ is a leaf
    then Save $(p1, p2)$ on a list.
    else

```
if p1 has just one child p1c
then
    if p1c is marked
    then
        for each child p2c of p2 do
            temp_par(p2c) ← p2
            Call set_tpp(p1c, p2c).
        endfor
    endif
else
    if the left child p1L of p1 is marked
    then
        for each child p2L of p2 that corresponds to V_j × V_r
                where V_j is a left child do
            temp_par(p2L) ← p2
            Call set_tpp(p1L, p2L).
        endfor
    endif
    if the right child p1R of p1 is marked
    then
        for each child p2R of p2 that corresponds to V_j × V_r
                where V_j is a right child do
            temp_par(p2R) ← p2
            Call set_tpp(p1R, p2R).
        endfor
    endif
endif
endif
```

Consider the list of pairs $(p1, p2)$ corresponding to certain leaves that is created by procedure *set_tpp*. This list is then used to initialize a simultaneous bottom-up traversal of the portion of $T1D(T)$ whose nodes are marked and the portion of $T2D(T)$ whose nodes have temporary parent pointers. During the traversal, actions are performed in $T2D(T)$ that are analogous to those of *basic_swap* on $T1D(T)$, except that no nodes in $T2D(T)$ are deleted. Instead, new nodes are created and the child pointers of these nodes are set to both new nodes and nodes in $T2D(T)$. The new nodes and their pointers are set up to be consistent with the structure of $T1D(T')$. A pointer to the root of the resulting 2-dimensional topology tree $T2D(T')$ is returned. Let the above approach be called algorithm *persist_swap*.

THEOREM 3.3. *Algorithm persist_swap maintains a fully persistent version of 2-dimensional topology trees, using $O(m + km^{1/2})$ space to store k versions and generating a new version reflecting the result of a swap in $O(m^{1/2})$ time.*

Proof. We first consider the correctness of algorithm *persist_swap*. Algorithm *basic_swap* correctly marks each node in $T1D(T)$ that has been deleted or whose corresponding node in $T1D(T')$ represents a cluster that has changed. Next, a proof by induction can be used to establish that if a node in $T1D(T)$ is marked, then all ancestors of that node are marked. Now any subtree rooted at a node $V_j \times V_r$ in $T2D(T)$ such that V_j and V_r are not marked will remain the same in $T2D(T')$. Thus the only nodes that may get deleted or changed in modifying $T2D(T)$ to get $T2D(T')$

are nodes $V_j \times V_r$ such that at least one of V_j and V_r are marked. A proof by induction establishes that procedure *set_tpp* correctly sets temporary parent pointers for all such nodes. A bottom-up procedure for deleting and changing all such nodes then can then simulate the effect of *basic_swap* in $T2T$ by following the temporary parent pointers.

We next consider the resource usage of *persist_swap*. Given the choice of z, there are $\Theta(m^{1/2})$ basic clusters, and each is of size $O(m^{1/2})$. It follows from Lemma 2.2 that there are $\Theta(m^{1/2})$ vertices in the topology tree $T1D(T)$. Thus a copy can be made in $O(m^{1/2})$ time. By Lemma 3.1 the modified version of *basic_swap* will take $O(m^{1/2})$ time. By Lemma 3.2, the time used in modifying the 2-dimensional topology tree $T2D(T)$ is $O(m^{1/2})$, and this is a bound also on the number of nodes examined. The time of *set_tpp* is proportional to the number of nodes examined, so that this time is also $O(m^{1/2})$. It follows that the total time to perform a swap is $O(m^{1/2})$. Since the number of new nodes is bounded by the time, each of $k-1$ versions after the first will use $O(m^{1/2})$ additional space. □

In section 9, we consider a problem in which the underlying graph can change by inserting or deleting edges or vertices. Inserting edges into or deleting edges from the original graph can be handled similarly to that discussed at the beginning of section 8 of [F1]. We supply some additional explanation since the discussion in [F1] is brief. In particular, we discuss what happens when the insertion of an edge increases the degree of a vertex above 3 or decreases the degree of a vertex that is above 3. In the either case, the transformation that replaces a vertex by a ring of vertices of degree 3 must be modified. For simplicity of discussion, we assume that every vertex of degree at least 2 is converted into a ring of degree-3 vertices. In the case of edge insertion, we define an *inflate* operation as follows. For each endpoint v of the inserted edge, do the following. If the degree of v was previously 1, then treat the existing v as v_0 and treat the endpoint of the new edge as v_1. Generate the topology tree data structures for the single edge. Then insert edges (v_0, v_1) and (v_1, v_0), rebuilding the data structures in a fashion similar to that done in *basic_swap*. If the previous degree d of v was greater than 1, then treat the endpoint of the new edge as v_d. Generate the topology tree data structures for the single edge. If edge (v_0, v_{d-1}) is a tree edge, then identify a nontree edge with which it can swap and perform the swap. Then delete edge (v_0, v_{d-1}) and insert edges (v_{d-1}, v_d) and (v_d, v_0), rebuilding the data structures in a fashion similar to that done in *basic_swap*. Note that the vertices v_0 through v_{d-1} are identified by position rather than labeled as such. Identifying a nontree edge which can participate in a swap can be accomplished by organizing the data structure for maintaining a minimum spanning tree and changing the cost of the edge to be deleted to ∞.

An operation *deflate* can be defined to perform essentially the reverse of *inflate*. If an edge to be deleted is a tree edge, first determine if there is a nontree edge that can be swapped for the edge to be deleted and, if so, then perform the swap.

It is not hard to cast the problems of edge and vertex insertion and deletion as edge insertion or deletion. We allow a vertex to be inserted whenever it is an endpoint of an edge that is being inserted and the other endpoint of the edge is already in the graph. A vertex is deleted whenever it is an endpoint of degree 1 and its incident edge is being deleted. (We thus force our graph to always be connected.)

THEOREM 3.4. *Consider a structure based on a restricted multilevel partition. The time required to insert or delete an edge or vertex is $O(m^{1/2})$, where m is the current number of edges.*

Proof. As in [F1], splitting and merging basic vertex sets will use $O(z)$ time. The

time to modify all affected nodes in the topology and 2-dimensional topology tree will be $O(m/z)$. □

4. Adjacency in embedded planar graphs. For embedded planar graphs, we characterize the adjacency relationships between clusters in the multilevel partition presented in section 3. Nontree edges with precisely one endpoint in any given vertex cluster will be grouped together and ordered according to the embedding. Our work will follow the general idea in [F1] but will elaborate the details with more care than in [F1].

We shall first choose a size for basic vertex clusters and then make a number of simple observations about the consequences of this choice. We then define sets of nontree edges, called "boundary sets," with precisely one endpoint in any given vertex cluster. We show how to generate a boundary set of a cluster that is the union of two other clusters from their boundary sets. The situation is complicated by what we call "separating edges," which are not always easy to identify efficiently. Our approach will first generate "pseudoboundary sets," which can contain the separating edges, and then later at opportune times it will remove the separating edges to give the boundary sets.

First, we choose $z = 1$ in our restricted multilevel partition so that each basic vertex cluster will be a vertex by itself. We carefully examine how to represent the nontree edges. Recall that a cluster of tree degree 3 will consist of a single vertex and will have no nontree edges incident on it. Next, consider a cluster V_j of tree degree 1. All nontree edges with exactly one endpoint in V_j can be ordered in clockwise order around V_j, starting with the first edge in a clockwise direction from the tree edge with one endpoint in V_j. This ordering will be entirely consistent with the embedding. Next, consider a cluster V_j of tree degree 2. There will be a unique path of tree edges between the two boundary vertices of V_j. We partition all nontree edges with precisely one endpoint in V_j into two sets, depending on which "side" of the path an edge is incident on. Each set can be ordered in a natural way corresponding to the embedding and represented by a balanced tree. Call each such ordered set of edges a *boundary set*. It is easy to identify the zero, one, or two boundary sets of a basic cluster in constant time, given a list of edges incident on the single vertex in the cluster, as well as an indication of which edges are tree edges.

Consider the embedded planar graph in Fig. 7. The spanning tree edges are in bold, the nontree edges are dashed, and a multilevel partition is shown by the closed curves. The vertex cluster $\{10, 11\}$ has an associated path from vertex 10 to vertex 11. Cluster $\{10, 11\}$ has edges $(10, 9)$ and $(11, 7)$ in the boundary set to the left of the path and no edges in the boundary set to the right of the path. Cluster $\{4, 5, 6, 7\}$ has an associated path from vertex 4 to vertex 7. It has edge $(5, 3)$ in one boundary set and edge $(7, 11)$ in the other boundary set. Note that edge $(6, 4)$ has both endpoints in this cluster and thus is not in either boundary set.

We next discuss how to determine the boundary sets of the clusters. Clearly, a cluster that has just one cluster as its child has the same boundary sets as its child. Given cluster V_j that is the result of the union of two clusters $V_{j'}$ and $V_{j''}$, we show how to generate the boundary set of V_j from the boundary sets of $V_{j'}$ and $V_{j''}$. We first discuss two simple cases. The first case is that $V_{j'}$ and $V_{j''}$ are both of tree degree 1. Then V_j is the set of all vertices, and all edges in the boundary sets of $V_{j'}$ and $V_{j''}$ will be interior with respect to V_j. Thus there will be no boundary set for V_j. The second case is that $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3. In this case, V_j will be of tree degree 2. Make the boundary set of $V_{j'}$ one of the two boundary sets

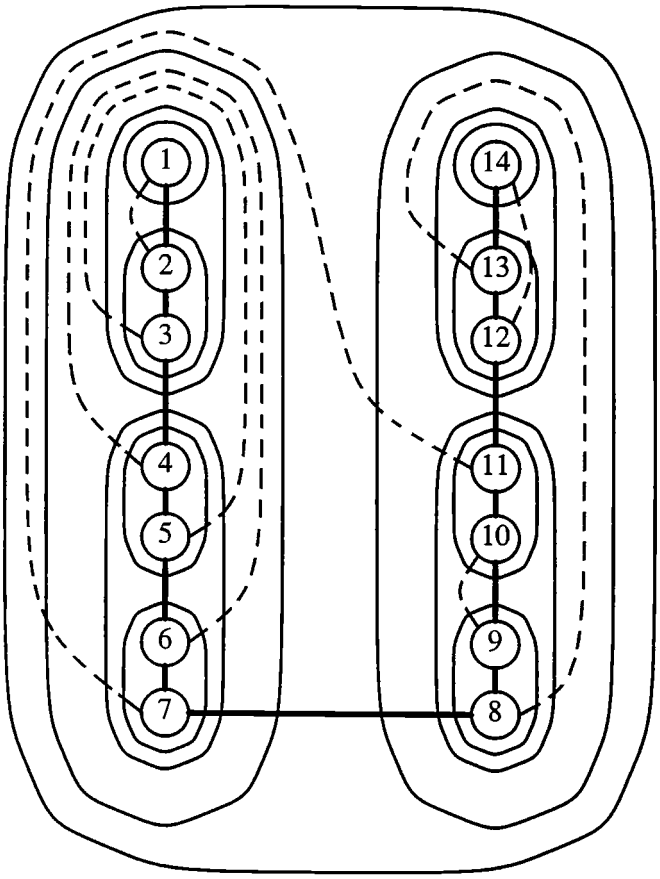


FIG. 7. *An embedded planar graph, its spanning tree, and a multilevel partition.*

of V_j. The other boundary set of V_j is empty.

The third case is that $V_{j'}$ and $V_{j''}$ are both of tree degree 2. Each of $V_{j'}$ and $V_{j''}$ will have two boundary sets, one on each side of the path between the boundary vertices of V_j. Consider a nontree edge with one endpoint in each of $V_{j'}$ and $V_{j''}$. If every such edge is a member of two boundary sets that are on the same side of the path, then things are easy: we shall describe in due course a simultaneous search of two boundary sets to identify the subset of edges contained in both boundary sets. However, suppose that there are nontree edges that are members of the boundary set of $V_{j'}$ on one side of the path and are also members of the boundary set of $V_{j''}$ on the other side of the path. We call any such edge e a *separating edge for* V_j since the cycle induced by e in the tree separates some pair of clusters that are different from V_j. Let the *separating set* of edges for V_j be the separating edges with one endpoint in $V_{j'}$ and the other endpoint in $V_{j''}$. Consider, for example, Fig. 7. The cluster containing vertices 4 and 5 has a separating edge $(4, 5)$. It separates the cluster containing vertices 2 and 3 from the cluster containing vertices 6 and 7. The presence of separating edges complicates matters considerably since there appears to be no efficient way to identify this set by merely examining the boundary sets for $V_{j'}$ and $V_{j''}$. Our solution will be to avoid identifying these edges when initially examining

the union of two clusters of tree degree 2 and to identify only "pseudoboundary sets" at that time.

A *pseudoboundary set* of a cluster V_j of tree degree 2 is defined as follows. If V_j is a basic vertex cluster, then the pseudoboundary sets of V_j are identically the boundary sets of V_j. If V_j has just one child cluster, then the pseudoboundary sets of V_j are identically the pseudoboundary sets of the child cluster. If V_j is the union of two clusters of tree degree 1 and 3, then the pseudoboundary sets of V_j are identically the boundary sets of V_j. Otherwise, V_j is the union of two clusters $V_{j'}$ and $V_{j''}$ of tree degree 2. A simultaneous search of each pseudoboundary set of $V_{j''}$ and the pseudoboundary set of $V_{j'}$ that is on the same side of the path can be performed to identify the subset of edges common to both sets. Split these subsets off from the pseudoboundary sets of $V_{j'}$ and $V_{j''}$ and concatenate the remaining portions to get the two pseudoboundary sets of V_j. It follows that the edges in the pseudoboundary sets of V_j that are not in the boundary sets of V_j are separating edges of either V_j or certain clusters that are descendants of V_j. (In a multilevel partition, one cluster is a descendant of another cluster if the former is contained in the latter.)

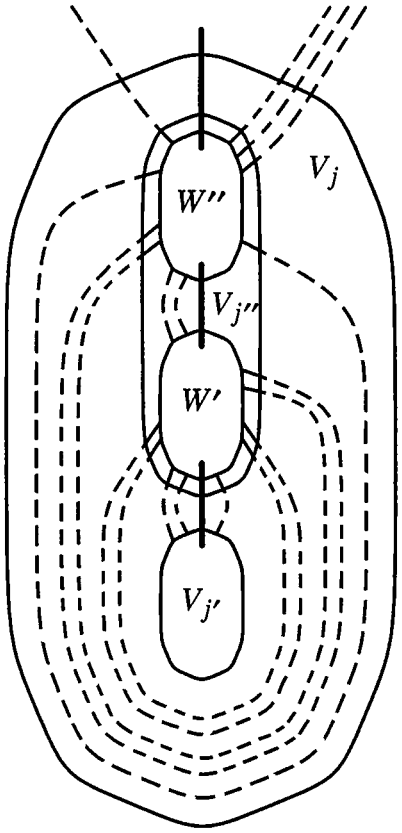


FIG. 8. *An example that illustrates finding sets of separating edges.*

Consider Fig. 8, in which the spanning tree edges are in bold, the nontree edges are dashed, and a portion of a multilevel partition is shown by the closed curves. There are two separating edges for cluster W', one seperating edge for cluster W'', and two separating edges for cluster $V_{j''}$. Looking just at $V_{j''}$, there is no efficient way

to find the separating edges for $V_{j''}$ because they are sandwiched between edges to clusters that are not unioned yet (at this level) to $V_{j''}$. The left pseudoboundary set of W'' contains six edges, and the right contains four edges. The left pseudoboundary set of W' contains six edges, and the right contains five edges. The left pseudoboundary set of $V_{j''}$ contains eight edges, and the right contains nine edges.

The fourth and final case is that $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2. Let the sides of the path between the boundary vertices of $V_{j''}$ be designated as L for left and R for right. We differentiate the directions "down" and "up," with down being closer to $V_{j'}$ and up being farther away from $V_{j'}$. Perform a simultaneous search from either end of the boundary set of $V_{j'}$ with the appropriate pseudoboundary set of $V_{j''}$ to identify the two subsets of edges of set $V_{j'}$ that are in either of the pseudoboundary sets of $V_{j''}$. Split these two subsets off from the boundary set of $V_{j'}$ and from the pseudoboundary sets of $V_{j''}$. If any edges remain in the boundary set of $V_{j'}$, then there are no separating edges for $V_{j''}$. In this case, take the remaining portions of the two pseudoboundary sets of $V_{j''}$ and concatenate them with the remaining portion of the boundary set of $V_{j'}$ to give the boundary set of V_j. If no edges remain in the boundary set of $V_{j'}$, then one can identify all separating edges of V_j (and also of certain of its descendants) that are in the pseudoboundary set of $V_{j''}$. Perform a simultaneous search from the lower end of the remaining portions of the two pseudoboundary sets of $V_{j''}$ to identify all edges that are in the remaining portions of both pseudoboundary sets. Then split this subset off from the remaining pseudoboundary sets of $V_{j''}$. Take the remaining portions of the two pseudoboundary sets of $V_{j''}$ and concatenate them together to give the boundary set of V_j.

We return to our example in Fig. 7. When clusters $\{8, 9\}$ and $\{10, 11\}$ are unioned together, the edge $(9, 10)$ is identified as staying on the same side of the path (from vertex 8 to vertex 11). Thus it is not in the pseudoboundary sets for $\{8, 9, 10, 11\}$. When clusters $\{4, 5\}$ and $\{6, 7\}$ are unioned together, edge $(6, 4)$ is a separating edge, though it would not in general be determined as such at that time. The pseudoboundary sets of cluster $\{4, 5, 6, 7\}$ are $\{(5, 3), (6, 4)\}$ and $\{(4, 6), (7, 11)\}$. When cluster $\{1, 2, 3\}$ of tree degree 1 and cluster $\{4, 5, 6, 7\}$ of tree degree 2 are unioned together, we detect the separating edges as follows. First, cluster $\{1, 2, 3\}$ has a boundary set containing edge $(3, 5)$, which is deleted from that set as well as from a pseudoboundary set for $\{4, 5, 6, 7\}$. The separating edge $(6, 4)$ is uncovered from each pseudoboundary set of $\{4, 5, 6, 7\}$. Note that there are no separating edges to be discovered within either cluster $\{4, 5\}$ or $\{6, 7\}$. The boundary set for cluster $\{1, 2, \ldots, 7\}$ will then just contain edge $(7, 11)$.

Note that we have not yet completed our discussion of how to determine the boundary sets of all clusters since the above only determines pseudoboundary sets for the case of a cluster that is the union of tree degree 2. First, we make some additional remarks about how to represent and manipulate these sets of edges. For each cluster, keep track of the portions of pseudoboundary sets of the children that are not used in building the pseudoboundary set of the cluster. We call the set of such edges on each side to be the *newly interior set* of edges.

We next describe how to take the separating edges determined when clusters $V_{j'}$ of tree degree 1 and $V_{j''}$ of tree degree 2 are unioned together, and we determine the boundary sets and separating sets for the descendant clusters for which only pseudoboundary sets were previously known. Let W be any descendant cluster of $V_{j''}$ such that all ancestors of W that are also descendants of $V_{j''}$ have tree degree 2. Call such a cluster W a *relevant descendant* for $V_{j''}$. We introduce the following notation.

Let $pbs_L(W)$ be the pseudoboundary set of W on the left side of the path, and let $pbs_R(W)$ be the pseudoboundary set of W on the right side of the path. Let $s_T(W)$ be the number of separating edges with both endpoints in W. Let $d_L(W)$ be the number of edges in $pbs_L(W)$ with the other endpoint down from W, and let $u_L(W)$ be the number of edges in $pbs_L(W)$ with the other endpoint up from W. Similarly, let $d_R(W)$ be the number of edges in $pbs_R(W)$ with the other endpoint down from W, and let $u_R(W)$ be the number of edges in $pbs_R(W)$ with the other endpoint up from W.

To determine the boundary sets and separating sets for all relevant descendants of $V_{j''}$, call the recursive procedure *sep_edge* with arguments $V_{j''}$, $s_T(V_{j''})$, $u_L(V_{j''})$, $u_R(V_{j''})$, $d_L(V_{j''})$, and $d_R(V_{j''})$. Procedure *sep_edge*$(W, s_T(W), u_L(W), u_R(W), d_L(W), d_R(W))$ will take a relevant descendant W of tree degree 2 and construct representations of the boundary sets and separating sets of W and all of its relevant descendants. We make the following notational simplifications. The argument (W) will be omitted, using, for example, u_L rather than $u_L(W)$. In the case that W has two children, W' will be down from W''. An argument such as (W'') will be replaced by a double prime, so that u''_L represents $u_L(W'')$ and similarly for (W'). The number of separating edges of W, i.e., separating edges with one endpoint in W' and the other in W'', will be s. The number of such separating edges with the left endpoint higher than the right will be s_L, and the number of such separating edges with the right endpoint higher than the left will be s_R. The number of edges that are in both $pbs_L(W'')$ and $pbs_L(W')$ will be c_L, and the number of edges that are in both $pbs_R(W'')$ and $pbs_R(W')$ will be c_R. Let $bs_L(W)$ and $bs_R(W)$ represent the left and right boundary sets, respectively, of W, and let $ss(W)$ represent the separating set of W.

proc *sep_edge*$(W, s_T, u_L, u_R, d_L, d_R)$
  if the boundary sets of W are not defined
  then
    if W has a single child W'
    then
      Call *sep_edge*$(W', s_T, u_L, u_R, d_L, d_R)$.
      $bs_L(W) \leftarrow bs_L(W')$; $bs_R(W) \leftarrow bs_R(W')$; $ss(W) \leftarrow \emptyset$
    else
      Let W' and W'' be the children of W.
      $u'_L \leftarrow u_L$; $u''_L \leftarrow u_L$; $d'_L \leftarrow d_L$; $d''_L \leftarrow d_L$
      $u'_R \leftarrow u_R$; $u''_R \leftarrow u_R$; $d'_R \leftarrow d_R$; $d''_R \leftarrow d_R$
      if $s_T = 0$
      then
        $bs_L(W) \leftarrow pbs_L(W)$; $bs_R(W) \leftarrow pbs_R(W)$; $ss(W) \leftarrow \emptyset$
        Call *sep_edge*$(W', 0, u'_L, u'_R, d'_L, d'_R)$.
        Call *sep_edge*$(W'', 0, u''_L, u''_R, d''_L, d''_R)$.
      else
        Determine c_L by a simultaneous search from the top of $pbs_L(W')$
          and the bottom of $pbs_L(W'')$.
        Determine c_R similarly.
        if the $(u_L + 1)$st edge down in $pbs_L(W'')$
          is not the same as the $(u_R + 1)$st edge down in $pbs_R(W'')$
        then $s''_T \leftarrow 0$
        else

Determine s''_T by a search down in $pbs_L(W'')$ and $pbs_R(W'')$,
starting at positions specified in the if-expression, respectively.
$d''_L \leftarrow size(pbs_L(W'')) - u_L - s''_T$
$d''_R \leftarrow size(pbs_R(W'')) - u_R - s''_T$
endif
if the $(d_L + 1)$st edge up in $pbs_L(W')$
is not the same as the $(d_R + 1)$st edge up in $pbs_R(W')$
then $s'_T \leftarrow 0$
else
Determine s'_T by a search up in $pbs_L(W')$ and $pbs_R(W')$,
starting at positions specified in the if-expression, respectively.
$u'_L \leftarrow size(pbs_L(W')) - d_L - s'_T$
$u'_R \leftarrow size(pbs_R(W')) - d_R - s'_T$
endif
$s \leftarrow s_T - s'_T - s''_T$
if $(s > 0)$ **and**
[$(s''_T > 0$ **and** the $(u_L + s''_T + 1)$st edge down in $pbs_L(W'')$
is the same as the $(c_R + 1)$st edge down in $pbs_R(W'))$ **or**
$(s'_T > 0$ **and** the $(d_R + s'_T + 1)$st edge up in $pbs_R(W')$
is the same as the $(c_L + 1)$st edge up in $pbs_L(W''))$ **or**
$(s'_T = 0$ **and** $s''_T = 0$ **and** $(u_L + 1)$st edge down in $pbs_L(W'')$
is the same as the $(d_R + s)$th edge up in $pbs_R(W'))$]
then $s_L \leftarrow s$; $s_R \leftarrow 0$
else $s_L \leftarrow 0$; $s_R \leftarrow s$
endif
Let $ss(W)$ be the corresponding set of s edges.
Call $sep\_edge(W', s'_T, u'_L, u'_R, d'_L, d'_R)$.
Call $sep\_edge(W'', s''_T, u''_L, u''_R, d''_L, d''_R)$.
Create $bs_L(W)$
by deleting from $bs_L(W'')$ the bottom c_L edges and the
$(u''_L + 1)$st through $(u''_L + s_L)$th edges from the top,
deleting from $bs_L(W')$ the top c_L edges and the
$(d''_L + 1)$st through $(d''_L + s_R)$th edges from the bottom,
and concatenating what remains of $bs_L(W'')$ and $bs_L(W')$.
Create $bs_R(W)$ in a similar way.
endif
endif
endif

As an example, consider the portion of an embedded planar graph in Fig. 8. The spanning tree edges are in bold, the nontree edges are dashed, and a portion of a multilevel partition is shown by the closed curves. In particular, a cluster V_j is shown that is the union of a cluster $V_{j'}$ of tree degree 1 and a cluster $V_{j''}$ of tree degree 2. As discussed above, the set of separating edges of $V_{j''}$, as well as the set of separating edges of certain descendants of $V_{j''}$, can be determined. There are three edges in the boundary set of $V_{j'}$ and eight and nine edges, respectively, in the left and right pseudoboundary sets of $V_{j''}$. There are two edges in the newly interior set

to the left of the path in Fig. 8 and none in the newly interior set to the right of the path. There are one edge from the left pseudoboundary set and three from the right pseudoboundary set that will be in the boundary set of V_j. For the sake of discussion, let W be $V_{j''}$. Then $s_T = 5$, $u_L = 1$, $u_R = 3$, $d_L = 2$, and $d_R = 1$. On the recursive call *sep_edge*$(V_{j''}, 5, 1, 3, 2, 1)$, it is determined that W has two children W' and W''. It would then be determined that $c_L = 2$, $c_R = 0$, $s''_T = 1$, $d''_L = 2$, $d''_R = 1$, $s'_T = 2$, $u'_L = 2$, $u'_R = 0$, $s = 2$, $s_L = 2$, and $s_R = 0$. By initialization, $u''_L = 1$, $d'_L = 2$, $u''_R = 3$, and $d'_R = 1$. Note that $size(pbs_L(W'')) = 6$, $size(pbs_L(W')) = 6$, $size(pbs_R(W'')) = 4$, and $size(pbs_R(W')) = 5$. We have not shown the internal structure of W' and W'', and so we will not discuss what happens during the recursive calls *sep_edge*$(W', 2, 2, 0, 2, 1)$ and *sep_edge*$(W'', 1, 1, 3, 2, 0)$. Upon the returns, $bs_L(W)$ would be created, containing three edges, and $bs_R(W)$ would be created, containing four edges.

We designate as algorithm *build_sets* the algorithm presented above to compute the boundary sets, newly interior sets, and separating sets for the clusters in a multilevel partition of a planar graph.

LEMMA 4.1. *Algorithm build_sets correctly computes the boundary sets, newly interior sets, and separating set for the clusters in a multilevel partition of a planar graph.*

Proof. We shall prove by induction that for any level $l \leq q$, *build_sets* correctly computes pseudoboundary sets and newly interior sets for all clusters of tree degree 2 whose level is at most l and that have no ancestor of tree degree 1 whose level is at most l, and it correctly computes boundary sets, newly interior sets, and separating set for all other clusters whose level is at most l. Since every cluster except the one containing all vertices has an ancestor of tree degree 1, the lemma will then follow.

The proof is by induction on level number. For the basis, $l = 0$. Any cluster at level at most 0 is a basic cluster, and the identification of boundary sets is clearly correct, as is the identification of pseudoboundary sets for clusters of tree degree 2. For the induction step, $l > 0$. We assume as the induction hypothesis that the above claim holds for clusters at any levels $l' < l$. For the newly interior sets of a cluster V_j, an examination of cases indicates that these are correctly computed from the boundary or pseudoboundary sets of the children of V_j.

For the sets other than newly interior, we consider cases for a cluster V_j at level l. Suppose cluster V_j at level l is of tree degree 0. It will have no boundary set since all vertices are contained within it. Suppose cluster V_j at level l is of tree degree 2. If V_j has just one child cluster, then by the induction hypothesis, that child's pseudoboundary set is correct. Clearly, the pseudoboundary set of V_j will be the same set. If V_j is the union of two clusters of tree degree 1 and 3, then its boundary sets and pseudoboundary sets are formed from the boundary set for its child of tree degree 1, which is correctly generated, by the induction hypothesis. Otherwise, V_j is the union of two clusters of tree degree 2. By the induction hypothesis, the pseudoboundary sets of the children are correctly computed. The only edges in those sets that are not in the pseudoboundary set of V_j are the edges from one child to the other that stay on the same side of the path between the boundary vertices of V_j. Algorithm *build_sets* correctly identifies these and removes them before concatenating the remaining lists of edges.

Next, suppose cluster V_j at level l is of tree degree 1. If V_j has just one child cluster, then by the induction hypothesis, that child's boundary set is correct. Clearly, the boundary set of V_j will be the same set. Otherwise, V_j is the union of two clusters

$V_{j'}$ and $V_{j''}$ of tree degree 1 and 2, respectively. By the induction hypothesis, the boundary set of $V_{j'}$ and the pseudoboundary set of $V_{j''}$ are computed correctly. If any edges in the boundary set of $V_{j'}$ have their other endpoint in a cluster up from $V_{j''}$, then no edge in the pseudoboundary set of $V_{j''}$ can be a separating edge. Thus removing those edges from the pseudoboundary sets of $V_{j''}$ that are in the boundary set of $V_{j'}$ and then concatenating the remainder will give the boundary set of V_j. Otherwise, there can be separating edges in the pseudoboundary sets of $V_{j''}$. A pseudoboundary set will contain first the edges with the other endpoint down from $V_{j''}$, then the separating edges for $V_{j''}$ and its relevant descendants, and finally the edges with the other endpoint up from $V_{j''}$. Thus, after removing edges from the boundary set of $V_{j'}$, the separating edges come next, in order, starting with the lowest edge in the remaining portions of the pseudoboundary sets of $V_{j''}$. Once these are identified and removed, the remaining portions of the pseudoboundary sets of $V_{j''}$ will comprise the boundary set of V_j.

Finally, consider the relevant descendants of cluster $V_{j''}$. We argue that *sep_edge* correctly computes the boundary sets and separating sets of all such clusters that have their boundary sets undefined. By the induction hypothesis, the pseudoboundary sets have been computed correctly. The proof that *sep_edge* correctly computes the boundary sets and separating sets is by induction on the distance to the deepest relevant descendant. For the basis, the distance is zero, and the cluster W has no proper relevant descendant. Then W is either a basic cluster or the union of clusters of tree degree 1 and 3. In both cases, the boundary sets of W will already be defined. For the induction step, we have the following. If the boundary set of W is already defined, then nothing need be done since all relevant descendants of W will have their boundary sets defined. Otherwise, if W has a single child W', then the recursive call for W' will, by the induction hypothesis for *sep_edge*, correctly compute the boundary sets of all relevant descendants of W'. Then the boundary set of W will be the boundary set of W', and there will be no separating set for W. Finally, if W has two children, then we have the following. If there are no separating edges with endpoints in W, then the boundary sets of W are the same as the pseudoboundary sets of W. In this case, there will be no separating edges with endpoints in either W' or W'', so that it does not matter what the u_L, u_R, d_L, and d_R parameters are in the recursive calls on W' and W''. Clearly, the number s of actual separating edges for W will equal the total number of separating edges with both endpoints in W minus the total number of separating edges with both endpoints in W' minus the total number of separating edges with both endpoints in W''. If $s > 0$, then one of s_L and s_R is 0 and the other is s. For $s_L > 0$, we must have one of the following cases. If $s''_T > 0$, then there are no edges from W' to a cluster up from W'', and thus the topmost separating edge out of the right side of W' is the $(c_R + 1)$st edge from the top in $pbs_R(W')$. If $s'_T > 0$, then there are no edges from W'' to a cluster down from W', and thus the bottommost separating edge out of the left side of W'' is the $(c_L + 1)$st edge from the bottom in $pbs_L(W'')$. If $s'_T = 0$ and $s''_T = 0$, then the $(u_L + 1)$st edge down in $pbs_L(W'')$ is the topmost separating edge for W, as is the $(d_R + s)$th edge up in $pbs_R(W')$. Upon the return from the recursive calls, the separating and nonseparating edges between W' and W'' are located and removed from the boundary sets of W' and W'', and the results are concatenated to give the boundary sets of W. Note that if $s_L > 0$, then edges up from the left of W are necessarily precisely the edges up from the left of W'', and if $s_L = 0$, it does not matter whether u''_L is set correctly or not. Similar remarks apply to the other cases. □

5. Data structures for embedded planar graphs. We describe the data structures for representing embedded planar graphs and show how to update them quickly when an update occurs. We use the topology tree of section 3 as a basis for our update data structure. Edges in any boundary, pseudoboundary, or newly interior set of a vertex cluster will be ordered according to the embedding and then represented by a balanced tree structure called an "edge-ordering tree." Next we define an "edge-ordered topology tree," which is a topology tree augmented by the edge-ordering information. We discuss how to update the edge-ordered topology tree to show the result of a swap. We then discuss how to update the edge-ordered topology tree to show the effect of the insertion or deletion of an edge or vertex. Finally, we show how to make edge-ordered topology trees fully persistent by introducing "internal names" of vertices, which are based on a vertex's position in the topology tree, and by showing how to keep track of internal indices while performing operations that change the structure of edge-ordering trees.

Let each set of nontree edges associated with a cluster, either a boundary, pseudoboundary, or newly interior set, be represented by a balanced tree called an *edge-ordering tree*. Each leaf in the edge-ordering tree will represent an edge in the corresponding set. When the pseudoboundary set of a cluster V_j is formed from the pseudoboundary sets of the children, do not change the edge-ordering trees for the pseudoboundary sets of the children, but rather build a new edge-ordering tree by introducing some new nodes and sharing subtrees with the already existing edge-ordering trees. The same idea applies for generating a representation of the boundary sets and the newly interior sets.

We define an *edge-ordered topology tree* for an embedded planar graph of maximum degree 3 to be the topology tree, along with pointers from each node in the topology tree to the edge-ordering trees for its one or two boundary sets, and its one or two newly interior sets. It is understood that the root of the tree has a pointer to an empty boundary set. We then note that the algorithm *build_sets* from the last section can be adapted to build an edge-ordered topology tree.

We now consider how to swap a nontree edge into the tree, replacing a tree edge. We will perform an operation similar to *basic_swap* of section 3, with the following additional work. When a node in the topology tree is removed, its boundary, pseudoboundary, and newly interior sets should be removed. Since subtrees are being shared in the edge-ordering trees, we keep a reference count in each node in a balanced tree indicating how many pointers have been set to point at it. When a node in an edge-ordering tree is removed, the reference count in each of its two children should be decremented. If a reference count goes to zero, then its node should be deleted. (In the case that we wish to enforce a particular bound on the time per operation, if some operation would cause a very large number of nodes to have their reference counts go to zero, then these nodes are saved in a list that can be reduced in size of the subsequent operations.) Thus edge-ordering trees will be removed as nodes are removed from the topology tree. In rebuilding the topology tree, whenever a parent for two nodes is created or changed, the boundary, pseudoboundary, and newly interior sets are recomputed in the fashion discussed in algorithm *build_sets*.

This approach is related to that in [F1], but we are specifying it carefully since we believe there is an error in [F1] with regard to the analysis of the running time. In particular, we believe that the time to search for the correct point to split boundary sets is underestimated in [F1]. The reason is the following. Consider a cluster V_j created by the union of two clusters $V_{j'}$ and $V_{j''}$, each of tree degree 2. We wish

to perform a simultaneous search in edge-ordering trees representing a boundary (or pseudoboundary) set for $V_{j'}$ and a boundary (or pseudoboundary) set for $V_{j''}$ to identify the edges common to both sets. It is easy to produce in constant time a suitable edge in one boundary set to test. The problem is determining whether that edge is also in the other boundary set. It is easy to keep a pointer from the leaf of one edge-ordering tree to the leaf in another edge-ordering tree representing the same edge but with respect to its other endpoint. However, it does not seem possible to deduce the name of the corresponding boundary set in constant time unless an excessive amount of work is performed on each update.

Since subtrees of the edge-ordering trees are shared, we give a top-down procedure to search within $V_{j'}$ and $V_{j''}$ simultaneously. To make the above searches efficient, we keep in each node of every edge-ordering tree the number of edges represented by the subtree rooted at that node. Then the position of the edges to be deleted can be computed quickly by keeping track of the positions of edges already deleted from the boundary sets. We then binary search to find the number of shared edges. For any test value, we search down through both trees to find the corresponding leaf in each tree. If the pointers in the leaves point at each other, then there are at least that many common edges; otherwise, there are fewer. Clearly, such a search will also involve $O(\log n)$ tests at $O(\log n)$ time per test, or $O((\log n)^2)$ time in total. Call the above procedure *plane_swap*.

LEMMA 5.1. *Procedure plane_swap correctly rebuilds an edge-ordered topology tree after a swap.*

Proof. Lemma 3.1 establishes that the topology tree is rebuilt correctly after a swap is performed. The simultaneous search of two boundary (or pseudoboundary) sets determines for any given size whether there is a common subset of that size, and it thus finds the size of the subset by binary search. Note that the search is indeed top-down, so that it works when subtrees of the edge-ordering trees are shared. Finally, the recomputing of the boundary, pseudoboundary, and newly interior sets is correct by arguments similar to those in the proof of Lemma 4.1. □

LEMMA 5.2. *The edge-ordered topology tree for an n-vertex embedded planar graph of maximum degree 3 uses $O(n)$ space, can be set up in $O(n)$ time, and can be updated to show the result of a swap in $O((\log n)^3)$ time.*

Proof. The topology tree itself uses $O(n)$ space. Each nontree edge will appear in two boundary sets (one for each endpoint) at the lowest level in the partition. Thus edge-ordering trees at level 0 use $O(n)$ space. We count the additional space used by the edge-ordering trees as follows. It follows from Lemma 2.2 that at level i, $i = 0, 1, \ldots, q$, there are at most $(5/6)^i n$ clusters. For a cluster V_j of size n_j, there are at most $c \log(2n_j)$ new nodes created in building additional boundary trees, where c is a constant. The sum of $c \log(2n_j)$ over all clusters V_j at level i is maximized when there are as many clusters as possible and each cluster is of roughly equal size. Thus we bound the total additional space used by edge-ordering trees by $\sum_{i=0}^{q} (5/6)^i n(1 + i \log(6/5))$. This quantity is clearly $O(n)$. It follows that the total space is $O(n)$.

We next discuss the setup time. Let V_j be a cluster of size n_j, with V_j being the union of clusters $V_{j'}$ and $V_{j''}$. If $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2, then charge the work of identifying each separating set to the parent W of the pair of clusters W' and W'' for which the separating set arose. This amounts to a charge of the cost of one search in the pseudoboundary sets of W, which is proportional to the square of the logarithm of the size of W. The time to remove the separating sets from the affected clusters should be apportioned similarly and will be of cost proportional

to the logarithm of the cluster size. Then the charge to generate the edge-ordering tree for cluster V_j will be at most $c(\log(2n_j))^2$, where c is a constant. The sum of $c(\log(2n_j))^2$ over all clusters V_j at level i is within a constant multiplicative factor of maximum when there are as many clusters as possible and each cluster is of roughly equal size. Thus we bound the setup time by $\sum_{i=0}^{q}(5/6)^i n(1 + i\log(6/5))^2$. This quantity is clearly $O(n)$.

By Lemma 3.1, a topology tree can be updated in $O(\log n)$ time to show the result of a swap, and thus $O(\log n)$ nodes are affected. As in setting up the edge-ordered topology tree, first find the pseudoboundary sets, then identify separating edges at higher nodes, and then correct the pseudoboundary sets to be boundary sets. As in the analysis of the setup time, charge the time to identify the separating edges to the lowest cluster to which both endpoints belong. Since there are $O(\log n)$ nodes that will be created, there will be $O(\log n)$ separating sets created, at most 1 per node created. Since each such node gets charged $O((\log n)^2)$, the total charge is $O((\log n)^3)$. In addition, there are a constant number of concatenations or splits per node, and there are at most $O(\log n)$ concatenations and splits that must be performed. Each such concatenation or split uses at most one search, at $O((\log n)^2)$ per search. □

The edge-ordered topology tree for the embedded planar graph can also be updated to reflect the insertion or deletion of an edge, as long as the insertion is consistent with the current embedding. The approach is similar to what is described in the discussion preceding Lemma 3.2, except that there is no need to adjust the value of z since $z = 1$ is independent of the number of vertices in the graph.

LEMMA 5.3. *The edge-ordered topology tree for an embedded planar graph of maximum degree* 3 *can be updated to show the result of an edge or vertex insertion or deletion that is consistent with the embedding in* $O((\log n)^3)$ *time, where* n *is the current number of vertices.*

Proof. The number of inserted and deleted vertices and edges will be a small constant. Thus by reasoning similar to that in the proof of Lemma 3.1, the number of nodes in the topology tree that are changed will be $O(\log n)$. The time bound then follows by the same argument as in Lemma 5.2. □

As shown in Lemma 5.2, an edge-ordered topology tree can be updated to show the result of a swap in $O((\log n)^3)$ time. It would at first appear easy to modify this representation in the same fashion as we did to the 2-dimensional topology tree in section 2 to give a persistent data structure. The difficulty is that on each update in the persistent structure, we made an unshared copy of a topology tree, at a cost of $O(m^{1/2})$ time. This was done because there appears to be no good way to both share subtrees and still have a pointer from each node to its parent. If we wish to achieve $O((\log n)^3)$ time per update in a persistent structure for planar graphs, we must do it without making complete copies of objects such as topology trees. Instead, we shall present a scheme for encoding new *internal* names of vertices. These names will be generated bottom-up and read top-down.

The names will be based on the structure of the topology tree and will thus change as the topology of the tree it is representing changes. We first note that the topology tree is a binary tree of height $O(\log n)$. We shall assume that any child of a node with exactly one child will be designated as a left child. We shall also assume that a node with two children will have the children designated as left or right in an arbitrary but fixed fashion. We encode the *level-l internal index* of a vertex, for $l = 0, 1, \ldots, q$, as follows. Let $a_l(v)$ be the ancestor at level l of the leaf in the topology tree that represents the vertex v. The level-0 internal index of any vertex v is the empty string.

For any l, $0 < l \leq q$, the level-l internal index of v is formed as follows. Take the level-$l-1$ internal index of v and concatenate onto its left a 0 if $a_{l-1}(v)$ is a left child of $a_l(v)$ and a 1 otherwise. To differentiate the original names from these new names, the original index of a vertex or vertex cluster will be called its *external index*.

Consider the graph in Fig. 1 along with the restricted multilevel partition of Fig. 3. We give the internal indices of vertex 13, using the topology tree as shown in Fig. 4. The level-l indices of vertex 13, for $l = 1, 2, 3, 4, 5$, respectively, are 1, 01, 001, 1001, and 11001.

Each node V_j at level l in the topology tree will have its (at most two) boundary vertices specified by both external index and level-l internal index and the (at most three) tree edges with precisely one endpoint in the cluster specified by external indices of the endpoints. In addition, each leaf in the topology tree will have the external name of its only vertex. If node V_j has two children, then it will have the level-l internal index for each endpoint of the tree edge that connects the two clusters corresponding to the children.

To represent the internal names of the endpoints of edges referred to in an edge-ordering tree, an additional mechanism is needed. Each value in an edge-ordering tree associated with node V_j will have the endpoints of its corresponding edge or pair of edges specified by a level-l' internal index for some $l' \leq l$. There will be an additional field *substr* in each node of the edge-ordering tree. The concatenation of the *substr* fields on a path from the root down to any node in the edge-ordering tree, when concatenated with a level-l' index there, will give a level-l internal index for the corresponding vertex. When two clusters are unioned, the *substr* fields at the roots of the corresponding edge-ordering trees are appended with either a 0 or 1 before the trees are concatenated. Clearly, these fields can be maintained as the trees are split and concatenated. Indeed, we note that the edge-ordering trees will be balanced naturally if every time we concatenate, we just add a new root above the current two (or three) roots, rather than performing some complicated rebalancing. This follows since there are only $O(\log n)$ levels in the topology tree.

We next discuss updating a persistent structure when it has been determined that a swap should be performed. We assume that the external and internal indices of the endpoints of the edges in the swap pair (e, f) will be provided. Using the internal indices of these endpoints, we can search down in the topology tree to the leaves, using constant time per level and setting temporary pointers as follows. For each node v on a path from the root down to an endpoint of e or f, set the parent pointer of v, the pointers between v and the nodes representing clusters adjacent to the cluster for v, and the parent pointer for each node representing an adjacent cluster.

Then proceed as in algorithm *plane_swap*, but doing the necessary work to maintain the following invariant with respect to nodes on the lists L_D, L_A, and L_C:

1. For the ancestor of any node in the lists L_D and L_C, there are temporary pointers to its parent.
2. For the ancestor of any node in the lists L_D, L_A, and L_C, there are temporary pointers to nodes representing adjacent clusters.
3. For any node representing a cluster adjacent to an ancestor of any node in the lists L_D, L_A, and L_C, there is a temporary parent pointer.

Whenever we insert a node y onto a list, where y was not previously an ancestor of a member of some list, we do the following. Either node y is put on list L_A and thus has no parent, or it is put on list L_C and it or a child of it must have been adjacent to a node on a list. In the latter case, the temporary pointer to the parent of y is

already set, as well as temporary parent pointers for all of its ancestors. We also know the external names of the endpoints of tree edges with precisely one endpoint in the cluster represented by node y. If such a tree edge does not have a temporary pointer associated with it, search up through the ancestors of y until we find an ancestor node x such that that edge is internal to x. We then use the internal indices for the endpoint of that edge to search down to a node z that represents a cluster that is adjacent to the cluster represented by node y. As the search goes back down, set temporary parent pointers along this path, as well as pointers from each ancestor of y to the node representing an adjacent cluster on the path up from z. When all such tree edges from the cluster represented by node y have been handled, the invariant is once again satisfied. We call this search from y an *invariant-enforcing search.*

Let the above approach be called algorithm *plane_persist_swap.*

THEOREM 5.4. *Algorithm plane_persist_swap maintains a fully persistent version of edge-ordered topology trees, using $O(n + k(\log n)^2)$ space to store k versions and generating a new version reflecting the result of a swap in $O((\log n)^3)$ time.*

Proof. By Lemma 5.1, algorithm *plane_swap* correctly updates an edge-ordered topology tree when a swap is performed. We verify that *plane_persist_swap* maintains the invariant stated above. The proof is by induction on the number of nodes that have been placed on L_D, L_A, and L_C. Initially, four nodes are placed on L_D and four nodes are placed on L_A. The nodes on L_D are the endpoints of edges e and f, and the ancestors of these nodes constitute the search paths along which temporary pointers are set. The four nodes on L_A are the replacements for the nodes on L_D, and copying the adjacency information of those nodes ensures that the ancestors of all adjacent nodes have the appropriate temporary pointers set. Subsequently, the following cases describe nodes placed on lists. These cases result from a close examination of *basic_swap.* For a node to be placed on L_D, a child of it must have been on L_D or L_C, or a child of it must have been adjacent to a node on L_C or L_A. The invariant is satisfied in all but the latter case, in which the invariant-enforcing search restores the invariant. For a node to be placed on L_C, either it had a sibling on L_D, or it had a child on L_C or L_A, or a child of it must have been adjacent to a node on L_C or L_A. The invariant is satisfied in all but the latter case, in which the invariant-enforcing search restores the invariant. For a node to be placed on L_A, either it or an adjacent node is on L_C, or a child is on L_C or L_A. The invariant-enforcing search restores the invariant in the former case. Thus the invariant is maintained. Given that the invariant is maintained, the algorithm is then able to access nodes to test whether or not to combine clusters.

Furthermore, we assert that for any quantity in an edge-ordering tree that is associated with an endpoint expressed by a level-l internal index, its level-l internal index can be maintained and manipulated as the edge-ordering trees are split and concatenated.

Finally, we establish the claimed resource bounds. From Lemma 5.1, the time used by *plane_swap* is $O((\log n)^3)$. In setting up temporary parent and adjacency pointers, the number of additional nodes examined is just a constant times the number of nodes examined in *plane_swap.* Furthermore, each node that is examined is examined just a constant number of times. Thus setting up temporary parent and adjacency pointers will take $O((\log n)^3)$ time. Each operation using level-l internal indices will take just constant time, so that the time to split and concatenate edge-ordering trees will be proportional to what it was in *plane_swap.* Thus the total time for one swap in a fully persistent version of edge-ordered topology trees will be $O((\log n)^3)$ time. By

Lemma 3.1, a topology tree can be updated in $O(\log n)$ time to show the result of a swap, and thus $O(\log n)$ nodes in the topology tree are affected. Since a constant number of boundary sets, newly interior sets, and separating sets are created for each node, and each split or concatenation of edge-ordering trees will create $O(\log n)$ nodes, $O((\log n)^2)$ additional space is used whenever a swap is performed. □

6. Basic approach for finding the *k* smallest spanning trees. In this section, we discuss the overall structure of our algorithm for finding the k smallest spanning trees, leaving out the description of the particular data structure that we employ. We shall assume that there are at least k distinct spanning trees of the graph. (It is easy to modify the algorithm to detect the case in which there are fewer than k distinct spanning trees.) We shall also assume that all edge weights are nonnegative. (If not, we can add a positive value to each edge weight to give an equivalent problem with all edge weights nonnegative.)

We first find a minimum spanning tree of our graph using the fast algorithm of [GGST] for general graphs or [CT] for planar graphs. Then we use Eppstein's technique to reduce the problem to one in which there are $O(k)$ vertices and edges [E]. If $k < m - n$, this technique identifies and deletes $m - n - k$ edges that will be in none of the k smallest spanning trees, and if $k < n$, it identifies and contracts $n - k$ edges that will be in all of these trees. Identifying these edges uses an algorithm for the sensitivity analysis of minimum spanning trees, either Tarjan's algorithm [T1], [T2] for general graphs or the algorithm of Booth and Westbrook [BW] for planar graphs. Also used is the linear-time selection algorithm [BFPRT]. We call the resulting graph the *contracted graph*. Note that the k smallest spanning trees of the contracted graph are in one-to-one correspondence with the k smallest spanning trees of our original graph.

Next, we transform the contracted graph into a graph in which every vertex has degree no greater than 3 using the transformation discussed in section 2. Note that each edge of cost $-\infty$ will be in all of the k smallest spanning trees, and each edge of cost ∞ will be in none. It follows that the k smallest spanning trees of the transformed graph are in one-to-one correspondence with the k smallest spanning trees of the contracted graph.

Let T_i denote the ith smallest spanning tree of the transformed graph. Thus T_1 denotes the minimum spanning tree. Having already found T_1, our algorithm will generate the $k - 1$ spanning trees $T_2, \ldots, T_k$ one at a time. Each tree T_i with $i > 1$ will be derived from some tree T_j, $j < i$, by a swap (e_i, f_i), in which a tree edge e_i is replaced by a nontree edge f_i. To guarantee that no tree is derived more than once, the trees will have certain restrictions placed on them of the form that any tree derived from T_j must include certain edges and exclude certain other edges. This inclusion–exclusion approach was presented by Lawler in [L1] and [L2, pp. 100–104].

Associated with each spanning tree T_i that is generated will be a *best-swap structure* R_i. We shall discuss the best-swap structure in greater detail later but mention a few properties now. Structure R_i will represent all spanning trees derivable from T_i by a sequence of swaps and will identify a swap for T_i of minimum cost. The algorithm will maintain a heap on the costs of the trees obtainable via these minimum-cost swaps. (When k is very large, our final version of the algorithm will manage the heap somewhat differently; see the discussion at the end of this section.)

We now proceed with a description of the rest of the algorithm. Given the minimum spanning tree T_1, we generate a best-swap structure for T_1. We initialize the heap with the value representing the cost of the spanning tree derived from T_1 by

applying the swap of minimum cost. We then repeat the following $k-1$ times. Extract the minimum from the heap. The extracted value represents the cost of a tree T_i produced by applying a swap (e_i, f_i) to spanning tree T_j. Generate a best-swap structure R_i from R_j using the fully persistent versions of the data structures discussed in sections 3 and 5. The changes in generating R_i from R_j should reflect the effect of two changes: replacing e_i by f_i in the spanning tree and resetting the cost of edge e_i to be the value ∞ for the purpose of determining the best swap. Resetting the cost of edge e_i effectively keeps edge e_i out of any of the spanning trees that are subsequently derived (transitively) from T_i. Finally, modify R_j to reflect the resetting of the cost of e_i to be $-\infty$ for the purpose of determining the best swap. Resetting the cost of edge e_i in this manner effectively forces edge e_i to be in all of the spanning trees that are subsequently derived (transitively) from T_j. The minimum costs identified by each of R_i and R_j correspond to swaps to be applied to T_i and T_j, respectively. Compute the costs of the trees generated by these trees and insert them into the heap. Note that the original costs of edges should be used in computing these costs. This completes the description of the repeat loop.

As we have described the algorithm, its output will be in the form of a minimum spanning tree plus a sequence of triples (e_i, f_i, j_i), $i = 2, 3, \ldots, k$. Note that it is easy to include the cost of tree T_i with the triple.

We can visualize the inclusion–exclusion using a binary tree B. Each node x in B represents a modified version $G(x)$ of the original graph G based on the inclusion and exclusion conditions. Associated with each node is the minimum spanning tree $T(x)$ for $G(x)$, along with a value that is the cost of $T(x)$ with respect to the edge weights in G. The root of B represents G, $T(root)$ is the minimum spanning tree T_1 of G, and the value associated with the root is the cost of T_1. For any node x in B, we determine the children of x as follows. If there is a swap of finite cost that can be applied to $T(x)$, let $(e(x), f(x))$ be the minimum-cost such swap. Then x will have right and left children. Graph $G(right(x))$ will be graph $G(x)$ with the cost of $e(x)$ reset to ∞, and spanning tree $T(right(x))$ will be $T(x) - e(x) + f(x)$. Graph $G(left(x))$ will be graph $G(x)$ with the cost of $e(x)$ reset to $-\infty$, and spanning tree $T(left(x))$ will be $T(x)$. This completes the definition of binary tree B.

As an example, we consider the spanning trees for the graph in Fig. 1. We shall name edges by their weights. In Fig. 9, we give binary tree B for this graph. The minimum spanning tree, as shown in Fig. 1, has a cost of 91 and is represented by the root of the tree in Fig. 9. The best swap for this tree is $(13, 14)$. The right child of the root represents the resulting tree, with cost 92. Note that edge 13 is excluded from being a member of any of the spanning trees represented by this node or any of its descendants. Conversely, edge 13 is required to be included in any tree represented by the left child of the root or any of its descendants. The minimum-cost swap given that edge 13 must be included is $(12, 14)$, yielding a tree with cost of 93. In our representation, the edge to the right child is labeled with a tree edge that is excluded, and the edge to the left child is labeled with a tree edge (the same edge) that must be included. To make the representation less cluttered, we subsequently put the included/excluded tree edge between the edges to the right and left children. Note that we label a nonroot node with its cost only if its spanning tree differs from that of its parent. Also, we do not draw the complete representation but only the first four levels, noting that all nodes shown have children except the lower rightmost one.

The time required by the algorithm will be the following. From [GGST], [E], [T1],

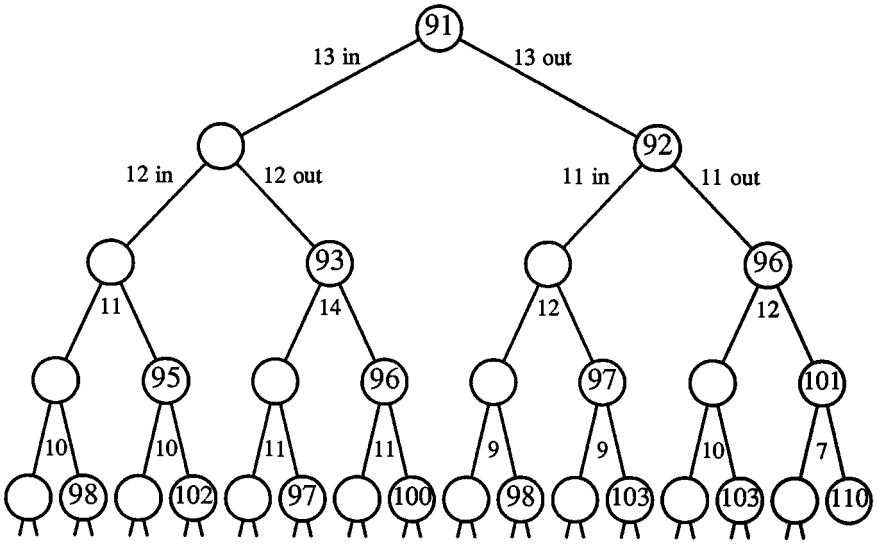


FIG. 9. *The first four levels of inclusion/exclusion for Fig.* 1, *with spanning tree costs indicated.*

[T2], [BFPRT], and [F1], finding the contracted graph and transforming it into one with maximum degree 3 will take $O(m \log \beta(m, n))$ time and $O(m)$ space. From [CT], [E], [BW], [BFPRT], and [F1], finding the contracted graph of a planar graph and transforming it into one with maximum degree 3 will take $O(n)$ time and space. In addition to setting up R_1, the algorithm will perform $2(k-1)$ updates of best-swap structures. With regard to the heap, $k-1$ *extractmin*s and $2(k-1)$ *insert*s will be performed. Thus the total time for all heap operations is $O(k \log k)$.

For very large values of k, the total time for maintaining the heap on the costs of trees may dominate the total time for updating the best-swap structures. In such a case, we may reduce the $O(k \log k)$ charge for maintaining the heap to $O(k)$ as follows. Note that when a best-swap structure is modified, the cost of the new spanning tree induced by the new best swap is never smaller than the spanning tree from which it was derived.

Suppose that these costs can be viewed as forming a min-heap. From [F4] it is known that the kth smallest value in a min-heap can be selected in $O(k)$ time. This algorithm is then used in place of the simple heap mechanism. Given $O(k)$ values that include the costs of all k smallest spanning trees, it is then straightforward to identify the costs of the k smallest spanning trees. Note, however, that these costs will not necessarily be output in sorted order.

It remains to show that the costs in binary tree B can be viewed as forming a min-heap. Since the spanning tree for each left child is the same as that of its parent, we need to compress B to get our min-heap. Note that for any node x such that $right(left(x))$ is defined, the value labeling $right(x)$ is no larger than the value labeling $right(left(x))$. This follows since a swap of smallest cost relative to $T(x)$ in $G(x)$ is of cost no larger than a swap of smallest cost relative to $T(left(x))$ in $G(left(x))$. Thus we generate our min-heap to contain nodes that correspond to a subset of the nodes in B in the following way. The root of the min-heap corresponds to the root of B. For any node y in the min-heap corresponding to node x in B, we determine the children of y as follows. If $right(x)$ is defined, then $right(y)$ is defined

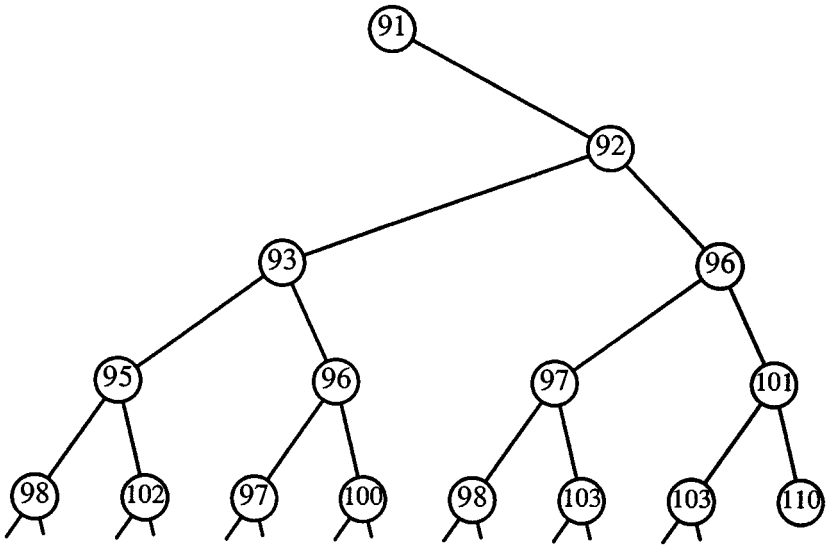


FIG. 10. *The min-heap induced by inclusion/exclusion on Fig.* 1.

to be a node that corresponds to $right(x)$. If node x has a parent, $parent(x)$, and if $right(left(parent(x)))$ is defined, then $left(y)$ is defined to be a node that corresponds to $right(left(parent(x)))$. Since the algorithm in [F4] first accesses an element in the min-heap only after having accessed its parent, the portion of the min-heap actually accessed by that algorithm can be constructed on the fly as we create and access our replacement data structures. Thus only $O(k)$ nodes in the min-heap need to be created. The binary min-heap corresponding to binary tree B in Fig. 9 is shown in Fig. 10.

7. Ambivalent data structures I: Best-swap structures. In this section, we adapt the data structures from section 3 to give an efficient best-swap structure for the case of general graphs. This will lead to an efficient algorithm for finding the k smallest spanning trees of a graph. We first give a more formal definition of an ambivalent data structure. Then we define what we call a "pseudoswap," which will allow us to design an ambivalent data structure. Using pseudoswaps, we next describe the information maintained in the nodes of the 2-dimensional topology tree and show how to generate this information for a node, given the information for its children. We then specify the best-swap data structure and discuss how to update this structure. We conclude with a claim of the time and space bounds on our algorithm for finding the k smallest spanning trees.

We now give a more formal definition of an ambivalent data structure. An item or set of items of data is said to be *substantiated* if that item or set of items represents an actual state of affairs. If that item or set of items does not represent an actual state of affairs but rather a hypothetical state of affairs that does not actually hold, then it is said to be *nonsubstantiated*. Let an item or set of items be called an *alternative* if examination of that item or set of items in isolation cannot determine whether it is substantiated or nonsubstantiated, but examination of a larger context of data will determine whether it is substantiated or nonsubstantiated. A *substantiated set of alternatives* is a set of two or more alternatives, one of which will be substantiated and

the rest of which will be nonsubstantiated. A data structure is said to be *ambivalent* if at many locations within itself it maintains substantiated sets of alternatives, and the data structure as a whole contains sufficient data to determine which alternative in every substantiated set of alternatives is substantiated. The ambivalence in our particular data structures comes from considering two clusters of vertices in a spanning tree and then attempting to represent how a nontree edge with one endpoint in each cluster relates to tree edges in these clusters. Such a relation might be whether a tree edge is in the cycle induced by the nontree edge. Determining whether a tree edge is in the cycle induced by the nontree edge may require information not stored with either cluster, i.e., it is information about the topology of the spanning tree. However, this information will always be available in our data structure as a whole.

We seek to build a data structure in which we can maintain a large set of swaps in a heap-like fashion so that a best swap can be identified quickly. We do this by considering nontree edges that have both endpoints in the same cluster and nontree edges that have their endpoints in different clusters. It is not hard to compute the most advantageous swap involving edges both of whose endpoints are in the same basic cluster. Thus the more challenging task is handling nontree edges that have their endpoints in different clusters. We set up ambivalent information for each cluster V_j of tree degree 2 as follows. Let V_r be a cluster at the same level as V_j. For any boundary vertex w of V_j, a *pseudoswap* is a pair (e, f) of edges, where f is a nontree edge having one endpoint in each of V_j and V_r and e is an edge on the path in the tree from w to the endpoint of f in V_j. Let w and w' be the boundary vertices of V_j. Suppose (e, f) is a pseudoswap for w and (e', f) is a pseudoswap for w'. Then one of those pseudoswaps is actually a swap, depending on whether w or w' is nearer V_r in tree T. Thus the set of pseudoswaps $\{(e, f), (e', f)\}$ is a substantiated set of alternatives.

Consider clusters V_j and V_r shown in Fig. 11. The spanning tree edges within these clusters are shown in bold, and the only nontree edge with an endpoint in each cluster is indicated by a dashed line. Some of the edges are labeled with their costs. Since the whole spanning tree is not shown, it is not clear whether the path in the tree between the endpoints of edge 18 goes through vertex w or vertex w'. Pseudoswap $(16, 18)$ is the best for clusters V_j and V_r and boundary vertex w. Pseudoswap $(17, 18)$ is the best for clusters V_j and V_r and boundary vertex w'.

We next discuss carefully the additional information that will be maintained in the nodes of the 2-dimensional topology tree. This includes the cost of a maximum-weight edge on the path between certain pairs of boundary vertices, the cost of a minimum-cost nontree edge between given pairs of clusters, the cost of a minimum-cost pseudoswap from a certain class of pseudoswaps, and the cost of a minimum-cost swap from a certain class of swaps.

We first discuss the cost of a maximum-weight edge on the path between certain pairs of boundary vertices. Let V_j be a vertex cluster with tree degree 2, and let $treemax(j)$ be the cost of a tree edge of maximum weight on the path between the two boundary vertices. We store $treemax(j)$ for a given j in node $V_j \times V_j$ in the 2-dimensional topology tree. If V_j is a basic vertex cluster, then $treemax(j)$ can be determined by inspection of V_j. If V_j is not a basic vertex cluster, then $treemax(j)$ can be computed in constant time given the $treemax$ values of the children in the topology tree and the cost of the tree edge between the children's corresponding clusters. Note that it is easy to keep track of the edge that yields the $treemax(j)$ value.

We next discuss the cost of a minimum-cost nontree edge between a given pair of

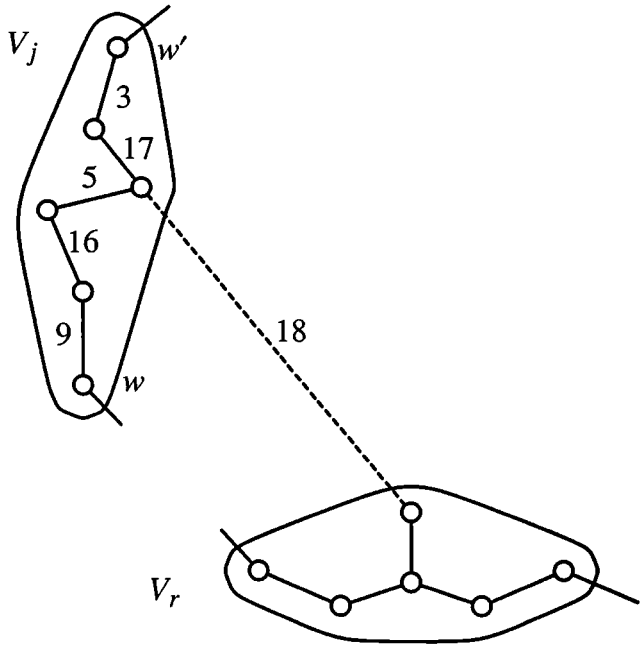


FIG. 11. *Example for illustrating best pseudoswaps.*

clusters. Let V_j and V_r be two distinct clusters at the same level. Let $nontreemin(j, r)$ be the cost of a nontree edge of minimum cost with an endpoint in each of V_j and V_r. Store $nontreemin(j, r)$ at node $V_j \times V_r$. If V_j and V_r are basic vertex clusters, then $nontreemin(j, r)$ can be computed for any particular j and all r by inspection of V_j. If V_j and V_r are not basic vertex clusters, then $nontreemin(j, r)$ is the minimum of the values $nontreemin(j', r')$, where $V_{j'} \times V_{r'}$ is a child of $V_j \times V_r$ in the 2-dimensional topology tree. Note that it is easy to keep track of the edge that yields each $nontreemin(j, r)$ value.

We next discuss the cost of a minimum-cost pseudoswap from a certain class of pseudoswaps. Let V_j and V_r be two distinct clusters at the same level. For each boundary vertex w of V_j, let $pswapmin(j, r, w)$ be the minimum value from the set of differences consisting of the cost of a nontree edge f with an endpoint in each of V_j and V_r, minus the cost of an edge e of maximum cost on the path in T from w to the endpoint of edge f that is in V_j. The values $pswapmin(j, r, w)$ for any particular value of j and r are stored in the node labeled $V_j \times V_r$. If V_j and V_r are basic vertex clusters, then $pswapmin(j, r, w)$ can be computed for any particular j, all boundary vertices w of V_j, and all r by inspection of V_j.

If V_j and V_r are not basic vertex clusters, then $pswapmin(j, r, w)$ can be computed in constant time given the *treemax* values for all children of V_j, and the *nontreemin* and *pswapmin* values for all children of $V_j \times V_r$ in the 2-dimensional topology tree. We specify this computation in detail. If the node for V_j in the topology tree has a single child $V_{j'}$, then $pswapmin(j, r, w)$ is the minimum of $pswapmin(j', r', w)$ taken over the one or two clusters $V_{r'}$ that form V_r. Otherwise, V_j is formed from two clusters $V_{j'}$ and $V_{j''}$, and we assume without loss of generality that w is contained in $V_{j'}$. Let w' be the boundary vertex of $V_{j'}$ adjacent to $V_{j''}$, and let w'' be the boundary vertex of $V_{j''}$ adjacent to $V_{j'}$. Then $pswapmin(j, r, w)$ is the minimum taken over the

one or two clusters $V_{r'}$ that form V_r of $pswapmin(j', r', w)$, $pswapmin(j'', r', w'')$, $nontreemin(j'', r') - c(w', w'')$, and $nontreemin(j'', r') - treemax(j')$. Once again, it is easy to keep track of the pair of edges that yield each $pswapmin(j, r, w)$ value.

We finally discuss the cost of a minimum-cost swap from a certain class of swaps Let V_j be a vertex cluster. Let $swapmin(j)$ be the cost of the minimum-cost swap such that the nontree edge has both endpoints in V_j. This value can be maintained in node $V_j \times V_j$ of the 2-dimensional topology tree. If V_j is a basic vertex cluster, then $swapmin(j)$ can be computed by inspection of V_j. If V_j is not a basic vertex cluster, then $swapmin(j)$ can be computed in constant time given the *swapmin*, *nontreemin*, and *pswapmin* values for children of $V_j \times V_j$ in the 2-dimensional topology tree. We specify this computation in detail. If the node for V_j in the topology tree has a single child $V_{j'}$, then $swapmin(j) = swapmin(j')$. Otherwise, V_j is formed from two clusters $V_{j'}$ and $V_{j''}$. Let w' be the boundary vertex of $V_{j'}$ adjacent to $V_{j''}$, and let w'' be the boundary vertex of $V_{j''}$ adjacent to $V_{j'}$. Then $swapmin(j)$ is the minimum of $pswapmin(j', j'', w')$, $pswapmin(j'', j', w'')$, and $nontreemin(j', j'') - c(w', w'')$. Once again, it is easy to keep track of the pair of edges that yield each $swapmin(j)$ value.

Our best-swap structure will be based on the fully persistent data structure described in section 3. A best-swap structure R_i will consist of a topology tree for tree T_i, a pointer to a 2-dimensional topology tree, many of whose subtrees are shared with other 2-dimensional topology trees, and pointers to representations of basic vertex clusters, most of which are shared. Each node in the topology tree will have the index of the corresponding cluster and a pointer to the node's parent. Each node in the 2-dimensional topology tree will have the indices of the corresponding clusters, along with the following. If the node is of type $V_j \times V_j$, it will have a *treemax* value and a *swapmin* value, while if it is of type $V_j \times V_r$, for $r \neq j$, it will have a *nontreemin* value and *pswapmin* values. Note that each such *treemax*, *nontreemin*, *pswapmin*, and *swapmin* value should also carry with it the index of the edge or edges involved and the indices of its basic vertex clusters. The representation of a basic vertex cluster V_j will consist of a list of vertices, a list of edges with both endpoints in the cluster (both tree and nontree edges), and, for every other basic cluster V_r, a pointer to a list of nontree edges with one endpoint in each of V_j and V_r. In addition, for each nontree edge with both endpoints in the same basic cluster, there will be the largest-cost tree edge with which it can swap.

We now discuss how to update a best-swap structure. If the swap causes a basic vertex cluster to be split or combined, generate a description of each new basic cluster V_j consisting of a list of vertices, a list of edges with both endpoints in V_j, and, for every other basic cluster V_r, a list of nontree edges with one endpoint in each of V_j and V_r. If the new basic cluster V_j is merged from old basic clusters V_j' and V_j'', determine the best swap for each nontree edge with one endpoint in each of V_j' and V_j'' as follows. Let tree edge (w', w'') connect V_j' to V_j'' with w' in V_j' and w'' in V_j''. Find the maximum-cost edge from each vertex in V_j' to w' and from each vertex in V_j'' to w''. Given nontree edge (v', v'') with v' in V_j' and v'' in V_j'', the best tree edge that can swap with (v', v'') can then be found in constant time. A similar approach can be used if a tree edge (w', w'') with both endpoints in basic cluster V_j has its cost set to $-\infty$ to find the new swaps that replace those involving edge (w', w'').

We next generate the new topology tree and the new 2-dimensional topology tree. For each basic vertex cluster V_j that has changed, do the following. For each pair of boundary vertices w and w' in V_j, determine the value $treemax(j, w, w')$. Next, determine the value $swapmin(j)$ by finding the minimum-cost swap over all best

swaps for nontree edges with both endpoints in V_j. For each set of nontree edges with one endpoint in each of V_j and V_r, set the appropriate pointers in the descriptions of V_j and V_r. Also, find the minimum-cost edge in the set, giving the $nontreemin(j, r)$ value. For each vertex v in V_j and each boundary vertex w of V_j, determine the maximum-cost tree edge on the path from v to w. For every other basic cluster V_r, examine every edge with one endpoint in each of V_j and V_r to find the best pseudo-swap for each boundary vertex w of V_j. Thus we can determine the $pswapmin(j, r, w)$ values. Create a new copy of the topology tree, and then modify the structure of the new topology tree and the 2-dimensional topology tree. As selected portions of the 2-dimensional topology tree are being rebuilt bottom-up, modify the information in the *treemax*, *nontreemin*, *pswapmin*, and *swapmin* fields.

THEOREM 7.1. *Let G be a graph with m edges and n vertices for which we know the minimum spanning tree T_1 and the best swap for each nontree edge. The best-swap structure R_1 can be set up in $O(m)$ time and space. A best-swap structure R_i can be updated in $O(m^{1/2})$ time, using $O(m^{1/2})$ additional space.*

Proof. The above algorithm indicates how to update basic clusters as a result of a swap or resetting the cost of a tree edge to $-\infty$. Once basic clusters have been updated, for any basic cluster V_j that has changed, the fields associated with nodes $V_j \times V_r$ must be recomputed. Then the fields for all ancestors of such nodes must be recomputed. The above algorithm does this.

Basic vertex clusters can be found in $O(m)$ time using procedure *cluster* from section 2. Similar to that in [F1], a restricted multilevel partition, a topology tree, and a 2-dimensional topology tree can be found in $O(m)$ time. Generating all other values can be done in time proportional to the number of them.

We next discuss the resources needed to update R_i. By Theorem 3.3, the time and space to perform the structural changes to the data structure is $O(m^{1/2})$. The time to compute each value in a newly created node is constant if these values are computed bottom-up. Thus the total time to update R_i is $O(m^{1/2})$. □

THEOREM 7.2. *The k smallest spanning trees of a weighted undirected graph can be found in $O(m \log \beta(m, n) + \min\{k^{3/2}, km^{1/2}\})$ time and $O(m + \min\{k^{3/2}, km^{1/2}\})$ space.*

Proof. The algorithm used is that described in section 6, with the best-swap structure R_1 just described. The algorithm uses the algorithm for finding the kth smallest element in a min-heap, as discussed at the end of section 6. Correctness follows from the discussion in section 6, plus Theorem 7.1.

As discussed in section 6, the time to find the minimum spanning tree and also find a transformed graph with $O(\min\{k, m\})$ edges will be $O(m \log \beta(m, n))$. By the discussion in section 6, there will be $O(k)$ such updates. By the discussion at the end of section 6, the cost of the kth smallest spanning tree can be found by performing $O(k)$ updates which produce $O(k)$ values, from which one selects the kth smallest in $O(k)$ time. By Theorem 7.1, updating a best-swap structure for a graph with $O(\min\{k, m\})$ edges will take $O((\min\{k, m\})^{1/2})$ time. The time bound then follows. By Theorem 7.1, each update will introduce $O(\min\{k^{3/2}, km^{1/2}\})$ additional space. The space bound then follows. □

8. Best-swap data structures for embedded planar graphs. In this section, we describe our ambivalent data structure to find a best swap for a spanning tree of an embedded planar graph. We first discuss how to store in fully persistent edge-ordering trees ambivalent information with respect to boundary sets, pseudoboundary sets, newly interior sets, and separating sets. We then describe how to compute the

minimum-cost swap for a vertex cluster, given the appropriate information about the cluster's children. We next describe the best-swap structure and its updating. Finally, we claim the time and space bounds for our algorithm that finds the k smallest spanning trees in a planar graph.

We first describe how to maintain ambivalent information in the edge-ordering tree for each set of edges. Consider a cluster V_j of tree degree 2. Consider the path P_j between the two boundary vertices of V_j. For any vertex u in V_j, we define $proj(j, u)$, the *projection* of u onto P_j, to be the vertex on P_j that is closest to u in the tree. First, consider the left boundary set $bs_L(V_j)$. For each edge (u, v) in $bs_L(V_j)$ with u in V_j, consider $proj(j, u)$. For edges (u, v) in $bs_L(V_j)$, there may be some vertex on P_j that has several vertices u projected onto it, and there may be some vertex that has no vertices projected onto it. We consider the *left modified path* $m_L(P_j)$ in which every vertex of $m_L(P_j)$ has exactly one vertex projected onto it except for the endpoints, which have none. Between two consecutive vertices x and y of $m_L(P_j)$, we shall have an edge whose cost is the cost of a maximum-cost edge on the subpath between x and y in P_j. In the case that x and y represent the same vertex in P_j, this cost will be $-\infty$. Note that $m_L(P_j)$ should be set up so as to be consistent with the planar embedding. We define and represent the *right modified path* $m_R(P_j)$ in a similar way. It is clear that a modified path is a generalization of a boundary set and can be represented by an edge-ordering tree.

We represent information about a modified path within the edge-ordering tree as follows.

- For each leaf, we keep
  1. the cost of the edge in the boundary set,
  2. the cost of the next edge in a given direction on the modified path,
  3. the cost of the swap using this next edge,
  4. the cost of the next edge in the other direction on the modified path,
  5. the cost of the swap using that next edge.
- For each nonleaf node in the edge-ordering tree, we keep
  1. the cost of the minimum-cost edge in the portion of the boundary set represented in the subtree rooted at the nonleaf node,
  2. the maximum of the costs of next edges in the given direction on the modified path in the subtree,
  3. the cost of the best swap if all edges in the portion of the boundary set for the subtree were able to swap with their next edges in this given direction,
  4. the maximum of the costs of next edges in the other direction on the modified path in the subtree,
  5. the cost of the best swap if all edges in the portion of the boundary set for the subtree can swap with their next edges in that other direction.

Given these values for any two siblings in the edge-ordering tree, the values for the parent can be computed in constant time.

If cluster V_j has tree degree 1, then the information is represented in an especially simple form. Since we know in which direction any nontree edge goes, we consider in computing *pswapmin* the swap of any nontree edge with a tree edge in V_j. We let P_j be the trivial path (of no edges) whose endpoints are both the single boundary vertex of V_j. The endpoints of all edges in the boundary set of V_j then project onto this single point in P_j, and all edges in $m(P_j)$ will have cost $-\infty$. For a cluster V_j of tree degree 2, the information corresponding to *treemax*, *nontreemin*, and *pswapmin* which we

maintained in section 7 is now held within the edge-ordering trees for the modified paths corresponding to the boundary sets, pseudoboundary sets, newly interior sets, and separating set. For a cluster V_j of tree degree 1, the same is true, except that no *treemax* is maintained.

As in section 7, we shall keep for each cluster V_j the cost $swapmin(j)$ of the best swap found within V_j. We discuss the additional changes that are necessary in the handling of edge-ordering trees when two clusters $V_{j'}$ and $V_{j''}$ are unioned to give cluster V_j. This involves examining four cases. Suppose $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3. We take $swapmin(j)$ to be the minimum of $swapmin(j')$ and $c(f) - c(e)$, where e is the tree edge between $V_{j'}$ and $V_{j''}$ and f is the edge of smallest cost in the boundary set of $V_{j'}$. In a fashion similar to that discussed before, the now modified edge-ordering tree for the boundary set of $V_{j'}$ will represent the modified edge-ordering tree for one of the two boundary sets of V_j.

Suppose $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2. We split and concatenate boundary sets as in sections 4 and 5. For any remaining portions of these sets, we now know in which direction the connection lies. For each newly interior set, we query the corresponding edge-ordering tree to get the minimum swap in the appropriate direction. (In particular, suppose $V_{j'}$ is "down" from $V_{j''}$. Then for the edge-ordering tree that represents the portion of the left boundary set of $V_{j''}$ that is newly interior, identify the minimum-cost swap in a downward direction. Do the same for the portion of the right boundary set of $V_{j''}$ that is newly interior.) Also, we query the edge-ordering tree for the remaining portion of each boundary set of $V_{j''}$ to identify the minimum-cost swap in an upward direction. We then take the minimum of the costs of these swaps, along with the values $swapmin(j')$, $swapmin(j'')$, and $c(f) - c(e)$, where e is a maximum-weight tree edge on the path between the top of $V_{j'}$ and the top of $V_{j''}$ and f is the edge of smallest cost in the boundary set of $V_{j'}$. Since V_j is of tree degree 1, we must reset the cost of tree edges in the modified path to be $-\infty$. We do this symbolically by letting the cost of the maximum-cost edge in this be set to $-\infty$. In any subsequent splits that affect this node, we propagate this value down as necessary in the edge-ordering tree. (This can be done by creating new nodes.)

When two boundary sets are concatenated together, values in the edge-ordering trees must be changed. One important change is that on each side of the modified path, we must find the maximum cost of a tree edge on the path between the nearest pair of edges, one in the boundary set of $V_{j'}$ and the other in the boundary set of $V_{j''}$, that will be in boundary sets for V_j. This can be done by taking the maximum over the tree edges in the edge-ordering trees representing newly interior edges on that side, along with the tree edge between $V_{j'}$ and $V_{j''}$.

Suppose $V_{j'}$ and $V_{j''}$ are both of tree degree 2. As discussed in sections 4 and 5, we split and concatenate pseudoboundary and boundary sets and form newly interior sets and a separating set. For each newly interior set, we query the corresponding edge-ordering tree to find the minimum swap in the appropriate direction. (Suppose $V_{j'}$ is "down" from $V_{j''}$. Then for the edge-ordering tree that represents the portion of the left boundary set of $V_{j''}$ that is newly interior, identify the minimum-cost swap in a downward direction. Do the same for the portion of the right boundary set of $V_{j''}$ that is newly interior. Then for the edge-ordering tree that represents the portion of the left boundary set of $V_{j'}$ that is newly interior, identify the minimum-cost swap in an upward direction. Do the same for the portion of the right boundary set of $V_{j''}$ that is newly interior.) For the separating set, we query the corresponding portions of the edge-ordering trees for the boundary sets of $V_{j'}$ and $V_{j''}$ to find the minimum swap in

the appropriate direction. (If the edges go from the left of $V_{j''}$ to the right of $V_{j'}$, then for the edge-ordering tree that represents the portion of the separating set incident on the left of $V_{j''}$, identify the minimum-cost swap in a downward direction, and for the edge-ordering tree that represents the portion of the separating set incident on the right of $V_{j'}$, identify the minimum-cost swap in an upward direction. The mirror-image case is similarly handled.) We then take the minimum of the cost of these swaps, along with $swapmin(j')$, $swapmin(j'')$, and $c(f) - c(e)$, where e is as before and f is the minimum-cost edge over all the newly interior sets and the separating set of V_j.

Finally, suppose $V_{j'}$ and $V_{j''}$ are both of tree degree 1. We take the minimum of $swapmin(j')$, $swapmin(j'')$, and $c(f) - c(e)$, where e is as before and f is the minimum-cost edge over all the newly interior sets of V_j. This concludes the examination of the four cases when two vertex clusters are unioned.

We next specify the swap structure. The best-swap structure R_i will consist of a pointer to a persistent edge-ordered topology tree for tree T_i. Shared by all R_i will be descriptions of the basic vertex clusters. The description of each basic vertex cluster will be in the form of the original name of the vertex contained in it, along with each edge incident on it, specified by the original names of the endpoints and the cost. Each node V_j at level l in the topology tree will have the cost of the pair of edges realizing its *swapmin* value, along with the level-l internal indices of the endpoints of the edges. As discussed in section 5, each value in an edge-ordering tree associated with node V_j will have its corresponding edge or pair of edges specified by a level-l' internal index for some $l' \leq l$.

We discuss how to update a best-swap structure when a swap occurs. Perform the procedure *plane_persist_swap* from section 5. Note that this procedure swaps nontree edge f in for tree edge e. We also wish to reset edge e to have cost ∞, so we reset the cost of this edge before rebuilding the edge-ordered topology tree. As we rebuild these structures, we update the *swapmin* value and the values in the edge-ordering trees of the affected nodes. As in section 7, a similar but simpler approach is used to handle an update when no swap is performed but the cost of a tree edge is reset to $-\infty$.

The full algorithm for generating the k smallest spanning trees uses the algorithm in section 6 and the persistent best-swap structure R_i.

THEOREM 8.1. *The k smallest spanning trees of a weighted undirected planar graph can be found in $O(n + k(\log n)^3)$ time and $O(n + k(\log n)^2)$ space.*

Proof. The algorithm used is that described in section 6, with the best-swap structure R_1 just described. The algorithm uses the algorithm for finding the kth smallest element in a min-heap, as discussed at the end of section 6. Correctness follows from the discussion in section 6, Theorem 5.4, plus the following. We first verify that $swapmin(j)$ is correctly computed when two clusters $V_{j'}$ and $V_{j''}$ are unioned to give cluster V_j. When $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3, then no nontree edges are incident on $V_{j''}$, and $swapmin(j)$ results either from within $V_{j'}$ or from swapping the edge between $V_{j'}$ and $V_{j''}$ with an edge in the boundary set of $V_{j'}$. When $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 2, then $swapmin(j)$ results either from within $V_{j'}$ or within $V_{j''}$, or from swapping a tree edge between the top of $V_{j'}$ and the top of $V_{j''}$ with an edge contained in the boundary sets of both $V_{j'}$ and V_j, or from swapping a tree edge in $V_{j''}$ with an edge in the boundary sets of both $V_{j''}$ and V_j, or from swapping an edge in a newly interior set with a tree edge. When both $V_{j'}$ and $V_{j''}$ are of tree degree 2, then $swapmin(j)$ results either from within $V_{j'}$

or within $V_{j''}$, or from swapping an edge in a newly interior set with a tree edge, or from swapping an edge in the separating set with a tree edge. When both $V_{j'}$ and $V_{j''}$ are of tree degree 1, the effect of all nontree edges in the boundary sets has already been accounted for, except for swapping with the tree edge connecting $V_{j'}$ and $V_{j''}$.

The time claim in the theorem follows from Theorem 5.4, the discussion at the end of section 6, and the following. We first consider setting up the trees for R_1. Clearly, the structure of the topology tree can be determined and set up in $O(n)$ time. The information in each leaf can be set up in constant time. Then the information in nonleaf nodes can be determined in a bottom-up fashion, at a constant cost per operation in setting up an edge-ordered topology tree as in section 5. To form the value $swapmin(j)$ for a cluster V_j, a constant number of operations must be performed on edge-ordering trees. As discussed in Lemma 5.2 and Theorem 5.4, these operations will take $O((\log n)^2)$ time. The space follows from Theorem 5.4 and the observation that a constant number of nodes per level are changed in the edge-ordered topology tree for each update and that the edge-ordering trees at a node can be updated by introducing $O(\log n)$ new nodes. □

Note that for values of $k < n$, $n + k(\log k)^3$ would seem to be better than $n + k(\log n)^3$. However, $k(\log n)^3 < n$ for $k < n/(\log n)^3$, and for $n/(\log n)^3 \leq k \leq n$, $\log k$ is $\Theta(\log n)$.

9. Ambivalent data structures II: 2-edge-connectivity information. In this section, we adapt the data structure from section 3 to give a data structure for updating and querying 2-edge-connectivity information in the case of general graphs. We first give two simple characterizations of 2-edge-connectivity and 2-edge-connected components. We then discuss the set of "complete paths," which are a partition of a subset of the spanning tree, and "partial paths," from which the complete paths are formed. We show how to generate these paths for a cluster when the paths of the children are known. We motivate how complete paths are used in answering a query. We next define "pseudocovering edges," which allow us to define an ambivalent data structure. We discuss the additional information stored at a node in the 2-dimensional topology tree and show how to generate it given the information of the children. We then give a summary of the update structure, including the specification of information associated with a basic cluster. We then describe how to perform queries and updates. Finally, we establish the resource bounds of our approach.

Let G be an undirected graph. Graph G is 2-*edge-connected* if there is no edge whose removal disconnects G. An edge whose removal disconnects G is called a *bridge*. The 2-*edge-connected components* of G are the subgraphs that result when all bridges are removed. We first present two propositions that characterize 2-edge-connectivity and 2-edge-connected components. Let T be a spanning tree of graph G. For each edge e in T, let $cover(e) = 1$ if there is a nontree edge f such that e is on the path in T between the endpoints of f, and let $cover(e) = 0$ otherwise.

PROPOSITION 9.1. *Graph G is 2-edge-connected if and only if $cover(e) = 1$ for each edge e in T.*

PROPOSITION 9.2. *Vertices v' and v'' are in the same 2-edge-connected component if and only if there is no edge e on the path from v' to v'' in T with $cover(e) = 0$.*

We seek to build a data structure in which we can maintain *cover* values easily. We partition the edges of the tree into two sets and maintain the *cover* information about each set differently. We use the topology tree and 2-dimensional topology tree to organize this information. Let the *boundary tree* be the set of all tree edges that are on a path in the tree between any two boundary vertices. The first set of edges

consists of tree edges that are not in the boundary tree, and the second set consists of all edges that are in the boundary tree. It is relatively easy to maintain information about the first set, so we shall concentrate for the moment on the second set of tree edges.

We next define partial and complete paths. We define a partition of the edges in the boundary tree into paths, which we call *complete paths*. This partition is based on the multilevel partition. The complete paths are built up from what we call *partial paths* in a manner that we now describe. There will be a partial path associated with each cluster, and there will be a complete path associated with each cluster that is the union of two clusters of odd tree degree. For any multilevel partition with more than one level, we have the following. No basic vertex cluster will have a complete path associated with it. A basic vertex cluster of tree degree 1 will have a partial path containing the path of zero length beginning and ending at its single boundary vertex. A basic vertex cluster of tree degree 2 will have a partial path consisting of the path in the tree between its two boundary vertices. A basic vertex cluster of tree degree 3 will have the partial path consisting of no edges and the single vertex contained in the cluster.

Let PP_j and CP_j designate the partial path and complete path (if any) of vertex cluster V_j. If the node in the topology tree for vertex cluster V_j has a single child $V_{j'}$, then $PP_j = PP_{j'}$. When two vertex sets $V_{j'}$ and $V_{j''}$ are unioned to form V_j, the partial paths are handled as follows. If $V_{j'}$ is of tree degree 1 or 2 and $V_{j''}$ is of tree degree 2, then the resulting vertex cluster will have a partial path that is the concatenation of $PP_{j'}$ and $PP_{j''}$ and the tree edge between them. If $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 3, then the resulting vertex cluster will have the following two paths associated with it. First, it will have a complete path that is the concatenation of $PP_{j'}$ and the tree edge between the two clusters. Second, it will have a partial path consisting of the single vertex of $V_{j''}$. If $V_{j'}$ is of tree degree 1 and $V_{j''}$ is of tree degree 1, then the resulting vertex cluster will have a complete path that is the concatenation of $PP_{j'}$ and $PP_{j''}$ and the tree edge between them.

Three of the four cases discussed above are shown in Fig. 12. Each small circle represents a single vertex. The top part of Fig. 12 shows two vertex clusters of tree degree 2 being unioned together. Their partial paths are shown in bold, and the new partial path for the unioned cluster is the path containing the bold edges and the dashed edge. The middle part of Fig. 12 shows vertex clusters of tree degree 1 and 3 being unioned together. The partial path of the cluster of tree degree 1 is shown in bold, and the new complete path for the unioned cluster is the path containing the bold edges and the dashed edge. The bottom part of Fig. 12 shows two vertex clusters of tree degree 1 being unioned together. Their partial paths are shown in bold, and the new complete path for the unioned cluster is the path containing the bold edges and the dashed edge. Note that the clusters shown are not basic clusters, which is why the partial paths of clusters of tree degree 1 appear to start in the "middle" of those clusters.

Note that a vertex cluster will have a complete path if and only if it is the union of two clusters of odd tree degree. For any complete path generated when clusters of tree degree 1 and 3 are unioned together, let the single vertex in the cluster of tree degree 3 be called the *top* of the path. We say that complete path P' *dominates* complete path P if and only if the top of path P is contained in P'.

Consider the restricted multilevel partition shown in Fig. 3. The complete paths are shown in Fig. 13. The complete path between vertices 12 and 13 will be associated

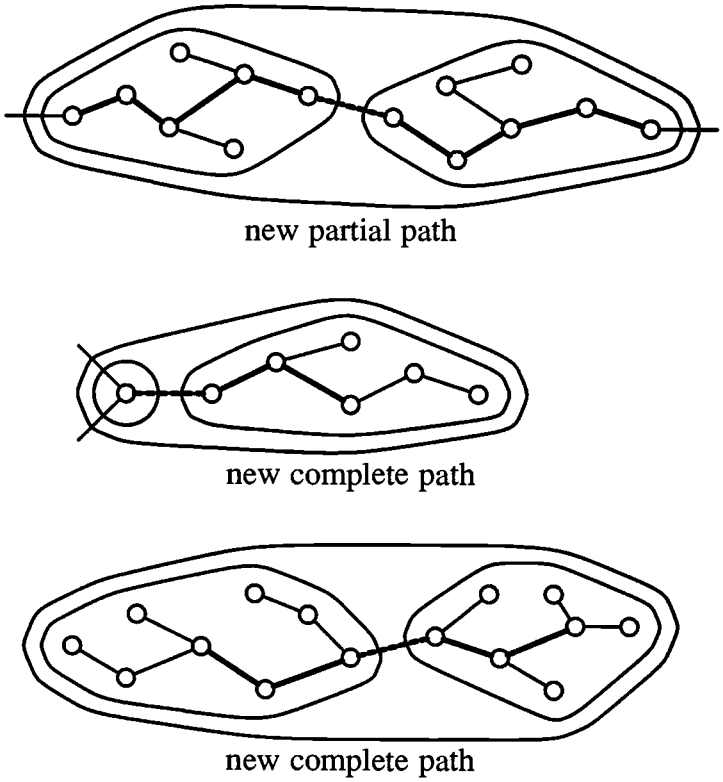


FIG. 12. *Combining partial paths.*

with V_{23} of Fig. 3, the complete path between 8 and 10 will be associated with V_{28}, the complete path between 4 and 7 will be associated with V_{32}, and the complete path between 1 and 14 will be associated with V_{37}. Each of the complete paths except the last has a top, and these are 12, 10, and 4, respectively. The complete path from 1 to 14 dominates each of the other paths. Examples of partial paths are the following. The partial path PP_{15} consists of the single vertex 11, PP_{16} is the path from 2 to 3, PP_{25} and PP_{31} are both the path from 1 to 3, PP_{32} consists of the single vertex 4, PP_{35} is the path from 1 to 4.

We describe how complete paths are used in answering a *same-2-edge-component* query. When a *same-2-edge-component* query on vertices v' and v'' is made, the path in the spanning tree T between v' and v'' is considered in the following way. Either the path contains no edges in the boundary tree, or it contains at least one edge in the boundary tree. In the former case, the path will be wholly contained in one basic cluster, and information associated with the basic cluster will be examined to see if there is a bridge between v' and v''. In the latter case, the path between v' and v'' will consist of three subpaths: the first and third subpaths will contain only edges not in the boundary tree, while second subpath will contain only edges in the boundary tree. First, information associated with the basic clusters containing v' and v'' will be examined to see if there is a bridge on the first or third subpath. If no bridge is found on these subpaths, then information associated with the complete paths containing edges on the path from v' to v'' will be examined.

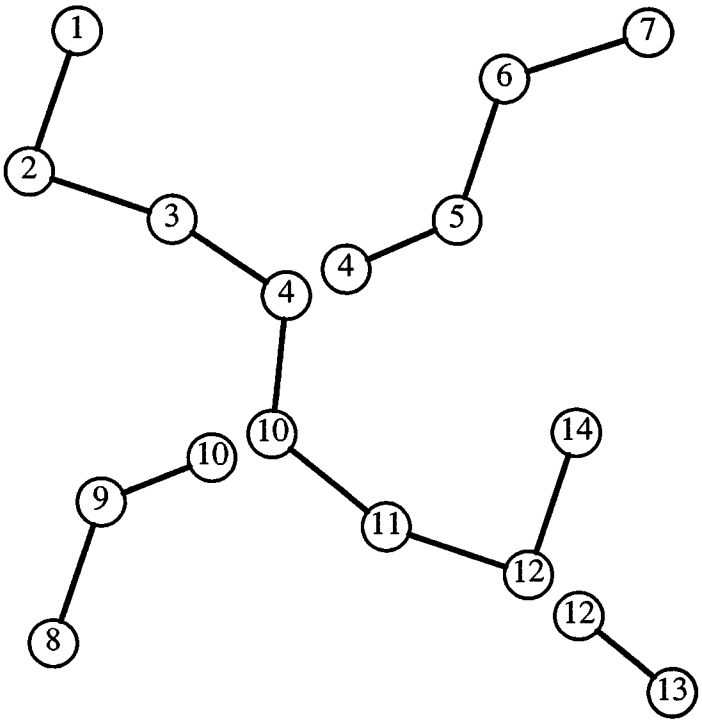


FIG. 13. *Complete paths for the restricted multilevel partition in Fig.* 3.

We maintain ambivalent information for V_j as follows. For boundary vertex w of V_j and cluster V_r at the same level, a nontree edge f with one endpoint in each of V_j and V_r is a *pseudocovering edge* for tree edge e if e is in PP_j and e is on the path in T from w to the endpoint of f in V_j. Pseudocovering edge f actually covers e if w is on the path in T between the endpoints of f. For each boundary vertex w of V_j and cluster V_r at the same level, we shall maintain a best pseudocovering edge in the following sense. A *best pseudocovering edge* for V_j, V_r, and w is a pseudocovering edge with respect to V_j, V_r, and w for the most tree edges in PP_j. This set of tree edges constitutes a subpath of PP_j with one endpoint at w.

Consider the graph in Fig. 1, with a restricted multilevel partition in Fig. 3. For cluster V_{18} and boundary vertex 6, edge 17 is a pseudocovering edge and also a best pseudocovering edge (since it is the only one). However, edge 17 does not actually cover any edges in V_{18}. For cluster V_{27} and boundary vertex 5, edge 17 is a pseudocovering edge. However, edge 17 is not a best pseudocovering edge for V_{27} and boundary vertex 5 since it covers zero edges in the partial path of V_{27} up to vertex 5, while edge 14 is a pseudocovering edge for V_{27} and vertex 5 that covers two edges in the partial path of V_{27} up to vertex 5.

We next discuss carefully the additional information that will be maintained in the nodes of the 2-dimensional topology tree. This includes pointers to partial and complete paths associated with the node, the number of edges in the partial paths, the number of edges from the top of a complete path to the first bridge (if any) in that path, and the best pseudocovering edges.

We first discuss the length of various paths. Note that by *distance* we mean the

number of edges in a path in the spanning tree. For the remainder of this section, we shall use the term distance in this way. Let V_j be a vertex cluster of tree degree 1 or 2. Let $length(j)$ be the length of the partial path in V_j. We store $length(j)$ at the node $V_j \times V_j$ in the 2-dimensional topology tree. If V_j is a basic vertex cluster, then any value $length(j)$ can be determined by inspection of V_j. If V_j is not a basic vertex cluster, then any value $length(j)$ can be computed in constant time given the lengths of the partial paths of the children in the topology tree.

We next discuss the pseudocovering edges. Let V_j and V_r be two distinct clusters at the same level, with V_j of tree degree 1 or 2. Let w be a boundary vertex of V_j, and let PP_j be a partial path that contains w as an endpoint. For any vertex u in V_j, recall from the previous section the definition of $proj(j, u)$, which is the vertex on PP_j that is nearest to u in the tree. (Here we use a slight extension of the definition from the last section, in that now we allow V_j to be also of tree degree 1.) For each boundary vertex w of V_j, let $maxcover(j, r, w)$ be the maximum of the distances from w to $proj(j, u)$ taken over the set of nontree edges (u, v) with u in V_j and v in V_r. (If there is no edge with one endpoint in each of V_j and V_r, let $maxcover(j, r, w)$ be $-\infty$.) A *best pseudocovering edge* is a pseudocovering edge that realizes a particular $maxcover(j, r, w)$ value. The values $maxcover(j, r, w)$ for any particular value of j and r are stored in node $V_j \times V_r$ of the 2-dimensional topology tree.

If V_j and V_r are basic vertex clusters, then $maxcover(j, r, w)$ can be computed for any particular j, all boundary vertices w of V_j, and all r by inspection of V_j. Then $maxcover(j, r, w)$ is the maximum distance $d(w, proj(j, u))$ taken over all edges (u, v) with u in V_j and v in V_r. If V_j and V_r are not basic vertex clusters, then $maxcover(j, r, w)$ can be computed in constant time given the *length* values for all children of V_j in the topology tree and the *maxcover* values for all children of $V_j \times V_r$ in the 2-dimensional topology tree. We specify this computation in detail. If the node for V_j in the topology tree has a single child $V_{j'}$, then $maxcover(j, r, w)$ is the maximum of $maxcover(j', r', w)$ taken over the one or two clusters $V_{r'}$ that form V_r. Otherwise, V_j is formed from two clusters $V_{j'}$ and $V_{j''}$, and we assume without loss of generality that w is in $V_{j'}$. If $V_{j''}$ is of tree degree 3, then the partial path of V_j is trivial and thus for the single vertex w in $V_{j''}$, $maxcover(j, r, w)$ is 0 if there is some nontree edge with one endpoint in each of V_j and V_r and is $-\infty$ otherwise. Such a nontree edge exists if $maxcover(j', r', w') > -\infty$ for $V_{r'}$ a child of V_r and w' the single boundary vertex of $V_{j'}$. Suppose that neither $V_{j'}$ nor $V_{j''}$ are of tree degree 3. Let w' be the boundary vertex of $V_{j'}$ adjacent to $V_{j''}$, and let w'' be the boundary vertex of $V_{j''}$ adjacent to $V_{j'}$. Then $maxcover(j, r, w)$ is the maximum taken over the one or two clusters $V_{r'}$ that form V_r of $maxcover(j', r', w)$ and $maxcover(j'', r', w'') + 1 + length(j')$. Note that it is easy to keep track of an edge that yields each particular $maxcover(j, r, w)$ value. (This is needed when we consider deleting a tree edge.)

We next discuss partial and complete paths. Let V_j be a vertex cluster. We maintain partial and complete paths in the following form. Each such path is represented by a balanced tree in which the leaves from left to right represent consecutive edges on the path. Associated with each node in the tree is a value *somecov* such that $cover(e) = 1$ for edge e on the path between boundary vertices if and only if $somecov(x) = 1$ for some node x on the path from the root to the leaf representing edge e. In addition, there is a value *allcov* such that $allcov(x)$ is 1 if and only if $somecov(x) = 1$ or $allcov(y) = 1$ for each child y of x. We assume that partial path PP_j and complete path CP_j (if it exists) are specified as a pointer to a structure of the above type.

We describe how to set up partial paths. If V_j is a basic vertex cluster, then the partial path can be set up by inspection of V_j. If V_j is not a basic vertex cluster and has just one child in the topology tree, then its partial path is the same as its child. If V_j is the union of two clusters $V_{j'}$ and $V_{j''}$ and is also not the set of all vertices, then its partial path is formed by concatenating the relevant partial paths of children, as discussed previously. The tree edge between $V_{j'}$ and $V_{j''}$ is initially assumed to have a *cover* value of 0. Certain *somecov* values are adjusted to reflect the effect of the best pseudocovering edges between $V_{j'}$ and $V_{j''}$. For example, suppose V_j, $V_{j'}$, and $V_{j''}$ are all of tree degree 2. Let w' be the boundary vertex of $V_{j'}$ adjacent to a vertex in $V_{j''}$, let w'' the boundary vertex of $V_{j''}$ adjacent to w', and suppose $maxcover(j', j'', w') \neq -\infty$. Then we modify the *somecov* values to reflect the fact that a subpath of length $maxcover(j', j'', w') + 1$, starting in $V_{j'}$ and ending at w'', is covered. This can be done by searching in the tree structure for the partial path to find the extreme edges in the subpath. A set of $O(\log n)$ nodes in the tree cover all and only the edges in the subpath, and the *somecov* values of these nodes should be set to 1. The *allcov* values of these nodes and their ancestors should also be adjusted. A similar operation would be performed with respect to $maxcover(j'', j', w'')$. Other cases, in which V_j is of tree degree less than 2 (meaning that the sum of the tree degrees of $V_{j'}$ and $V_{j''}$ is less than 4), are handled similarly.

We next discuss complete paths. Let V_j be a vertex cluster. Suppose V_j is a cluster that is the union of a cluster $V_{j'}$ of tree degree 1 and a cluster $V_{j''}$ of tree degree 3. Then the single vertex of $V_{j''}$ is the top of the complete path at V_j. Complete path CP_j is the result of concatenating the edge between $V_{j'}$ and $V_{j''}$ onto $PP_{j'}$. In addition, the *somecov* and *allcov* values must be modified to show the effect of nontree edges with precisely one endpoint in $V_{j'}$. Let $mcov(j)$ be one plus the maximum of the values $maxcover(j', r, w)$, where $r \neq j'$ and w is the single boundary vertex in $V_{j'}$. Then modify the *somecov* values to reflect the fact that the first $mcov(j)$ edges of CP_j from the top are covered. Let $toptobr(j)$ be the distance from the top of the complete path to the first bridge (if any) in the complete path. This can be found by searching in the tree structure for the complete path. There will be a bridge in the complete path if and only if the *allcov* value of the root of the tree structure is 0. Search down from the root, always taking the child representing a subpath closer to the top of the complete path when there are two children and both have *allcov* value equal to 0. Note that if both $V_{j'}$ and $V_{j''}$ are of tree degree 1, then V_j is the set of all vertices, and there is no top of the complete path.

The operations performed on partial and complete paths are concatenation and the update of *allcov* and *somecov* values. To make these operations efficient, we share common subtrees in the tree representations of partial and complete paths. When two paths are concatenated together, no nodes in the existing structures are changed or deleted. Rather, new nodes are allocated to handle structural changes. When an *allcov* or *somecov* value is changed, new copies are made of any node that has a value change or has a descendant that has a value that changes. To prevent the space from increasing without bound, a reference-count system should be used. The appropriate pointers PP_j and/or CP_j are maintained in node $V_j \times V_j$ of the 2-dimensional topology tree. Whenever a node V_j is identified as changing during an update, the appropriate pointers PP_j and/or CP_j are set to null, and then any nodes whose reference counts go to zero are deleted.

An update data structure Q will consist of a topology tree for a spanning tree T, a 2-dimensional topology tree, and representations of basic vertex clusters. Each node

in the topology tree will have the index of the corresponding cluster and a pointer to the node's parent. Each node in the 2-dimensional topology tree will have the indices of the corresponding clusters, along with the following. If the node is of type $V_j \times V_j$, it will have *length*, *PP*, *CP*, and *toptobr* values. If the node is of type $V_j \times V_r$, it will have *maxcover* values. The representation of a basic vertex cluster V_j will consist of a list of vertices, a list of edges with both endpoints in the cluster (both tree and nontree edges), and, for every other cluster V_r, a list of nontree edges with one endpoint in each of V_j and V_r and the associated $maxcover(j, r, w)$ values. In addition, we keep information that will help to determine if there is a bridge between two vertices in V_j, as discussed below.

We can find the *cover* values for edges not in the boundary tree as follows. Generate a *reduced cluster* for the basic cluster V_j as follows. The vertex set will be the same in the reduced cluster as in the basic cluster. Any edge (tree or nontree) with both endpoints in the basic cluster will be in the reduced cluster. For any nontree edge (u, v) with u in the basic cluster, v not in it, and $proj(j, u) \neq u$, edge $(u, proj(j, u))$ will be in the reduced cluster. We then find the biconnected components of the reduced cluster. Any tree edge that is in a biconnected component consisting of more than one edge will have nonzero *cover* value. To represent this information, we maintain the following. For each vertex u, keep the distance $d(u, proj(j, u))$, and also keep the distance $disttobr(u)$ to the bridge nearest to u on the path from u to $proj(j, u)$. (Let $disttobr(u)$ be ∞ if there is no bridge between u and $proj(j, u)$.) For each y on a partial path in a basic cluster, keep the distance to both endpoints of the partial path, and keep a data structure to find the lowest common ancestor [HT], [SV] with respect to the tree rooted at y and induced on all vertices u such that $proj(j, u) = y$.

We are now ready to discuss how a *same-2-edge-component*(v', v'') query is handled. Let v' and v'' be in basic clusters $V_{j'}$ and $V_{j''}$, respectively. If $j' = j''$ and $proj(j', v') = proj(j', v'')$, we do the following. Let $y = proj(j', v') = proj(j', v'')$. Find the lowest common ancestor z of v' and v'' in the tree rooted at y. It follows that v' and v'' are in the same bridge-component if and only if $disttobr(v') \geq d(v', y) - d(z, y)$ and $disttobr(v'') \geq d(v'', y) - d(z, y)$.

Suppose $j' \neq j''$ or $proj(j', v') \neq proj(j'', v'')$. If $disttobr(v') < d(v', proj(j', v'))$ or $disttobr(v'') < d(v'', proj(j'', v''))$, then v' and v'' are not in the same bridge-component. Otherwise, we examine the set of complete paths containing edges in the path from $proj(j', v')$ to $proj(j'', v'')$. We examine these paths as we search up through the topology tree. At the top of each complete path in the sequence except the highest one, we shall use the *toptobr* value to test if a bridge comes in the relevant portion of the complete path. At the highest complete path, we examine the appropriate *allcov* values to see if a bridge appears in the relevant portion.

We present the identification and the search of the complete paths in our algorithm *search_cps*. As input to our algorithm are the vertices v' and v'' and pointers to the nodes representing $V_{j'} \times V_{j'}$ and $V_{j''} \times V_{j''}$ in the 2-dimensional topology tree, where $V_{j'}$ and $V_{j''}$ are the basic clusters containing v' and v'', respectively. The algorithm sets variable *isbridge* to **true** if there is a bridge between $proj(j', v')$ and $proj(j'', v'')$. The algorithm maintains two variables $vert'$ and $vert''$, representing vertices on the path from $proj(j', v')$ to $proj(j'', v'')$, to indicate that it has checked for bridges between $proj(j', v')$ and $vert'$ and between $vert'$ and $proj(j', v')$. For the partial path of a cluster containing v' that it is examining, it maintains the distances $dist1'$ and $dist2'$ from $vert'$ to the endpoints of the partial path, and similarly for v'' and $vert''$. It uses these distances when checking a complete path for a bridge in the

relevant portion of the complete path, then resets $vert'$ and $vert''$ to reflect the additional portions of the path from $proj(j', v')$ to $proj(j'', v'')$ that have been checked. Note that it is not necessary to maintain both $dist1'$ and $dist2'$ if a preliminary search is used to determine which direction to search in on a cluster of tree degree 2. We have chosen not to employ such a preliminary search.

```
proc search_cps(v', v'', j', j'')
    isbridge ← false
    /* Initialize vert', w1', w2', dist1', dist2', and r' as follows: */
    vert' ← proj(j', v')
    w1' ← one endpoint of PP_j'
    w2' ← other endpoint of PP_j'
    dist1' ← distance from vert' to w1'
    dist2' ← distance from vert' to w2'
    r' ← j'
    /* Initialize vert'', w1'', w2'', dist1'', dist2'', and r'' similarly. */
    while V_r' has tree degree greater than 0 and vert' ≠ vert'' do
        /* Handle V_r' as follows: */
        if V_r' is of tree degree at most 2 and has a sibling
        then
            V_s' ← sibling of V_r'
            Let ws1' and ws2' be the endpoints of the PP_s'
            Without loss of generality, assume w2' is adjacent to ws1'.
            dist2' ← dist2' + 1 + length(s')
            w2' ← ws2'
            if V_s' is of tree degree 3
            then
                if the toptobr value at the parent of V_r' is less than dist2'
                then isbridge ← true
                endif
            endif
        endif
        V_r' ← parent of V_r'
        /* Handle V_r'' similarly. */
    endwhile
    if vert' ≠ vert''
    then
        Check the portion of CP(r') between vert' and vert''.
        if an allcov value along this portion of CP(r') equals 0
        then isbridge ← true
        endif
    endif
```

This concludes the discussion of how to handle a *same-2-edge-component* query. Let the above algorithm be called *2ec_query*.

LEMMA 9.3. *Algorithm 2ec_query identifies a bridge between two given vertices in* $O(\log n)$ *time.*

Proof. We first consider correctness. If $j' = j''$ and $proj(j', v') = proj(j', v'')$, then no edge of the boundary tree is in the path in T from v' to v''. The path from

v' to v'' will go from v' to z to v'', where z is the lowest common ancestor of v' and v'' in the tree rooted at $proj(j', v')$. Checking for a bridge in the two portions of this path yields the correct result.

If $j' \neq j''$ or $proj(j', v') \neq proj(j', v'')$, then the path from v' to v'' in T consists of a subpath from v' to $proj(j', v')$ that is not in the boundary tree, followed by a subpath from $proj(j', v')$ to $proj(j'', v'')$ in the boundary tree, followed by a subpath from $proj(j'', v'')$ to v'' that is not in the boundary tree. A bridge in the first or third subpath is identified by checking $disttobr(v')$ and $disttobr(v'')$. The second subpath is checked correctly by *search_cps*, as we now argue. Algorithm *search_cps* simultaneously searches up through the 2-dimensional topology tree from $V_{j'}$ and $V_{j''}$. For each pair of ancestors $V_{r'}$ and $V_{r''}$ of $V_{j'}$ and $V_{j''}$, respectively, the algorithm maintains the distance of $vert'$ and $vert''$ to the endpoints of the partial paths. In particular (referring to $V_{r'}$, with a similar argument for $V_{r''}$), if $V_{r'}$ has no sibling, then the same information will be maintained at its parent. If $V_{r'}$ is of tree degree 3, then it consists of a single vertex, which is necessarily $vert'$, so that the distances to the endpoints of the partial path of the parent, even when $V_{r'}$ is unioned with a sibling (of tree degree 1), will both remain 0. If $V_{r'}$ is of tree degree at most 2 and has a sibling, then one of the two distances to endpoints of the partial path must be updated. If the sibling is of tree degree 3, then the parent will contain a complete path, in which case the portion of the complete path from $vert'$ to the vertex of $V_{s'}$ must be checked and $vert'$, $dist1$, and $dist2'$ must be reset. When the tree degree of $V_{r'}$ is 0, then the complete path of $V_{r'}$ should be checked between $vert'$ and $vert''$ in the case that $vert' \neq vert''$.

We next consider the time. If v' and v'' are presented in terms of pointers, then clusters $V_{j'}$ and $V_{j''}$ can be identified in constant time; otherwise, $O(\log n)$ time can be used to search a dictionary. The values $proj(j', v')$ and $proj(j', v'')$ can be looked up in constant time. If $j' = j''$ and $proj(j', v') = proj(j', v'')$, then the lowest common ancestor z of v' and v'' can be found in constant time, and the values *disttobr* can be accessed in constant time as well. If $j' \neq j''$ or $proj(j', v') \neq proj(j', v'')$, then checking the first and third subpaths takes constant time to access and compare *disttobr* values. The second subpath can be checked in $O(\log n)$ time, as we now argue. Each iteration of the while loop of *search_cps* takes constant time. Searching the complete path for the cluster of tree degree 0 will take $O(\log n)$ time since there are $O(\log n)$ nodes in the complete path that cover exactly the portion of the path between $vert'$ and $vert''$, and it takes constant time to check the *allcov* value of each node. □

We consider our graph in Fig. 1, using the restricted multilevel partition of Fig. 3, with its associated complete paths, as shown in Fig. 13. There is only one bridge in the graph, edge $(3, 4)$. The value *toptobr* value is 0 for each of the complete paths from 4 to 7, 8 to 10, and 12 to 14. Recall that each basic vertex cluster is a single vertex. The query *same-2-edge-component*$(6, 8)$ will determine that there are no bridges from 8 to 10 on that complete path, that there are no bridges from 6 to 4 on the corresponding complete path, and that there are no bridges from 4 to 10 on the topmost complete path. Thus 6 and 8 are in the same 2-edge-connected component. The query *same-2-edge-component*$(2, 7)$ would examine the complete path from 7 to 4 and then the portion of the topmost complete path from 4 to 2, identifying a bridge, namely edge $(3, 4)$.

We next discuss how to insert or delete an edge. The approach builds on the way an edge was inserted or deleted in section 3. If a tree edge is to be deleted, then we

first attempt to swap a nontree edge in to replace it. If there is no edge with which it can swap, then deleting the edge splits the spanning tree and thus also the topology-tree-based data structures. On each of the swap, insert, or delete operations specified by the appropriate *inflate* or *deflate*, the following is done. First, all necessary changes are made to the basic clusters, and all information local to the changed or new basic clusters is computed. For a basic cluster, this information includes a description of each new cluster V_j, consisting of a list of vertices, a list of edges with both endpoints in V_j, and, for every other cluster V_r, a list of nontree edges with one endpoint in each of V_j and V_r, as well as the partial path PP_j, the values $proj(j,u)$, $length(j)$, $maxcover(j,r,w)$, and $disttobr(u)$, and the lowest common ancestor structures for each subtree of T rooted at a vertex on PP_j. For any basic cluster V_r, regroup edges with one endpoint in V_r and the other in a basic cluster V_j that has changed or is new, and recompute $maxcover(r,j,w)$ values.

After rebuilding certain basic clusters, rebuild portions of the 2-dimensional topology tree bottom-up. At nodes that are examined, recompute the information in the *maxcover*, *PP*, *CP*, *length*, and *toptobr* fields. Recall that rules were given earlier on how to generate a partial path from the partial paths of the children, including the *maxcover* values with an endpoint in each of two children. Also discussed is how to generate the complete path, computing and using the *mcov* value.

We discuss a little more how partial and complete paths are handled, since subtrees in the tree structures representing these paths are shared, and there are thus no parent pointers. Each vertex in a partial path will be identified by its distance from the end of the path. Each internal node of the balanced tree representing the path will contain a count of the number of edges in the subpath represented by that node. Thus a vertex in a path can be located in the tree using this positional information.

THEOREM 9.4. *Let G be a graph with n vertices and m edges at the current time. The update data structure Q can be set up in $O(m)$ time and space. Structure Q can be updated in $O(m^{1/2})$ time, while still using $O(m)$ space, and accommodates same-2-edge-component queries in $O(\log n)$ time.*

Proof. Given a spanning tree T for G, basic vertex clusters can be found in $O(m)$ using procedure *cluster* in section 2. Similar to that in [F1], a restricted multilevel partition, a topology tree, and a 2-dimensional topology tree can be found in $O(m)$ time. Since there are $O(m^{1/2})$ basic clusters, it follows from Lemma 2.2 that the number of nodes in the topology tree is $O(m^{1/2})$. Creating a partial path by concatenating the partial paths of the children will cost $O(\log n)$ for each node in the topology tree, or $O(m^{1/2}\log n)$ overall. Generating all other values can be done in time proportional to the number of them.

An edge insertion or edge deletion is handled by *inflate* or *deflate*, respectively. Since all data regarding clusters that change is recomputed, the updating is performed correctly. We next discuss the resources needed to update Q. The size of a description of a basic vertex cluster is $O(m^{1/2})$, and at most a constant number of basic vertex clusters are changed by any update operation. The time to generate the new information associated with a new basic cluster is $O(m^{1/2})$ if we are given the description of the basic cluster(s) from which it is formed. The number of nodes examined and created in generating the new 2-dimensional topology tree is $O(m^{1/2})$ by an argument similar to one in [F1]. The time to compute each value except $mcov(j)$ in a newly created node of the 2-dimensional topology tree is constant if these values are computed bottom-up. For $mcov(j)$ values, these can be found by scanning $maxcover(j',r,w)$ values. Each such *maxcover* value is from a node $V_{j'} \times V_r$ that is

examined as the 2-dimensional topology tree is rebuilt, so that each of $O(m^{1/2})$ nodes in the 2-dimensional topology tree can be charged a constant. There are $O(\log n)$ nodes of the form $V_j \times V_j$ that change, so that $O(\log n)$ complete or partial paths must be recomputed, at $O(\log n)$ time each. For the nearest bridge values, $O(\log n)$ complete or partial paths can have their nearest bridges change. The new values can be found in $O(\log n)$ time each. Thus the total time to update Q is $O(m^{1/2})$. By using a reference-count scheme, the form in the updated structure will be the same as if the structure were recomputed from scratch. Thus the space usage will remain $O(m)$.

The correctness and time for queries are established by Lemma 9.3. □

Acknowledgments. I am grateful to John Hershberger, Subhash Suri, Monika Rauch, Pino Italiano, Carsten Bjerring, Haim Kaplan, Sean Ahern, Sean Vyain, and Pok-yin Yu for especially helpful comments. I would also like to thank the referees for their thorough reading of the manuscript and many helpful suggestions.

REFERENCES

[ADKP] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka, *A simple parallel tree contraction algorithm*, J. Algorithms, 10 (1989), pp. 287–302.

[BFPRT] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1972), pp. 448–461.

[BW] H. Booth and J. Westbrook, *A linear algorithm for analysis of minimum spanning and shortest-path trees of planar graphs*, Algorithmica, 11 (1994), pp. 341–352.

[BH] R. N. Burns and C. E. Haff, *A ranking problem in graphs*, in Proc. 5th Southeast Conference on Combinatorics, Graph Theory and Computing 19, Utilitas Mathematica Publishing, Winnepeg, MB, Canada, 1974, pp. 461–470.

[CFM] P. M. Camerini, L. Fratta, and F. Maffioli, *The k shortest spanning trees of a graph*, Technical Report Int. Rep. 73-10, IEE–LCE Politecnico di Milano, Milan, Italy, 1974.

[CT] D. Cheriton and R. E. Tarjan, *Finding minimum spanning trees*, SIAM J. Comput., 5 (1976), pp. 310–313.

[CV] R. Cole and U. Vishkin, *The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time*, Algorithmica, 3 (1988), pp. 329–346.

[DSST] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, *Making data structures persistent*, J. Comput. System Sci., 38 (1989), pp. 86–124.

[E] D. Eppstein, *Finding the k smallest spanning trees*, BIT, 32 (1992), pp. 237–248.

[EGI] D. Eppstein, Z. Galil, and G. F. Italiano, *Improved sparsification*, Technical Report 93-20, Department of Information and Computer Science, University of California at Irvine, Irvine, CA, 1993.

[EGIN] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, *Sparsification: A technique for speeding up dynamic graph algorithms*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 60–69.

[F1] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[F2] G. N. Frederickson, *Ambivalent data structures for dynamic* 2*-edge-connectivity and k smallest spanning trees.* in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 632–641.

[F3] G. N. Frederickson, *A data structure for dynamically maintaining rooted trees*, in Proc. 4th ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1993, pp. 175–184.

[F4] G. N. Frederickson, *An optimal algorithm for selection in a min-heap*, Inform. and Comput., 104 (1993), pp. 197–214.

[FT] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[G] H. N. Gabow, *Two algorithms for generating weighted spanning trees in order*, SIAM J. Comput., 6 (1977), pp. 139–150.

[GGST] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, *Efficient algorithms for minimum spanning trees on directed and undirected graphs*, Combinatorica, 6 (1986), pp. 109–122.

[GI] Z. Galil and G. F. Italiano, *Fully dynamic algorithms for* 2*-edge-connectivity*, SIAM J. Comput., 21 (1992), pp. 1047–1069.

[Hy] F. Harary, *Graph Theory*, Addison–Wesley, Reading, MA, 1969.

[HT] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[Hl2] D. Harel, private communication, 1983.

[KIM] N. Katoh, T. Ibaraki, and H. Mine, *An algorithm for finding k minimum spanning trees*, SIAM J. on Comput., 10 (1981), pp. 247–255.

[L1] E. L. Lawler, *A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem*, Management Sci., 18 (1972), pp. 401–405.

[L2] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, 1976.

[MR] G. L. Miller and J. H. Reif, *Parallel tree contraction part* I: *Fundamentals*, in Randomness and Computation, Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 47–72.

[M] K. G. Murty, *An algorithm for ranking all the assignments in order of increasing cost*, Oper. Res., 16 (1968), pp. 682–687.

[SV] B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[T1] R. E. Tarjan, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[T2] R. E. Tarjan, *Sensitivity analysis of minimum spanning trees and shortest path trees*, Inform. Process. Lett., 14 (1982), pp. 30–33.

[WT] J. Westbrook and R. E. Tarjan, *Maintaining bridge-connected and biconnected components on-line*, Algorithmica, 7 (1992), pp. 433–464.

[Y] J. Y. Yen, *Finding the k shortest loopless paths in a network*, Management Sci., 17 (1971), pp. 712–716.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 539–556, April 1997

010

TIME-ADAPTIVE ALGORITHMS FOR SYNCHRONIZATION*

RAJEEV ALUR†, HAGIT ATTIYA‡, AND GADI TAUBENFELD§

Abstract. We consider concurrent systems in which there is an unknown upper bound on memory access time. Such a model is inherently different from the asynchronous model, where no such bound exists, and also from timing-based models, where such a bound exists and is known a priori. The appeal of our model lies in the fact that while it abstracts from implementation details, it is a better approximation of real concurrent systems than the asynchronous model. Furthermore, it is stronger than the asynchronous model, enabling us to design algorithms for problems that are unsolvable in the asynchronous model.

Two basic synchronization problems, consensus and mutual exclusion, are investigated in a shared-memory environment that supports atomic read/write registers. We show that $\Theta(\Delta \frac{\log \Delta}{\log \log \Delta})$ is an upper and lower bound on the time complexity of consensus, where Δ is the (unknown) upper bound on memory access time. For the mutual exclusion problem, we design an efficient algorithm that takes advantage of the fact that some upper bound on memory access time exists. The solutions for both problems are even more efficient in the absence of contention, in which case their time complexity is a constant.

Key words. distributed computing, consensus, mutual exclusion, timing-based algorithms

AMS subject classification. 68

PII. S0097539794265244

1. Introduction.

The possibility and complexity of synchronization in a distributed environment depends heavily on timing assumptions. In the asynchronous model, no timing assumptions are made about the relative speeds of the processes, while a timing-based model assumes known bounds on the speeds of the processes. Although the asynchronous model is weaker than the timing-based model, it provides a useful abstraction of the timing constraints, and algorithms designed for the asynchronous model work correctly in all possible environments. However, sometimes the assumption of asynchrony is too weak, and many problems have been shown to be unsolvable in the asynchronous model. These impossibility results never seem to bother practitioners, which brings up the question of whether such a model is the correct abstraction for modeling real systems.

We focus on an intermediate model which provides an alternative abstraction of the timing details of concurrent systems. We assume that there is an unknown upper bound on memory access time. This assumption is inherently different from the asynchronous model, where no such bound exists, and from timing-based systems, where such a bound exists and is known a priori. The appeal of the model lies in the fact that while it abstracts from implementation details, it is a better approximation of the real concurrent systems than the asynchronous model. Furthermore, it is stronger than the asynchronous model, enabling us to design algorithms for problems that are

* Received by the editors March 25, 1994; accepted for publication (in revised form) June 1, 1995. A preliminary version of this paper appeared in *Proc.* 26*th Annual Symposium on Theory of Computing* (*STOC*), ACM, New York, 1994, pp. 800–809.

http://www.siam.org/journals/sicomp/26-2/26524.html

† Bell Laboratories, Lucent Technologies, 700 Mountain Avenue, Murray Hill, NJ 07974 (alur@research.bell-labs.com).

‡ Computer Science Department, Technion, Haifa 32000, Israel (hagit@cs.technion.ac.il). The research of this author was performed while visiting at AT&T Bell Laboratories and was partially supported by United States–Israel Binational Science Foundation (BSF) grant 92-0233

§ The Open University, 16 Klausner Street, P.O. Box 39328, Tel-Aviv 61392, Israel and AT&T Bell Laboratories, Murray Hill, NJ, 07974 (gadi@cs.openu.ac.il).

unsolvable in the asynchronous model. The importance of a timing-based model with unknown bounds is also supported by an earlier work of Dwork et al. in the context of message-passing systems [DLS88] (see also [ADLS91]).

We use a shared-memory model where processes communicate with each other by reading and writing to shared registers. We assume that there is an upper bound, denoted by Δ, on the time required for a single access to shared memory. There is no lower bound on time needed to execute a step, but a process can delay itself explicitly by executing a statement $delay(d)$, for some constant d. The resulting model is called the *known-delay* model or the *unknown-delay* model depending on whether or not the bound Δ is known a priori. An algorithm in the unknown-delay model is required to be correct for all possible choices of Δ and hence cannot refer to Δ directly. We show that the unknown-delay model is inherently different from both the known-delay model and the asynchronous model by investigating two basic synchronization problems, consensus and mutual exclusion.

In the *consensus* problem, processes need to agree on a common output in the presence of possible failures [PSL80]. It has been proven that when even one process can fail, the consensus problem is not solvable in asynchronous systems [DDS87, FLP85, LA87]. In the known-delay model, there is an algorithm which tolerates any number of failures and terminates within time $O(\Delta)$ [AT96].

Our first result is a consensus algorithm that works in the unknown-delay model. The algorithm guarantees that in every possible execution, processes never decide on conflicting values and the decision value is an input value of some process. If every step finishes within time Δ, then a process decides within time $O(\Delta \cdot fac^{-1}(\Delta))$ irrespective of the failures of other processes, where fac^{-1} is the inverse of the factorial function;[1] note that $fac^{-1}(d) = \Theta(\frac{\log d}{\log \log d})$. Furthermore, the algorithm is *fast*: in absence of contention, a process decides after a constant number of its own steps.

Our second result shows that the worst-case time complexity of any two-process algorithm for consensus in the unknown-delay model is $\Omega(\Delta \cdot fac^{-1}(\Delta))$; this implies that our algorithm is time-optimal. The lower bound implies that not knowing Δ multiplies the time complexity by a factor of $fac^{-1}(\Delta)$.

The *mutual exclusion* problem is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. A mutual exclusion algorithm satisfies the *fast access property* if in the absence of contention, a process needs to execute only a constant number of steps in order to enter or exit its critical section. In [Lam87], Lamport presented a fast mutual exclusion algorithm that satisfies the fast-access property. In the presence of contention, however small, the winning process in Lamport's algorithm may have to check the status of all of the other n processes before it is allowed to enter its critical section. In the known-delay model, there is an algorithm that satisfies the fast-access property without requiring the winning process to check the status of all of the other n processes in the presence of contention [AT96]. Other algorithms which satisfy the fast access property can be found in [CS93, MS93, Sty92, YA93].

Our third result is a mutual exclusion algorithm for the unknown-delay model where in the presence of contention, a process needs to delay itself for $2 \cdot \Delta$ time units before entering its critical section. The algorithm has a "warm-up" period during which the processes might have to access n registers before entering the critical section. The algorithm always provides fast access in the absence of contention.

[1] For a real number $d > 0$, $fac^{-1}(d)$ is the smallest natural number r such that $r! \geq d$.

For both problems, the knowledge of Δ is beneficial: in the case of consensus, the problem becomes solvable, and in the case of mutual exclusion, more efficient solutions can be obtained. Our results imply that these benefits can be achieved even when Δ is unknown.

Dwork et al. have studied the consensus problem in message-passing systems where there are unknown bounds on the time to deliver a message and on processes' speed [DLS88]. Their work concentrates on the percentage of faulty processes (compared to the total number of processes) that can be tolerated. Our consensus algorithm is wait-free, that is, it can tolerate any number of crash failures.

Herzberg and Kutten [HK89] have studied a message-passing model where an a priori upper bound on message delivery time is known but is much larger than the actual message delay; this encourages the use of the (unknown) message delivery time in the algorithm. They considered the problem of detecting faulty processes.

In section 2, a formal model is introduced and the issue of how to measure the time complexity is discussed. Section 3 is dedicated to the consensus problem; matching upper and lower bounds are presented for the worst-case time of a consensus algorithm. The fast algorithm for mutual exclusion appears in section 4. We conclude with a discussion of our results and directions for future work.

2. A timing-based model. In this section, we outline our model of distributed systems. Processes are modeled as (possibly infinite) state machines communicating via shared memory consisting of registers that support atomic reads and writes.

A *configuration* of the system includes the state of each process and the values of all shared registers. An *event* is a single step of some process and is either a read of a shared register, or a write to a shared register, or simply an update of the internal state of a process. Processes can also execute a delay statement $delay(d)$, for a positive integer d, and its effect on a configuration is the same as a skip statement.

As in the standard interleaving semantics, an *execution* α of the system is an alternating sequence $s_0 \stackrel{e_0}{\rightarrow} s_1 \stackrel{e_1}{\rightarrow} \cdots$ of configurations s_i and events e_i such that (1) the initial configuration s_0 satisfies some initial conditions and (2) every configuration s_{i+1} is derived from the previous configuration s_i by executing the event e_i.

We allow only crash failures: a failed process simply ceases to participate. Formally, a process p is *nonfaulty* in an execution α if and only if either α is finite or p takes infinitely many steps in α.

The notion of an execution captures only the asynchronous part of the system and not its timing requirements. Define the *explicit delay* of an event e, denoted by $d(e)$, to be n if e is the delay statement $delay(n)$ and 0 otherwise. A *time assignment* τ for an execution α is a mapping that assigns a real-valued occurrence time τ_i to each event e_i in α such that

1. the occurrence times are nondecreasing,
2. if α is infinite, then the sequence of occurrence times is unbounded,[2] and
3. whenever two events e_i and e_j are consecutive steps of the same process, then difference $\tau_j - \tau_i$ is greater than $d(e_i)$.

The last requirement captures the assumption regarding the lower bounds on execution speeds. A delay statement $delay(d)$ by a process p delays p for at least d time units before it can continue. For other statements, we simply require that a step of a process takes nonzero time.[3] Note that adjacent events belonging to different processes can be assigned the same time. As an example, consider the following

[2] Our results do not depend on this second requirement.

[3] Our definition allows delays of read or write steps to be as small as we want. In a context

execution, where each event is labeled with the process it belongs to:

$$\alpha_{\text{sample}}: \; s_0 \overset{p:read(x)}{\longrightarrow} s_1 \overset{q:write(y)}{\longrightarrow} s_2 \overset{p:delay(4)}{\longrightarrow} s_3 \overset{p:read(y)}{\longrightarrow} s_4 \overset{q:read(x)}{\longrightarrow} s_5.$$

A time assignment τ for the above execution α_{sample} is a sequence $\tau_0 \leq \tau_1 \leq \cdots \leq \tau_4$ such that

$$\tau_2 - \tau_0 > 0, \qquad \tau_3 - \tau_2 > 4, \qquad \tau_4 - \tau_1 > 0.$$

Thus a possible time assignment is

$$\tau_{\text{sample}}: \; \tau_0 = 0, \quad \tau_1 = 0, \quad \tau_2 = 0.1, \quad \tau_3 = 4.5, \quad \tau_4 = 5.$$

The definition of the explicit delay of an event is also extended to finite executions: for a finite execution $\alpha = s_0 \overset{e_0}{\rightarrow} \cdots s_n$, let $d(\alpha)$ denote the sum $\sum_{0 \leq i < n} d(e_i)$ of explicit delays of all the events in α. For instance, $d(\alpha_{\text{sample}})$ equals 4.

The assumption about the time needed to access shared memory is reflected in the following notion of admissibility. Let Δ be a positive real number.

> A timing assignment τ for an execution α is said to be Δ-*admissible* if and only if whenever two events e_i and e_j are consecutive steps of the same process, $\tau_j - \tau_i \leq \Delta + d(e_i)$.

Thus if the ith step in an execution is a read or a write by process p, then the next step by process p must be within time Δ; if it is the delay statement $delay(d)$, then p's next step must be within time $\Delta + d$. If a process does not take the next step within this bounded time period, then it can never take a step, implying a crash failure.

Every execution has several time assignments but may not have a Δ-admissible time assignment for a given Δ; this is because delay statements restrict the possible time assignments. An execution α is Δ-*admissible* if there exists a Δ-admissible time assignment for α. For our sample execution α_{sample}, the timing assignment τ_{sample} is 5-admissible (in fact, it is Δ-admissible if and only if $\Delta \geq 5$). The execution α_{sample} itself is Δ-admissible for every $\Delta > 4$.

A problem such as consensus or mutual exclusion is usually specified by listing the properties to be satisfied by all the executions. An algorithm A satisfies a property ϕ in the asynchronous model if and only if all of its executions satisfy ϕ. While solving a problem in the timing-based model with an upper bound of Δ on the step time, a key issue is whether the processes know the upper bound Δ.

In the *known-delay* model, we assume that individual processes know the upper bound Δ. Consequently, delay statements can refer directly to this value, and a process can enforce every other (nonfaulty) process to take at least one step by executing the statement $delay(\Delta)$. To solve a problem in this model, we want a family of algorithms $A(\Delta)$, parameterized by the upper bound Δ, such that for each Δ, all Δ-admissible executions of $A(\Delta)$ satisfy all the requirements of the problem.

In the *unknown-delay* model, we assume that some upper bound exists, but it is not known to individual processes a priori. In this model, we want a single algorithm A that works for all possible values of Δ without referring to its actual value. We will say that an algorithm A satisfies a property ϕ in the unknown-delay model if for every Δ, all Δ-admissible executions of A satisfy ϕ.

Before we consider specific problems, let us observe one property of algorithms in the unknown-delay model. A property ϕ is a *safety* property if the following holds:

---

where a nonnegligible lower bound, say ϵ, on these steps is more appropriate, we can simply insert the statement $delay(\epsilon)$ after every step.

An infinite execution α satisfies ϕ if and only if all finite prefixes of α satisfy ϕ; that is, a safety property has to be *prefix-closed*. The unknown-delay model is the same as the asynchronous model as far as safety properties are concerned.

LEMMA 2.1. *An algorithm A satisfies a safety property ϕ in the asynchronous model if and only if it satisfies ϕ in the unknown-delay model.*

Proof. Clearly, if A satisfies a property ϕ in the asynchronous model, then it satisfies ϕ in the unknown-delay model. Suppose A does not satisfy ϕ in the asynchronous model. Then there is an execution α of A which violates ϕ. If ϕ is a safety property, then there is a finite prefix α' of α such that α' violates ϕ. The finite execution α' is Δ-admissible for every $\Delta \geq d(\alpha')$. This implies the lemma. ∎

Lemma 2.1 does not hold for liveness properties such as termination.

We now define our time complexity measures. Given an execution α and a time assignment τ for it, suppose $time(\alpha, \tau)$ is a measure of time taken according to τ. The exact definition of *time* depends on the problem and on whether we are computing the worst-case complexity or the contention-free complexity. For instance, in consensus, *time* may denote the maximum time spent by a process between its first step and its decision step in α. We denote by $time_\Delta(\alpha)$ the maximum of $time(\alpha, \tau)$ over all Δ-admissible time assignments τ for α. For an algorithm A, $time_\Delta(A)$ denotes the maximum of $time_\Delta(\alpha)$ over all Δ-admissible executions α of A.

Sometimes we will also need an estimate of how much time is spent due to explicit delay statements. For an execution α, let $min\text{-}time_\Delta(\alpha)$ denote the greatest lower bound on $time(\alpha, \tau)$ over all Δ-admissible time assignments τ for α; it gives the minimum time spent by a process in α. For instance, if we define $time(\alpha_{\text{sample}}, \tau)$ as $\tau_4 - \tau_0$, then for $\Delta > 4$, $time_\Delta(\alpha_{\text{sample}})$ is $2\Delta + 4$ and $min\text{-}time_\Delta(\alpha_{\text{sample}})$ is 4.

If *time* measures the *contention-free* complexity, which is the time spent by a process when it executes by itself, then according to [Lam87, AT96], an algorithm is *fast* if and only if $time_\Delta(A)$ is $O(\Delta)$ and $min\text{-}time_\Delta(A)$ is zero. This implies that in absence of contention, a process executes only a constant number of steps and no explicit delay statements.

3. Time-adaptive consensus. In this section, we consider the problem of (binary) consensus in the unknown-delay model and provide tight bounds for its worst-case time complexity.

The *consensus problem* is to design an algorithm in which all correct processes reach a common decision based on their initial inputs [PSL80]. Formally, the problem is defined as follows. There are n processes, and each process p_i has an input value $in_i \in \{0, 1\}$. A process p_i *decides* on a value $v \in \{0, 1\}$ by executing the statement $decide(v)$. It may decide at most once. The consensus problem requires the following:

- *Agreement:* There exists a *decision value* $out \in \{0, 1\}$ such that if a process p_i decides on the value v, then $v = out$.
- *Validity:* If a process p_i decides on the value v, then v equals the input value in_j for some process p_j.

Thus no two processes decide on conflicting values, and if all input values are the same, then that value must be the decision value. Apart from the above safety requirements, we want the correct processes to eventually decide the following:

- *Wait freedom:* Each process p_i either takes only finitely many steps or decides on some value.

The requirement of wait freedom means that one process cannot prevent another process from reaching a decision, and thus the algorithm must tolerate arbitrary number of process failures.

3.1. The algorithm. In this section, we present a consensus algorithm. The algorithm always guarantees the safety requirements of agreement and validity. The liveness requirement is ensured using timing assumptions. If every step finishes within time Δ, then a process decides within time $O(\Delta \cdot fac^{-1}(\Delta))$ irrespective of the failures of other processes, where fac^{-1} is the inverse of the factorial function. Thus the algorithm satisfies wait freedom in the unknown-delay model. Furthermore, the algorithm is fast: in the absence of contention, a process decides after a constant number of steps without explicitly delaying itself.

Recall that there is no wait-free algorithm for consensus in the asynchronous model. In the known-delay model, a process can use its knowledge about the speeds of other processes by executing the statement $delay(\Delta)$, and it is possible to design a wait-free solution [AT96]. Let us see how such an algorithm can be constructed when the upper bound is not known.

Initially, each process starts with some estimate, say 1, for Δ. The algorithm proceeds in rounds. Each process has a preference for the decision value in each round; initially, this preference is the input value of the process. In each round r, processes execute a timing-based consensus algorithm with their current estimate of Δ, using their preferences for this round as inputs.[4] The algorithm guarantees that once processes have the same preference in some round, they will remain in agreement and will eventually decide. The timing-based algorithm used in each round avoids conflicting decisions even if the current estimate for Δ is wrong. If no decision is made in a round, then the processes advance to the next round, using a larger estimate for the time bound Δ. Eventually, processes either decide or end up using the correct estimate, in which case the timing-based algorithm guarantees that they will decide.

The code for the algorithm appears in Figure 1. The algorithm uses the following shared data structures: an infinite array $x[*, 0..1]$ of bits, and an infinite array $y[*]$; the possible values of each $y[i]$ are $\{\perp, 0, 1\}$. The decision value is written to the shared bit *out*. We use only atomic reads and writes to the shared registers. In addition, each process p_i has a local register v_i containing its current preference and a local register r_i containing its current round number. The estimate d_r used in round r is $r!$.

In round r, process p_i first flags its preference v by writing 1 to $x[r, v]$. Then the process checks the lock on this round by reading $y[r]$ and writes its preference to $y[r]$ if $y[r]$ has still its initial value $\perp$. Process p_i then reads the flag for the other preference (denoted by $\bar{v}$). If $x[r, \bar{v}]$ is not set, then every process that reaches round r with the conflicting preference $\bar{v}$ will find $y[r]$ set to v. Consequently, process p_i can safely decide on v, and it writes the decision value to *out*. Otherwise, it waits for d_r (the estimate of Δ for the current round) and then sets its preference for the next round by reading $y[r]$.

Two processes with conflicting preferences for round r will not resolve the conflict only if both of them find $y[r] = \perp$ first and one of them proceeds and chooses its preference for the next round before the other one finishes the assignment to $y[r]$. However, if each process is required to finish the assignment within time Δ, and the value of d_r exceeds Δ, then this cannot happen. Also, notice that if all processes in a round have the same preference, then a decision is reached in that round. These observations, together with the fact that the sequence $d_1, d_2, \ldots$ increases without a bound, ensure termination. The next section includes a complete proof of correctness for this algorithm.

[4] The idea of using preferences for consensus was used previously, e.g., in [AH90].

```
Shared registers: initially: out = ⊥, y[∗] = ⊥, x[∗, ∗] = 0.
Local registers: initially: r_i = 1, v_i = in_i.
Constants: d_r = r! for all r.

1 while out = ⊥ do
2       x[r_i, v_i] := 1;
3       if y[r_i] = ⊥ then y[r_i] := v_i fi;
4       if x[r_i, v̄_i] = 0 then out := v_i
5                          else delay(d_{r_i});
6                               v_i := y[r_i];
7                               r_i := r_i + 1 fi
8 od;
9 decide(out).
```

FIG. 1. *Time-adaptive consensus: the program for process p_i with input in_i.*

3.2. Correctness. We now present the correctness proof of the algorithm. We assume that a process keeps taking idling steps after it has decided. Thus an infinite execution contains infinitely many steps of every nonfaulty process.

LEMMA 3.1. *If process p_i decides on a value v, then $in_j = v$ for some process p_j.*

Proof. If there are two processes that have different inputs, then the lemma holds trivially. Suppose all processes start with the same input in. Consider the following formula ϕ:

$$\forall i.\, v_i \;=\; in \;\wedge\; \forall r.\, y[r] \in \{\bot, in\} \;\wedge\; out \in \{\bot, in\}.$$

Initially, ϕ holds. It is easy to check that each transition of the algorithm preserves ϕ. Thus ϕ is an invariant of the algorithm. The lemma follows immediately. □

Let $r \geq 1$ and $v \in \{0, 1\}$. Formally, a process p_i *reaches* round r if it executes statement 2 (see Figure 1) with $r_i = r$. A process p_i *prefers* the value v in round r if $v_i = v$ when p_i reaches round r. A process p_i *commits* to the value v in round r if it executes the assignment $out := v$ with $r_i = r$.

LEMMA 3.2. *If all processes reaching round r have the same preference v for round r, then all nonfaulty processes reaching round r commit to v in round r.*

Proof. Suppose all processes reaching round r have the same preference v for round r. Thus whenever some process p_i sets the bit $x[r, v_i]$ to 1, v_i equals v. Consequently, $x[r, \bar{v}] = 0$ is an invariant. Now consider a process p reaching round r. Assuming that p continues to take steps in round r, p will find $x[r, \bar{v}]$ unset at statement 4 and commit to the value v. □

LEMMA 3.3. *If some process commits to v in round r, then all processes reaching round $r + 1$ prefer v in round $r + 1$.*

Proof. Suppose some process p commits to v in round r. Since p finds $x[r, \bar{v}]$ unset at statement 4, it follows that every process with preference $\bar{v}$ for round r finds $y[r] \neq \bot$ at statement 3. This implies that for a committed value v, $y[r] \neq \bar{v}$ is an invariant of the program. Since a process decides on its preference for round $r + 1$ by reading $y[r]$, the lemma follows. □

LEMMA 3.4. *No two processes decide on conflicting values.*

Proof. Suppose two processes decide on conflicting values. This means that there

exist nonfaulty processes p_0 and p_1 such that p_0 commits to 0 in round r and p_1 commits to 1 in round r'. We will obtain a contradiction.

First, suppose that $r \neq r'$. Without loss of generality, let $r < r'$. Since p_0 commits to 0 in round r, by Lemma 3.3, all processes reaching round $r+1$ prefer 0 in round $r+1$, and consequently, by Lemma 3.2, if nonfaulty, commit to 0 in round $r+1$. Since p_1 reaches round $r+1$ and is nonfaulty, it follows that p_1 commits to 0 in round $r+1$, a contradiction.

Now suppose that $r = r'$. In round r, process p_0 prefers 0 and process p_1 prefers 1. If process p_0 finds $x[r,1]$ unset at statement 4, then process p_1 must find $x[r,0]$ set at statement 4, and vice versa. Consequently, it is not possible that both commit in round r. □

The proof of termination relies only on the fact that the sequence of delays, $d_1, d_2, \ldots$ is unbounded. The termination is guaranteed by the following lemma.

LEMMA 3.5. *In a Δ-admissible execution, if $d_r \geq \Delta$, then all processes reaching round $r+1$ have the same preference in round $r+1$.*

Proof. Assume $d_r \geq \Delta$. Consider a Δ-admissible execution α and a Δ-admissible time assignment τ for it. Let k be the smallest index such that the event e_k is the assignment $v_i := y[r]$ (at statement 6) by some process p_i that reaches round $r+1$. Let the event e_l correspond to the delay statement $delay(d_r)$ (at statement 5) by process p_i. We know that $\tau_k - \tau_l > d_r$ and hence $\tau_k - \tau_l > \Delta$.

Before it reaches the delay statement, process p_i either finds $y[r] \neq \perp$ or assigns its preference for round r to $y[r]$. Hence in states s_m, for $m \geq l$, $y[r] \neq \perp$. Let $y[r] = v$ in state s_k. We want to prove that $y[r] = v$ in all states s_m for $m \geq k$. Suppose not. Let p_j, $j \neq i$, be a process that writes to $y[r]$ (at statement 3) at step $k' > k$. Let $e_{l'}$ be the event that p_j tests the condition $y[r] = \perp$ (at statement 3). Since $y[r] \neq \perp$ in all states s_m for $m \geq l$, we have $l' < l$. This implies $\tau_{k'} - \tau_{l'} \geq \tau_k - \tau_l > \Delta$. Since l' and k' are consecutive steps of p_j, this contradicts the Δ-admissibility of τ. Thus $y[r] = v$ in all states s_m for $m \geq k$.

Since every process reaching round $r+1$ chooses its preference for round $r+1$ by reading $y[r]$ at some step $m \geq k$, the lemma follows. □

If $d_r \geq \Delta$, then Lemmas 3.5 and 3.2 imply that in a Δ-admissible execution, no process can reach round $r+2$, and every nonfaulty process decides in round $r+1$ or lower. If the sequence $d_1, d_2, \ldots$ is unbounded, then for every Δ, there is some r such that $d_r \geq \Delta$. Consequently, we get termination in each Δ-admissible execution. This implies the following theorem.

THEOREM 3.6 (correctness). *The algorithm in Figure 1 is a correct solution to wait-free consensus in the unknown-delay model.*

3.3. Time complexity. Now let us analyze the time complexity of the algorithm. Recall that the worst-case time complexity of the algorithm is the maximum time after which a nonfaulty process decides.

Formally, given an execution α and a time assignment τ, let $time(\alpha, \tau)$ be the maximum difference $\tau_k - \tau_l$ such that both the events e_l and e_k are nonidling steps of the same process. Recall that a process takes idling steps only after it has decided. The worst-case time complexity of an algorithm A when the upper bound is Δ is then $time_\Delta(A)$ as defined in section 2.

LEMMA 3.7. *Let R be the smallest index such that $d_R \geq \Delta$. Then the worst-case time complexity $time_\Delta(A)$ is at most $9(R+1)\Delta + d_R$.*

Proof. Let $d_R \geq \Delta$ and $d_{R-1} < \Delta$. Consider an execution α and a Δ-admissible time assignment τ. Let p be a process. Since $d_R \geq \Delta$, by Lemmas 3.5 and 3.2, either

p fails or p decides in round $R+1$ or lower. For the worst-case analysis, we assume that p decides in round $R+1$. For each round r, suppose p enters round r at the i_rth step in α.

Let e_k be the last nonidling step of process p. The time taken by p is $\tau_k - \tau_{i_1}$. In each round r, p executes only a constant, at most 8, number of steps, possibly including a delay statement. By Δ-admissibility, we have $\tau_{i_{r+1}} - \tau_{i_r} \leq 8\Delta + d_r$, for any round r. For $r < R$, $d_r < \Delta$. Hence $\tau_{i_{R+1}} - \tau_{i_1} \leq 9R\Delta + d_R$. In round $R+1$, p_i takes only a constant, at most 7, number of steps without executing the delay statement. Hence $\tau_k - \tau_{i_{R+1}} \leq 7\Delta$. Hence $\tau_k - \tau_{i_1} \leq 9(R+1)\Delta + d_R$. The lemma follows. □

The time complexity of the algorithm depends on the choice of the sequence d_r. If the sequence is fast growing, then the value of R will be small. However, if the sequence grows too fast, then the value of d_R can be much larger than Δ itself. For instance, if we let $d_r = r$, then $R = \Delta = d_R$, and the time complexity is $O(\Delta^2)$. For $d_r = 2^r$, $R = O(\log \Delta)$, $d_R \leq 2 \cdot \Delta$, and the time complexity is $O(\Delta \cdot \log \Delta)$. For $d_r = 2^{2^r}$, $R = O(\log \log \Delta)$, but $d_R \leq 2^\Delta$, giving time complexity $O(2^\Delta)$. As we shall see (as part of the lower-bound proof in the next section), the best sequence is $d_r = r!$.

Let fac^{-1} be the inverse of the factorial function, that is, $fac^{-1}(d)$ is the smallest integer r such that $r! \geq d$, for any real $d > 0$; note that $fac^{-1}(d) = \Theta(\frac{\log d}{\log \log d})$. In the case where $d_r = r!$ for all r, $R = fac^{-1}(\Delta)$, and $d_R = (fac^{-1}(\Delta))!$. Hence $d_R \leq \Delta \cdot fac^{-1}(\Delta)$. This gives the overall complexity of $O(\Delta \cdot fac^{-1}(\Delta))$.

THEOREM 3.8 (time complexity). *For the algorithm in Figure* 1 *with the sequence of delays $d_r = r!$, for every Δ, the worst-case time complexity $time_\Delta(A)$ is bounded by $10 \cdot \Delta \cdot (fac^{-1}(\Delta) + 1)$.*

Note that our algorithm uses unbounded space. In a Δ-admissible execution, only the first $(fac^{-1}(\Delta) + 1)$ elements of the arrays x and y are used. Since $fac^{-1}(\Delta)$ is small for any reasonable value of Δ, space is not a real problem. For instance, if our time unit is a second and the upper bound Δ is 1000 years, then $fac^{-1}(\Delta)$ is 14.

Finally, let us consider the contention-free complexity of the algorithm. Informally, we want the contention-free complexity to indicate the time taken by a process when it executes by itself without interference from other processes. Formally, given an execution α and a time assignment τ, let *cf-time*(τ, α) be the maximum difference $\tau_j - \tau_i$ such that both events e_j and e_i are nonidling events of the same process p, and every other process q either has decided before step i or has not taken any step before step j.

The contention-free time complexity of algorithm A for the upper bound Δ is then given by *cf-time*$_\Delta(A)$ and *min-cf-time*$_\Delta(A)$. Our algorithm has low contention-free complexity.

THEOREM 3.9 (fast decision in the absence of contention). *For the algorithm in Figure* 1, *for every Δ, cf-time$_\Delta(A) = 7 \cdot \Delta$, and min-cf-time$_\Delta(A) = 0$.*

Proof. Consider an execution α and indices i and j such that event e_i is the first event of process p, event e_j is the decision event of p, and every other process q either has decided before step i or has not taken any step before step j. There are two cases to consider.

If some process q has decided before process p starts, then the value of *out* is different from $\perp$ in state s_i, and p takes at most two steps before deciding, both of which have zero explicit delay. This implies that if τ is a Δ-admissible assignment for α, then $\tau_j - \tau_i \leq 2\Delta$. Furthermore, since p does not execute any delay statement,

the difference $\tau_j - \tau_i$ can be made as small as possible, implying that the infimum of $\tau_j - \tau_i$ over all Δ-admissible time assignments for α is 0.

If no process q has taken a step before event e_i, then p is the first process to start, and no process starts before p decides. In this case, p decides in the first round after taking at most seven steps, again without executing any delay statement. In this case, the maximum of $\tau_j - \tau_i$ over all Δ-admissible time assignments for α is 7Δ, and the infimum is 0. □

We point out that if some processes fail without deciding before a process p starts, then even if p runs by itself, it may execute for $O(\Delta \cdot fac^{-1}(\Delta))$ time. Thus failures of processes can lead to the worst-case time complexity.

3.4. Lower bound on time complexity. For the algorithm of section 3, if Δ is the upper bound on step time, then a process decides within time $O(\Delta \cdot fac^{-1}(\Delta))$. If a process knew the value of Δ in advance, then it can execute the algorithm with $d_1 = \Delta$, ensuring termination in the second round, within time $O(\Delta)$. Thus the lack of knowledge of the value of Δ multiplies the time complexity by a factor of $fac^{-1}(\Delta)$. In this section, we prove this increase in cost to be inherent: we prove that any algorithm for solving two-process consensus in the unknown-delay model has worst-case time complexity of $O(\Delta \cdot fac^{-1}(\Delta))$.

To prove the lower bound, we restrict our attention to a system with two processes, p_1 and p_2. Let A be an algorithm for wait-free consensus in the unknown-delay model. Consider an execution $\alpha = s_0 \stackrel{e_0}{\to} s_1 \stackrel{e_1}{\to} \cdots$. The execution can be partitioned into *blocks*, each containing a sequence of events by the same process. Formally, let $b_0, b_1, \ldots$ be an increasing sequence of integers with $b_0 = 0$ such that for each i, all of the steps indexed from b_i to $b_{i+1} - 1$ are of the same process, and the step indexed b_{i+1} is of a different process. Thus the ith block is the execution fragment $s_{b_i} \stackrel{e_{b_i}}{\to} s_{b_i+1} \cdots \stackrel{e_{b_{i+1}-1}}{\to} s_{b_{i+1}}$ consisting of steps of a single process.

By Lemma 2.1, A must also guarantee safety in the asynchronous shared-memory model. Therefore, from the proof of the impossibility of solving consensus in the shared-memory asynchronous model (e.g., [LA87, Theorem 4.1]), we can deduce the following.

LEMMA 3.10. *There exists an infinite sequence of executions $\alpha_0, \alpha_1, \ldots$ such that for all $k \geq 0$, (1) α_k is a finite execution with $k+1$ blocks, (2) α_{k+1} is an extension of α_k, and (3) no process has decided at the end of α_k.*

Let us recall the definition of the time complexity of consensus. Given an execution α of A and a Δ-admissible time assignment τ for α, whenever e_i and e_j are the (nonidling) steps of the same process, we have $time_\Delta(A) \geq \tau_j - \tau_i$. Now we consider another definition needed for the proof. Given an execution α, define d_i to be the total sum of the delays in delay statements appearing in the ith block. With each execution we can associate a sequence $d_0, d_1, \ldots$, called the *sequence of block delays*.

LEMMA 3.11. *Let α be a finite execution with $k+1$ blocks with $k \geq 1$ such that no process has decided at the end of α. Let $d_0, d_1, \ldots, d_k$ be the associated sequence of block delays, and let $\Delta \geq 1 + \sum_{i=0}^{k-1} d_i$. Then $time_{2\Delta}(A) \geq k \cdot \Delta + d_k$.*

Proof. Let α be $s_0 \stackrel{e_0}{\to} \cdots s_{n-1} \stackrel{e_n}{\to} s_n$ consisting of $k+1$ blocks starting at indices $b_0, \ldots, b_k$. Without loss of generality, assume that the last block corresponds to steps of process p_1.

Since no process has decided and A satisfies wait freedom, we know that p_1 can take an additional step, e_{n+1}; let α' denote the extended execution. Now we construct a time assignment τ for α' as follows. Let $\Delta \geq 1 + \sum_{i=0}^{k-1} d_i$.

We want each block, except the last one, to take Δ time. For every $0 \le j \le n$,

- if e_j is the first step of the ith block (i.e., $j = b_i$), for $i = 0, \ldots, k$, then let τ_j be $i \cdot \Delta$; otherwise,
- if e_j is the last step of the ith block (i.e., $j = b_{i+1} - 1$), for $i = 0, \ldots, (k-1)$, then let τ_j be $(i+1) \cdot \Delta$; otherwise,
- let τ_j be $\tau_{j-1} + d(e_{j-1}) + 1/n$.

For each block i, the sum of the explicit delays of events in the block i is d_i, and the number of events in the block i is bounded by n. By the choice of Δ, it is clear that the sequence of values defined above is nondecreasing. Furthermore, let the time τ_{n+1} of the last step be $\tau_n + d(e_n) + \Delta$.

Now consider events e_j and $e_{j'}$ that are consecutive steps of the same process. There are two cases to consider.

1. Both e_j and $e_{j'}$ belong to the same block (i.e., $j' = j + 1$). Then $\tau_{j'} - \tau_j > d(e_j)$. In each block, except possibly the last one, the difference between the time of the last step and the first step is Δ. In the last block, the time of the first step is $k \cdot \Delta$, and the times of the remaining steps are increased only when p_1 executes delay statements. Hence $\tau_{j'} - \tau_j \le \Delta + d(e_j)$.

2. The event e_j is the last event of a block, say the ith block, and $e_{j'}$ is the first event of the $(i+2)$th block. In this case, $\tau_{j'} = (i+2) \cdot \Delta$. If the ith block has only one event, then $\tau_j = i \cdot \Delta$; otherwise, $\tau_j = (i+1) \cdot \Delta$. Thus $\Delta \le \tau_{j'} - \tau_j \le 2\Delta$.

This implies that τ is a legal time assignment for α, and furthermore, it is 2Δ-admissible (i.e., we can choose 2Δ as the upper bound). Note that the total delay in the last block may be larger than 2Δ. Since only p_1 takes steps in the last block, this means that p_2 has failed if we put an upper bound of 2Δ on the step times.

Finally, observe that the time of the last step τ_{n+1} is $(k+1) \cdot \Delta + d_k$. The time of the first step of p_1 is either 0 or Δ depending on whether the first or the second block corresponds to steps of p_1. This means that p_1 has executed for at least $k \cdot \Delta + d_k$ time without deciding. Hence $time_{2\Delta}(A) \ge k \cdot \Delta + d_k$. □

LEMMA 3.12. *There exists an infinite nondecreasing sequence of values $\Delta_1, \Delta_2, \ldots$ such that for all $k \ge 1$, for all $\Delta \ge \Delta_k$, $2time_\Delta(A) \ge k \cdot \Delta + (\Delta_{k+1} - \Delta_k)$.*

Proof. Consider the infinite sequence of executions $\alpha_0, \alpha_1, \ldots$ of Lemma 3.10. Note that the lemma implies that α_{k+1} is obtained from α_k by adding one block. Let d_k be the delay of the last block of α_k. Let $\Delta_k = 2(1 + \sum_{i=0}^{k-1} d_i)$. For each $k \ge 1$, by applying Lemma 3.11 to α_k, we get that for all $\Delta \ge \Delta_k$, $time_\Delta(A) \ge k \cdot \Delta/2 + d_k$. The lemma follows since $d_k = (\Delta_{k+1} - \Delta_k)/2$. □

To complete the proof of the lower bound, we present the following technical lemma.

LEMMA 3.13. *For any nondecreasing sequence $\Delta_1, \Delta_2, \ldots$ of real numbers, for infinitely many indices k, $k \cdot \Delta_k + (\Delta_{k+1} - \Delta_k) \ge \Delta_k \cdot fac^{-1}(\Delta_k)$.*

Proof. The proof is by contradiction. Suppose there exists a nondecreasing sequence $\Delta_1, \Delta_2, \ldots$ and an index i such that for all $k \ge i$, $k \cdot \Delta_k + (\Delta_{k+1} - \Delta_k) < \Delta_k \cdot fac^{-1}(\Delta_k)$. Define $m_k = fac^{-1}(\Delta_k)$. By definition, for all k,

$$(m_k - 1)! < \Delta_k \le m_k!.$$

The proof follows the following steps:

(1) For all $k \ge i$, we have $k \cdot \Delta_k + (\Delta_{k+1} - \Delta_k) < \Delta_k \cdot m_k$. Also, $\Delta_{k+1} - \Delta_k \ge 0$ for all k. Hence $m_k > k$ for all $k \ge i$.

(2) Let $k \ge i$. We have

$$\Delta_{k+1} < (m_k - k + 1) \cdot \Delta_k \le (m_k - k + 1) \cdot m_k! < (m_k + 1)!.$$

Hence $m_{k+1} \leq m_k + 1$. Thus the sequence of values $(m_k - k)$, for $k \geq i$, is nonincreasing. From (1), $(m_k - k)$ is positive for all $k \geq i$. That is, $(m_k - k)$, $k \geq i$, forms a nonincreasing infinite sequence of positive numbers. Hence there exists a positive integer a and an index j such that $m_k - k = a$ for all $k \geq j$.

(3) Choose $k \geq j$ such that $k > (a+1)^2$. We have $\Delta_{k+1} < (a+1) \cdot \Delta_k$, and $\Delta_{k+2} < (a+1) \cdot \Delta_{k+1}$. Hence

$$\Delta_{k+2} < (a+1)^2 \cdot \Delta_k \leq (a+1)^2 \cdot m_k! < k \cdot (a+k)! < (a+k+1)!.$$

This implies $m_{k+2} \leq a + k + 1$. This contradicts the assertion $m_{k+2} = a + k + 2$ of (2), which completes the proof. □

Lemma 3.13, together with Lemma 3.12, implies the following.

THEOREM 3.14 (lower bound on time complexity). *For any algorithm A for solving two-process wait-free consensus in the unknown-delay model, for every real number d, there exists $\Delta > d$ such that the worst-case time complexity $time_\Delta(A)$ is at least $(\Delta \cdot fac^{-1}(\Delta))/2$.*

Proof. Lemma 3.12 gives a nondecreasing sequence $\Delta_1, \Delta_2, \ldots$ such that for all $k \geq 1$, for all $\Delta \geq \Delta_k$, $2time_\Delta(A) \geq k \cdot \Delta + (\Delta_{k+1} - \Delta_k)$. There are two cases to consider:

1. The sequence $\Delta_1, \Delta_2 \ldots$ is unbounded: For every real value d, there is an index i such that for all $k \geq i$, $\Delta_k > d$. Using Lemma 3.13, there is an index $k \geq i$ such that $\Delta_k > d$ and $k \cdot \Delta_k + (\Delta_{k+1} - \Delta_k) \geq \Delta_k \cdot fac^{-1}(\Delta_k)$, and hence $2time_{\Delta_k}(A) \geq \Delta_k \cdot fac^{-1}(\Delta_k)$.

2. The sequence $\Delta_1, \Delta_2 \ldots$ is bounded: There is a value d^* such that $\Delta_k < d^*$ for all k. Let $\Delta \geq d^*$. Then by Lemma 3.12, for all k, $2time_\Delta(A) \geq k \cdot \Delta$. Choosing $k > fac^{-1}(\Delta)$ gives $2time_\Delta(A) > \Delta \cdot fac^{-1}(\Delta)$. □

Notice that our lower bound implies that for every algorithm, there is an unbounded sequence of values Δ for which the worst-case time complexity is at least $(\Delta \cdot fac^{-1}(\Delta))/2$. This does not rule out the existence of a (different) unbounded sequence of values Δ for which the worst-case time complexity is less than $(\Delta \cdot fac^{-1}(\Delta))/2$. For the algorithm of Figure 1, if we choose the sequence $d_r = 2^{2^r}$, then setting $\Delta = d_r$ implies termination in $O(\log \log r)$ rounds, giving the worst-case time complexity of $O(\Delta \cdot \log \log \Delta)$; however, if we set $\Delta = d_r + 1$, the worst-case time complexity is $O(2^\Delta)$.

4. Time-adaptive mutual exclusion. The mutual exclusion problem is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. A solution to the problem should satisfy the following two properties:

- *Mutual exclusion:* No two processes are in their critical section at the same time.
- *Deadlock freedom:* If some process p starts executing its algorithm, then eventually some process (possibly different from p) is in its critical section.

We assume that each of the potentially n contending processes has a unique identifier taken from the set $\{1, \ldots, n\}$. While deadlock freedom is essential, starvation freedom—any process that is trying to enter its critical section eventually does enter its critical section—is less important in systems where contention is rare. When contention is rare, it is important to design algorithms satisfying the fast-access property:

- *Fast access:* In the absence of contention, a process executes only a constant number of steps in order to enter its critical section and only a constant number of steps in order to execute the exit code.

In [Lam87], Lamport presented a mutual exclusion algorithm that satisfies the fast-access property. However, in the presence of contention, however small, the winning process may have to check the status of all of the other n processes before it is allowed to enter its critical section. Alur and Taubenfeld overcome this limitation in the known-delay model [AT96] (see also [AT93]). Their algorithm satisfies the fast access property, and furthermore, in the presence of contention, a process does not have to check the status of all of the other n processes before it can enter its critical section, but it may need to delay itself for $2 \cdot \Delta$ time units. In the next section, we describe an algorithm with similar properties for the unknown-delay model. Other algorithms which satisfy the fast-access property can be found in [CS93, MS93, Sty92, YA93].

4.1. The algorithm. We now present a fast mutual exclusion algorithm for the unknown-delay model. Since in this model a time bound on the speed exists but is not known, the processes keep an estimate of this time (stored in a shared register) and update it when it is noticed that the estimate is not accurate. An entry to the critical section which involves an update (of the estimate) is going to be much slower than an entry without an update. However, the algorithm has the property that at most Δ updates are necessary. As we show, the algorithm is also time efficient when there is contention. As is usually assumed when designing a mutual exclusion algorithm, we also assume that process failures do not occur.

The precise code for the algorithm is given in Figure 2. Notice that the statement **await** *condition* is an abbreviation for **while** $\neg$*condition* **do** *skip* (and hence may involve many accesses to the shared memory). The algorithm is composed of two basic algorithms. The first is Alur and Taubenfeld's algorithm (abbreviated AT) for fast mutual exclusion using a timing assumption [AT96]. Statements 1–10 are the entry code of AT and statements 29–31 are its exit code. We point out that in the original AT, the register *bound* is initially set to Δ (which is assumed to be known). Furthermore, while AT satisfies mutual exclusion only when $bound \geq \Delta$, the proof of deadlock freedom does not depend on the value of *bound*. We will exploit this property in our construction. The critical section of AT is now replaced by Lamport's fast mutual exclusion algorithm, statements 11–28. These two algorithms are combined along with a mechanism for estimating and updating the current bound. All references to the register *update* and the array *trying* belong to this mechanism and are not part of the original AT and Lamport's algorithms.

Intuitively, the algorithm works as follows. First, each process executes AT, using the current estimate *bound*. If the estimate is correct, or if there is no contention, only one process will proceed to the next stage (i.e., get to statement 11). However, it is possible that the current estimate used by the processes is incorrect. In this case, more than one process may proceed to the next stage, and therefore, to guarantee mutual exclusion, we embed Lamport's fast algorithm at this point. If a process discovers contention while executing Lamport algorithm, it "knows" that the current estimate used is incorrect and has to be increased. Contention is discovered when either of the conditions in statements 14 and 18 evaluates *true*.

To avoid complications, only a process that enters its critical section (statement 23) is allowed to update the register *bound*. This guarantees that no two processes try to update the estimate at the same time, and thus the value of *bound* never decreases.

The update is done as follows. First, the process sets *update* to 1, signaling that it wants to make an update (statement 22). Then it waits until each active process returns to the beginning of its trying code and waits for *update* to become 0 (statement 2). It is easy to check that once *update* is 1, eventually every process will

```
Initially: y = 0, yy = 0, z = 0, bound = 1, update = 0, trying[i] = 0, and b[i] = 0 for all i.

1   start1: repeat
2             if update = 1 then trying[i] := 0; await update = 0 fi;
3             trying[i] := 1;
4             x := i;
5           until (y = 0);
6           y := i;
7           if x ≠ i then delay(2 · bound);
8                         if y ≠ i then goto start1 fi;
9                         await (z = 0) or (update = 1)
10          else z := 1;
11  start2: if update = 1 then goto start1 fi;
12          b[i] := 1;
13          xx := i;
14          if yy ≠ 0 then b[i] := 0;
15                        await (yy = 0) or (update = 1);
16                        goto start2 fi;
17          yy := i;
18          if xx ≠ i then b[i] := 0;
19                        for j := 1 to n do await (b[j] = 0) or (update = 1) od;
20                        if yy ≠ i then await (yy = 0) or (update = 1);
21                                       goto start2
22                        else update := 1 fi fi;                    (* set lock *)
23          critical section;
24          trying[i] := 0;
25          if update = 1 then for j := 1 to n do await trying[j] = 0 od;     (* wait *)
26                        bound := bound + 1 fi;                     (* increment bound *)
27          yy := 0;
28          b[i] := 0;
29          z := 0;
30          if update = 1 then y := 0; update := 0
31                        else if  y = i then y := 0 fi fi
```

FIG. 2. *Fast time-adaptive algorithm: process i's program.*

test it. Once process i finds that *update* is 1, it returns to the beginning of its code, signals to the updating process that it is at the beginning by setting $trying[i]$ to 0, and waits (statement 2). Once the updating process gets acknowledgments from all active processes, it safely increments *bound* (statement 26), executes the exit code of both the algorithms, and releases the lock (statement 30), which leaves the system in its initial configuration (except for the value of *bound*).

The fact that the processes return to the beginning of their code before *bound* is incremented guarantees that the value of the *bound* will never be greater than Δ.

Once *bound* equals Δ, the entry code of AT (statements 1–10) ensures that no two processes execute Lamport's algorithm (statements 11–28) at the same time. Hence from that point on, processes will always enter their critical section along the fast path of Lamport's algorithm.

In summary, each process starts by checking if an update of *bound* is taking place, in which case it waits until the update is finished. Then the process performs the entry code for the timing-based mutual exclusion algorithm using the current estimate. If it gains access to the critical section (of AT), the process executes Lamport's fast mutual exclusion algorithm. However, in the algorithm, if a process enters its critical section via the slow path, it "knows" that the current estimate in use is incorrect and should be increased. It does so by first signaling other processes to go to the beginning of

their code and, after they all do so, incrementing the register *bound*.

4.2. Correctness. The design of the algorithm and its correctness proof are based on the following straightforward general observation.

LEMMA 4.1. *Let A and B be mutual exclusion algorithms (with disjoint sets of shared registers), and let C be the algorithm obtained by replacing the critical section of A with algorithm B.*[5]

1. *If both A and B are deadlock-free, then C is deadlock-free.*
2. *If either A or B satisfies mutual exclusion, then C satisfies mutual exclusion.*

Proof. The entry code of C is composed of the entry code of A, denoted by C_A, followed by that of B, denoted by C_B. Assume that both A and B are deadlock-free. If some process starts executing algorithm C, then since A is deadlock-free, eventually some process will finish C_A and proceed to C_B. Since B is deadlock-free, eventually some process will finish C_B and enter its critical section. Thus C is deadlock-free.

If A satisfies mutual exclusion, then no two processes can be at their C_B code at the same time. If B satisfies mutual exclusion, then no two processes can finish their C_B code at the same time. In either case, it implies that no two processes are in their critical section at the same time. □

The correctness of the algorithm is based on Lemma 4.1 and the properties of Lamport's algorithm and AT. Note that the algorithm also satisfies the correctness requirements in the asynchronous model.

As already explained, the algorithm is obtained by replacing the critical section of Alur and Taubenfeld's algorithm with Lamport's algorithm (statements 11–28). These two algorithms are combined along with a mechanism for estimating and updating the current time bound. All references to the register *update* and the array *trying* belong to this mechanism and are not part of the original AT and Lamport's algorithm.

It is known that Lamport's algorithm satisfies mutual exclusion and deadlock freedom and that AT satisfies deadlock freedom regardless of the value of *bound*.

THEOREM 4.2. *The algorithm in Figure* 2 *satisfies deadlock freedom in the asynchronous model.*

Proof. As long as the value of *update* is 0, executing statements 1–10 or 11–22 is the same as executing the entry code of AT or the entry code of Lamport's algorithm, respectively. Since both AT and Lamport's algorithm are deadlock-free (regardless of the value of *bound*), Lemma 4.1 implies that the algorithm cannot be deadlocked while the value of *update* is continuously 0.

We observe that if the value of *update* is 1, then there must be some process (called the *winner*) which is either in its critical section or in its exit code. Once *update* is 1, eventually every process (other than the winner) will test it. Once process i finds that *update* is 1, it returns to the beginning of its code and signals to the winner that it is at the beginning by setting $trying[i]$ to 0. Thus eventually the winner gets acknowledgments from all active processes, which implies that the winner cannot be blocked forever in the *for* loop of statement 25, and will eventually set *update* back to 0. This implies that the system cannot be deadlocked while the value of *update* is continuously 1.

Thus a deadlock can occur in an infinite execution only if *update* changes values an infinite number of times. However, each time *update* changes its value, some process

[5] If the critical section of A has a label, then in C this label is associated with the first statement of B.

either enters or exits its critical section. Therefore, in an execution where *update* changes values an infinite number of times, no deadlock can occur. □

THEOREM 4.3. *The algorithm in Figure* 2 *satisfies mutual exclusion in the asynchronous model.*

Proof. As long as the value of *update* is 0, executing statements 11–28 is the same as executing Lamport's algorithm. Since Lamport's algorithm satisfies mutual exclusion, by Lemma 4.1, the new algorithm must also satisfy mutual exclusion when the value of *update* is 0.

If the value of *update* is 1, then there must be some process (the *winner*) which is either in its critical section or in its exit code.

If some process has tested *update* before it was set to 1, then the value of *update* was 0 at this time, and (as already explained above) since Lamport's algorithm satisfies mutual exclusion, this process will not enter its critical section, and eventually it will have to test *update* again.

Once the winner sets *update* to 1, no other process can enter the critical section until *update* is set back to 0. To see this, observe that once *update* is 1, eventually every process will test it, and once a process finds that *update* is 1, it returns to the beginning of its code, signals to the updating process that it is at the beginning by setting *trying*[*i*] to 0, and waits (statement 2) until *update* is set to 0. The winner sets *update* to 0 in its exit code only after it gets acknowledgments from all active processes. Because the processes retreat to *start*1 at the beginning of their code before *update* is reset, no process executes Lamport's (embedded) algorithm, which in turn guarantees that no two processes can enter their critical section as long as the value of *update* is not changed. □

4.3. Time complexity. Next, we show that the register *bound* is updated at most Δ times.

LEMMA 4.4. *bound* $\leq \Delta$ *is an invariant of the algorithm.*

Proof. Once *bound* reaches the correct Δ, the delay in statement 7 is $2 \cdot \Delta$. At this point, all the processes that participate in the algorithm, except the one that is updating the value of *bound*, are at the beginning of their code, waiting for *update* to become 0. Thus from that point on, this code (statements 1–10) behaves exactly like the original AT.

This means that from now on, only one process can be in Lamport's algorithm (statements 11–28). When only one process is in Lamport's algorithm, the test in line 18 is always evaluated to false, and hence statement 22 will not be reached, the value of *update* will remain at 0, and no more updates to *bound* will occur. (Statement 22 is the only place where the value of *update* is changed from 0 to 1.) Thus the number of times a winning process has to update *bound* after executing its critical section is bounded by Δ. □

The next theorem shows that the algorithm is time efficient. In the theorem, the time it takes for a process to enter its critical section is measured from the last time some process exited its critical section.

THEOREM 4.5. *The algorithm has the following properties:*

1. *Fast access: In the absence of contention, a process executes only a constant number of steps* (13) *from the location start*1 *to its critical section and only a constant number of steps* (8) *to execute the exit code. No delays are necessary.*

2. *In the presence of contention, a winning process which does not update the register bound executes a constant number of steps* (14) *and may need to delay itself for at most* $2 \cdot \Delta$ *time units before entering its critical section.*

3. *In the presence of contention, a winning process which needs to update the register bound may execute $O(n)$ steps and may need to delay itself for at most $2\cdot\Delta$ time units before entering its critical section. This may happen at most Δ times.*

Proof. The first part is straightforward.

By Lemma 4.4, *bound* $\leq \Delta$. Thus executing the delay in statement 7 takes at most $2\cdot\Delta$ time units. Note that a winning process updates *bound* if and only if it finds the condition in statement 18 (i.e., $xx \neq i$) to be true. The second part of the theorem is easily verified by counting steps in the algorithm.

Only when a process finds the condition in statement 18 to be true does it execute the *for* statement at statement 19, in which it may need to execute $O(n)$ steps. This implies the third part of the theorem and explains why the term $O(n)$ is added. □

Observe that there is a tradeoff between the number of updates of the register *bound* and its maximum value. For example, if instead of incrementing it by 1, we double its value when it is updated, then we can show that *bound* is updated at most $\log\Delta$ times and *bound* $\leq 2\cdot\Delta - 1$. Incrementing by 1 is the best strategy since it gives the best amortized time complexity when the number of entries to the critical section is much bigger than Δ.

Notice that our algorithm uses $2n + 5$ shared registers. It is possible to replace the arrays b and *trying*, each of n bits, with one array of n 3-valued registers. Lynch and Shavit proved that when the timing bounds are not known, n is a lower bound on the number of shared registers [LS92]. (The model they used for their algorithm design is the known-delay model, but their lower bound proof continues to hold for the unknown-delay model as well.) In contrast, the algorithms for mutual exclusion in the known-delay model use only a constant number of registers [Lam87, AT96, LS92].

In our algorithm, the value of the register *bound* can only be increased, and after it is updated Δ times, it will reach its maximum value Δ. In a dynamic system where processes are created and destroyed, the upper bound on the speed of the processes may change over time. Our algorithm adapts to an increase in Δ (that may be caused by adding slow processes). However, when Δ decreases (a slow process is destroyed), the value of *bound* may be too high, leading to inefficient utilization. This may be resolved by periodically resetting *bound* to zero and letting it adjust to reflect the current speed.

5. Discussion. We have defined the unknown-delay model, which formalizes systems in which there is an upper bound on memory access time, but this bound is not known. For the consensus problem, we have shown that $\Theta(\Delta\cdot fac^{-1}(\Delta))$ is an upper and lower bound on the time complexity of any algorithm, where fac^{-1} is the inverse of the factorial function. The algorithm that achieves this bound is fast in the absence of contention. Since consensus is universal [He91], our results imply that atomic reads and writes are universal in the unknown-delay model. For the mutual exclusion problem, we have presented an algorithm in which, in the presence of contention, a process needs only delay itself for $2\cdot\Delta$ time units before entering the critical section, when no update of the time estimate is needed. This algorithm is also fast in the absence of contention.

The standard definitions of the consensus problem and the mutual exclusion problem differ in two ways. First, a mutual exclusion algorithm is invoked repeatedly, while a consensus algorithm is invoked only once. Second, in the consensus problem, processes may fail, while in the mutual exclusion problem, it is assumed that processes do not fail. Note however, that our algorithms for both problems are constructed in

a similar manner, by combining an asynchronous algorithm that guarantees safety, a timing-based algorithm that converges when used with a correct estimate for Δ, and a mechanism for estimating Δ.

Acknowledgments. We wish to thank Yehuda Afek, Eli Gafni, Eyal Kushilevitz, and Michael Merritt for helpful discussions and the anonymous referees for a thorough review of the manuscript.

REFERENCES

[AT96] R. Alur and G. Taubenfeld, *Fast timing-based algorithms*, Distrib. Comput., 10 (1996), pp. 1–10.

[AT93] R. Alur and G. Taubenfeld, *How to share an object: A fast timing-based solution*, in Proc. 5th IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 470–477.
remove

[AT94] R. Alur and G. Taubenfeld, *Contention-free complexity of shared memory algorithms*, manuscript, 1994.

[AH90] J. Aspnes and M. Herlihy, *Fast randomized consensus using shared memory*, J. Algorithms, 11 (1990), pp. 441–461.

[ADLS91] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer, *Bounds on the time to reach agreement in the presence of timing uncertainty*, J. Assoc. Comput. Mach., 41 (1994), pp. 122–152.

[CS93] M. Choy and A. Singh, *Adaptive solution to the mutual exclusion problem*, in Proc. 12th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1993, pp. 183–194.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. Assoc. Comput. Mach., 34 (1987), pp. 77–97.

[Dij65] E. W. Dijkstra, *Solution of a problem in concurrent programming control*, Comm. Assoc. Comput. Mach., 8 (1965), p. 569.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer, *Consensus in the presence of partial synchrony*, J. Assoc. Comput. Mach., 35 (1988), pp. 288–323.

[FLP85] M. Fischer, N. Lynch, and M. Paterson, *Impossibility of distributed consensus with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374–382.

[He91] M. Herlihy, *Wait-free synchronization*, ACM Trans. Programming Lang. Systems, 11 (1991), pp. 124–149.

[HK89] A. Herzberg and S. Kutten, *Efficient detection of message forwarding faults*, in Proc. 8th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1989, pp. 339–353.

[LA87] M. Loui and H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, Adv. Comput. Res., 4 (1987), pp. 163–183.

[Lam87] L. Lamport, *A fast mutual exclusion algorithm*, ACM Trans. Comput. Systems, 5 (1987), pp. 1–11.

[LS92] N. Lynch and N. Shavit, *Timing-based mutual exclusion*, in Proc. 13th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 2–11.

[MS93] M. M. Michael and M. Scott, *Fast mutual exclusion, even with contention*, Technical Report 460, Department of Computer Science, University of Rochester, Rochester, NY, 1993.

[PSL80] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228–234.

[Sty92] E. Styer, *Improved fast mutual exclusion*, in Proc. 11th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1992, pp. 159–168.

[YA93] J-H. Yang and J. H. Anderson, *Fast, scalable synchronization with minimal hardware support*, in Proc. 12th ACM Symposium on Principles of Distributed Computing, ACM, New York, 1993, pp. 171–182.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 557–567, April 1997

011

A FIRST-ORDER ISOMORPHISM THEOREM*

ERIC ALLENDER†, JOSÉ BALCÁZAR‡, AND NEIL IMMERMAN§

Abstract. We show that for most complexity classes of interest, all sets complete under first-order projections (fops) are isomorphic under first-order isomorphisms. That is, a very restricted version of the Berman–Hartmanis conjecture holds. Since "natural" complete problems seem to stay complete via fops, this indicates that up to first-order isomorphism there is only one "natural" complete problem for each "nice" complexity class.

Key words. complexity classes, descriptive complexity, reduction, first-order projection

AMS subject classifications. 68Q15, 03D15

PII. S0097539794270236

1. Introduction.

In 1977, Berman and Hartmanis noticed that all NP-complete sets that they knew of were polynomial-time isomorphic [BH77]. They made their now-famous isomorphism conjecture, namely that all NP-complete sets are polynomial-time isomorphic. This conjecture has engendered a large amount of work (cf. [KMR90, You] for surveys).

The isomorphism conjecture was made using the notion of NP-completeness via polynomial-time many–one reductions because that was the standard definition at the time. In [Coo], Cook proved that the Boolean satisfiability problem (SAT) is NP-complete via polynomial-time Turing reductions. Over the years SAT has been shown to be complete via weaker and weaker reductions, e.g., polynomial-time many–one [Kar], logspace many–one [Jon], one-way logspace many–one [HIM], and first-order projections (fops) [Dah]. These last reductions, defined in section 3, are provably weaker than logspace reductions. It has been observed that *natural* complete problems for various complexity classes including NC^1, L, NL, P, NP, and PSPACE remain complete via fops; cf. [I87, IL, SV, Ste, MI].

On the other hand, Joseph and Young, [JY] have pointed out that polynomial-time many–one reductions may be so powerful that they allow *unnatural* NP-complete sets. Most researchers now believe that the isomorphism conjecture as originally stated by Berman and Hartmanis is false.[1]

We feel that the choice of polynomial-time many–one reductions in the statement of the isomorphism conjecture was made in part for historical rather than purely scientific reasons. To elaborate on this claim, note that the class NP arises naturally

* Received by the editors June 20, 1994; accepted for publication (in revised form) June 5, 1995. A preliminary version of this paper appeared in *Proc. 10th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Comput. Sci. 665, Springer-Verlag, Berlin, 1993, pp. 163–174.
http://www.siam.org/journals/sicomp/26-2/27023.html

† Department of Computer Science, Rutgers University, New Brunswick, NJ 08903 (allender@cs.rutgers.edu). The research of this author was supported in part by National Science Foundation grant CCR-9204874. Some of this work was done while this author was on leave at Princeton University.

‡ Departamento L.S.I., Universitat Politècnica de Catalunya, Pau Gargallo 5, E-08071 Barcelona, Spain (balqui@lsi.upc.es). The research of this author was supported in part by EC BRA ESPRIT-II project 3075 (ALCOM) and Acción Integrada Hispano-Alemana 131 B.

§ Computer Science Department, University of Massachusetts, Amherst, MA 01003 (immerman@cs.umass.edu). The research of this author was supported by NSF grant CCR-9207797.

[1] One way of quantifying this observation is that since Joseph and Young produced their unnatural NP-complete sets, Hartmanis has been referring to the isomorphism conjecture as the "Berman" conjecture.

in the study of logic and can be defined entirely in terms of logic, without any mention of computation [Fa]. Thus it is natural to have a notion of NP-completeness that is formulated entirely in terms of logic. On another front, Valiant [Val] noticed that reducibility can be formulated in algebra using the natural notion of a projection, again with no mention of computation. The sets that are complete under fops are complete in *all* of these different ways of formulating the notion of NP-completeness.

Since natural complete problems turn out to be complete via very low-level reductions such as fops, it is natural to modify the isomorphism conjecture to consider NP-complete reductions via fops. Motivating this in another way, one could propose as a slightly more general form of the isomorphism conjecture the following question: Is completeness a sufficient structural condition for isomorphism? Our work answers this question by presenting a notion of completeness for which the answer is yes. Namely, for every nice complexity class including P, NP, etc., any two sets complete via fops are not only polynomial-time isomorphic but first-order isomorphic.

There are additional reasons to be interested in first-order computation. It was shown in [BIS] that first-order computation corresponds exactly to computation by uniform AC^0 circuits under a natural notion of uniformity. Although it is known that AC^0 is properly contained in NP, knowing that a set A is complete for NP under polynomial-time (or logspace) reductions does not currently allow us to conclude that A is not in AC^0; however, knowing that A is complete for NP under first-order reductions does allow us to make that conclusion.

First-order reducibility is a uniform version of the constant-depth reducibility studied in [FSS, CSV]; sometimes this uniformity is important. For a concrete example where first-order reducibility is used to provide a circuit lower bound, see [AG92].

Preliminary results and background on isomorphisms follow in section 2. Definitions and background on descriptive complexity are found in section 3. The main result is stated and proved in section 4, and then we conclude with some related results and remarks about the structure of NP under first-order reducibilities.

2. Short history of the isomorphism conjecture. The isomorphism conjecture is analogous to Myhill's Theorem that all recursively enumerable (r.e.) complete sets are recursively isomorphic, [Myh]. In this section, we summarize some of the relevant background material. In the following, FP is the set of functions computable in polynomial time.

DEFINITION 2.1. *For $A, B \subseteq \Sigma^*$, we say that A and B are* p-isomorphic ($A \stackrel{\mathrm{p}}{\cong} B$) *iff there exists a bijection $f \in \mathrm{FP}$ with inverse $f^{-1} \in \mathrm{FP}$ such that A is many–one reducible to B ($A \leq_{\mathrm{m}} B$) via f (and therefore $B \leq_{\mathrm{m}} A$ via f^{-1}).*

OBSERVATION 2.2 ([BH77]). *All the* NP*-complete sets in* [GJ] *are* p*-isomorphic.*

How did Berman and Hartmanis make their observation? They did it by proving a polynomial-time version of the Schröder–Bernstein theorem. Recall the following.

THEOREM 2.3 ([Kel, Theorem 20]). *Let A and B be any two sets. Suppose that there are* 1:1 *maps from A to B and from B to A. Then there is a* 1:1 *and onto map from A to B.*

Proof. Let $f : A \to B$ and $g : B \to A$ be the given 1:1 maps. For simplicity, assume that A and B are disjoint. For $a, c \in A \cup B$, we say that c is an *ancestor* of a iff we can reach a from c by a finite (nonzero) number of applications of the functions f and/or g. Now we can define a bijection $h : A \to B$ which applies either f or g^{-1}

according to whether a point has an odd number of ancestors or not:

$$h(a) = \begin{cases} g^{-1}(a) & \text{if } a \text{ has an odd number of ancestors,} \\ f(a) & \text{if } a \text{ has an even or infinite number of ancestors.} \end{cases} \quad \square$$

The feasible version of the Schröder–Bernstein theorem is as follows.

THEOREM 2.4 ([BH77]). *Let $f : A \leq_{\mathrm{m}} B$ and $g : B \leq_{\mathrm{m}} A$, where f and g are* 1:1*, length-increasing functions. Assume that $f, f^{-1}, g, g^{-1} \in \mathrm{FP}$, where f^{-1} and g^{-1} are the inverses of f and g. Then $A \overset{\mathrm{p}}{\cong} B$.*

Proof. Let the ancestor chain of a string w be the path from w to w's parent, to w's grandparent, and so on. Ancestor chains are at most linear in length because f and g are length-increasing. Thus they can be computed in polynomial time. The theorem now follows as in the proof of Theorem 2.3. $\square$

Consider the following definition.

DEFINITION 2.5 ([BH77]). *We say that the language $A \subseteq \Sigma^*$ has* p-time padding functions *iff there exist $e, d \in \mathrm{FP}$ such that the following hold:*

1. *For all $w, x \in \Sigma^*$, $w \in A \Leftrightarrow e(w, x) \in A$.*
2. *For all $w, x \in \Sigma^*$, $d(e(w, x)) = x$.*
3. *For all $w, x \in \Sigma^*$, $|e(w, x)| \geq |w| + |x|$.*

As a simple example, the following is a padding function for SAT:

$$e(w, x) = (w) \wedge c_1 \wedge c_2 \wedge \cdots \wedge c_{|x|},$$

where c_i is $(y \vee \bar{y})$ if the ith bit of x is 1 and $(\bar{y} \vee y)$ otherwise, where y is a Boolean variable numbered higher than all of the Boolean variables occurring in w.

Then the following theorem follows from Theorem 2.4.

THEOREM 2.6 ([BH77]). *If A and B are* NP*-complete and have* p*-time padding functions, then $A \overset{\mathrm{p}}{\cong} B$.*

Finally, Observation 2.2 now follows from the following.

OBSERVATION 2.7 ([BH77]). *All of the* NP*-complete problems in* [GJ] *have* p*-time padding functions.*

Hartmanis also extended the above work as follows: Say that A has *logspace padding functions* if there are logspace-computable functions as in Definition 2.5.

THEOREM 2.8 ([Har]). *If A and B are* NP*-complete via logspace reductions and have logspace padding functions, then A and B are logspace isomorphic.*

Proof. Since A and B have logspace padding functions, we can create functions f and g as in Theorem 2.4 that are length-squaring and computable in logspace. Then the whole ancestor chain can be computed in logspace because each successive iteration requires half of the previous space. $\square$

Here we show that sets complete under a very restrictive notion of reducibility are isomorphic under a very restricted class of isomorphisms. This result is incomparable to a recent result of [AB], which showed that all sets complete under one-way logspace reductions (1-L reductions) are isomorphic under polynomial-time-computable-isomorphisms. (This work of [AB] improves an earlier result of [A88].) Note that it is easy to prove that the class of 1-L reductions is incomparable with the class of first-order projections. Other interesting results concerning 1-L reductions may be found in [BH90, HH].

3. Descriptive complexity. In this section, we recall the notation of descriptive complexity, which we will need to state and prove our main results. See [I89] for a

survey and [IL] for an extensive discussion of the reductions we use here, including first-order projections.

We will code all inputs as finite logical structures. The most basic example is a binary string w of length $n = |w|$. We will represent w as a logical structure:

$$\mathcal{A}(w) = \langle \{0, 1, \ldots, n-1\}, R \rangle,$$

where the unary relation $R(x)$ holds in $\mathcal{A}(w)$ (in symbols, $\mathcal{A}(w) \models R(x)$) just if bit x of w is a 1). As is customary, the notation $|\mathcal{A}|$ will be used to denote the universe $\{0, 1, \ldots, n-1\}$ of the structure $\mathcal{A}$. We will write $\|\mathcal{A}\|$ to denote n, the cardinality of $|\mathcal{A}|$.

A *vocabulary* $\tau = \langle R_1^{a_1} \ldots R_r^{a_r}, c_1, \ldots, c_s \rangle$ is a tuple of an input relation and constant symbols. We call the R_i's "input relations" because they correspond to the input bits to a Boolean circuit. In the case of binary strings, the input relation tells us which bits are 0 and which are 1. In the case of graphs, the input relation E tells us which edges are present.

Let STRUC$[\tau]$ denote the set of all finite structures of vocabulary τ. We define a complexity-theoretic *problem* to be any subset of STRUC$[\tau]$ for some τ.

For any vocabulary τ, there is a corresponding first-order language $\mathcal{L}(\tau)$ built up from the symbols of τ and the numeric relation symbols and constant symbols:[2] $=$, $\leq$, BIT, 0, $\mathbf{m}$, using logical connectives: $\wedge, \vee, \neg$, variables: $x, y, z, \ldots$, and quantifiers: $\forall$, $\exists$.

3.1. First-order interpretations and projections. In [Val], Valiant defined the *projection,* an extremely low-level many–one reduction.

DEFINITION 3.1. *Let $S, T \subseteq \{0,1\}^\star$. A k-ary* projection *from S to T is a sequence of maps $\{p_n\}$, $n = 1, 2, \ldots$, that satisfy the following properties. First, for all n and for all binary strings s of length n, $p_n(s)$ is a binary string of length n^k and*

$$s \in S \quad \Leftrightarrow \quad p_n(s) \in T.$$

Second, let $s = s_0 s_1 \ldots s_{n-1}$. Then each map p_n is defined by a sequence of n^k literals $\langle l_0, l_1, \ldots, l_{n^k-1} \rangle$, where

$$l_i \in \{0, 1\} \cup \{s_j, \bar{s}_j \mid 0 \leq j \leq n-1\}.$$

Thus as s ranges over strings of length n, each bit of $p_n(s)$ depends on at most one bit of s:

$$p_n(s)[[i]] = l_i(s).$$

Projections were originally defined as a nonuniform sequence of reductions—one for each value of n. That is, a projection can be viewed as a many–one reduction produced by a family $\{C_n\}$ of circuits of depth one. The circuits consist entirely of wires connecting input bits or negated input bits to outputs. If the circuit family $\{C_n\}$ is sufficiently *uniform*, we arrive at the class of *first-order projections.* (Recall that first-order corresponds to uniform AC^0 [BIS].) We find it useful to work in the

[2] Here $\leq$ refers to the usual ordering on $\{0, \ldots, n-1\}$, "BIT(i, j)" means that the ith bit of the binary representation of j is 1, and 0 and $\mathbf{m}$ refer to 0 and $n-1$, respectively. For simplicity, we will assume throughout that $n > 1$ and thus $0 \neq \mathbf{m}$. These relations are called "numeric" as opposed to the input relations because, for example, "BIT(i, j)" and "$i \leq j$" depend only on the numeric values of i and j and do not refer to the input.

framework of first-order logic rather than in the circuit model. The rest of this section presents the necessary definitions of first-order reductions.

The idea of the definition is that the choice of the literals $\langle l_0, l_1, \ldots, l_{n^k-1}\rangle$ in Definition 3.1 is given by a first-order formula in which no input relation occurs. Thus the formula can only talk about bit positions and not bit values. The choice of literals depends only on n. In order to make this definition, we must first define first-order interpretations. These are a standard notion from logic for translating one theory into another (cf. [End]), modified so that the transformation is also a many–one reduction [I87]. (For readers familiar with databases, a first-order interpretation is exactly a many–one reduction that is definable as a first-order query.)

DEFINITION 3.2 (first-order interpretations). *Let σ and τ be two vocabularies, with $\tau = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s\rangle$. Let $S \subseteq \mathrm{STRUC}[\sigma]$ and $T \subseteq \mathrm{STRUC}[\tau]$ be two problems. Let k be a positive integer. Suppose we are given an r-tuple of formulas $\varphi_i \in \mathcal{L}(\sigma)$, $i = 1, \ldots, r$, where the free variables of φ_i are a subset of $\{x_1, \ldots, x_{k\cdot a_i}\}$. Finally, suppose we are given an s-tuple of constant symbols[3] $t_1, \ldots, t_s$ from $\mathcal{L}(\sigma)$. Let $I = \lambda_{x_1 \ldots x_d}\langle \varphi_1, \ldots, \varphi_r, t_1, \ldots, t_s\rangle$ be a tuple of these formulas and constants. (Here $d = \max_i(ka_i)$.)*

Then I induces a mapping also called I from $\mathrm{STRUC}[\sigma]$ to $\mathrm{STRUC}[\tau]$ as follows. Let $\mathcal{A} \in \mathrm{STRUC}[\sigma]$ be any structure of vocabulary σ, and let $n = ||\mathcal{A}||$. Then the structure $I(\mathcal{A})$ is defined to be

$$I(\mathcal{A}) \;=\; \langle \{0, \ldots, n^k - 1\}, R_1, \ldots, R_r, t_1, \ldots, t_s\rangle,$$

where the relation R_i is determined by the formula φ_i for $i = 1, \ldots, r$ as follows. Let the function $\langle \cdot, \cdots, \cdot\rangle : |\mathcal{A}|^k \to |I(\mathcal{A})|$ be given by

$$\langle u_1, u_2, \ldots, u_k\rangle \;=\; u_k + u_{k-1}n + \cdots + u_1 n^{k-1}.$$

Then

$$R_i \;=\; \big\{(\langle u_1, \ldots, u_k\rangle, \ldots, \langle u_{1+k(a_i-1)}, \ldots, u_{ka_i}\rangle) \;\big|\; \mathcal{A} \models \varphi_i(u_1, \ldots, u_{ka_i})\big\}.$$

If the structure $\mathcal{A}$ interprets some variables $\bar{u}$, then these may appear freely in the the φ_i's and t_j's of I, and the definition of $I(\mathcal{A})$ still makes sense.

Suppose that I is a many–one reduction from S to T, i.e., for all $\mathcal{A}$ in $\mathrm{STRUC}[\sigma]$,

$$\mathcal{A} \in S \quad \Leftrightarrow \quad I(\mathcal{A}) \in T$$

Then we say that I is a k-ary first-order interpretation *of S to T.*

We are now ready to define first-order projections, a syntactic restriction of first-order interpretations. If each formula in the first-order interpretation I satisfies this syntactic condition, then it follows that I is also a projection in the sense of Valiant. In this case, we call I a first-order projection.

DEFINITION 3.3 (first-order projections). *Let $I = \langle \varphi_1, \ldots, \varphi_r, t_1, \ldots, t_s\rangle$ be a k-ary first-order interpretation from S to T as in Definition* 3.2. *Suppose further that the φ_i's all satisfy the following* projection condition:

$$\varphi_i \equiv \alpha_1 \vee (\alpha_2 \wedge \lambda_2) \vee \cdots \vee (\alpha_e \wedge \lambda_e), \tag{3.1}$$

[3] More generally, we could use closed terms, which are expressions involving constants and function symbols. An even more general way to interpret constants and functions is via a formula φ such that $\vdash (\forall \bar{x})(\exists! y)\varphi(\bar{x}, y)$. However, in this paper, the simpler definition involving constant symbols suffices.

where the α_j's are mutually exclusive formulas in which no input relations occur and where each λ_j is a literal, i.e., an atomic formula $P(x_{j_1}, \ldots, x_{j_a})$ or its negation.

In this case, the predicate $R_i(\langle u_1, \ldots, u_k\rangle, \ldots, \langle \ldots, u_{ka_i}\rangle)$ holds in $I(\mathcal{A})$ if $\alpha_1(\bar{u})$ is true, or if $\alpha_j(\bar{u})$ is true for some $1 < j \leq e$ and the corresponding literal $\lambda_j(\bar{u})$ holds in $\mathcal{A}$. Thus each bit in the binary representation of $I(\mathcal{A})$ is determined by at most one bit in the binary representation of $\mathcal{A}$. We say that I is a first-order projection. *We write rite $S \leq_{\text{fop}} T$ to mean that S is reducible to T via a first-order projection.*

Example 3.4. To help the reader grasp an intuition of the way an fop reduction behaves, let us describe an example. We present here the reduction from 3-SAT, satisfiability of CNF Boolean expressions with exactly three literals per clause, to 3-COL, the problem of coloring the vertices of a graph with three colors under the constraint that the endpoints of all edges get different colors. We use the same reduction as described in [Man, section 11.4.5] so that the reader in need of additional help can consult it there.

The respective vocabularies for the input and output structures are as follows. To describe instances of 3-SAT, clauses and Boolean variables are each numbered from 0 through $n-1$. There are six predicates: $P_i(x,c), N_i(x,c)$, $i = 1,2,3$, indicating that variable x occurs positively or negatively in the ith position of the clause c. The vocabulary for the output structures is simply a binary predicate E that stands for the Boolean adjacency matrix of the output graph. Thus $E(u,v)$ is true exactly when the edge (u,v) is present in the output graph.

The output graph consists of six vertices per clause and two vertices per Boolean variable, plus three additional vertices usually named T, F, and R (standing for true, false, and red). Let an arbitrary 3CNF formula be coded by an input structure,

$$\mathcal{A} = \langle \{0, 1, \ldots, n-1\}, P_1, P_2, P_3, N_1, N_2, N_3 \rangle.$$

The output structure will be a graph with $8n + 3$ relevant vertices. The easiest way for us to code this is to use an fop of arity 2. We will assume for simplicity that n is always greater than or equal to 9.

$$I(\mathcal{A}) \;=\; \langle \{\langle a, b\rangle \,:\, 0 \leq a, b < n\}, E\rangle \;=\; \langle \{0, \ldots, n^2 - 1\}, E\rangle,$$

where

$$E \;=\; \big\{(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) \;\big|\; \mathcal{A} \models \varphi(x_1, x_2, y_1, y_2)\big\}.$$

It remains to write down the first-order projection, φ. To do this, we need some nitty gritty coding. We will let the vertices T, F, and R be the elements $\langle 0,0\rangle, \langle 1,0\rangle$, and $\langle 2,0\rangle$ of $I(\mathcal{A})$, respectively. The formula φ will have three pieces:

$$\begin{aligned}\varphi(x_1, x_2, y_1, y_2) \;=\; \alpha(x_1, x_2, y_1, y_2) \vee \beta(x_1, x_2, y_1, y_2) \;\vee\; \beta(y_1, y_2, x_1, x_2) \\ \vee\, \gamma(x_1, x_2, y_1, y_2) \;\vee\; \gamma(y_1, y_2, x_1, x_2),\end{aligned}$$

where α says that there are edges between T, F, and R; β says that vertices $\langle x, 1\rangle$ and $\langle x, 2\rangle$ representing the variable x and its negation are connected to each other and to R; and γ says that for clause $C = (a \vee b \vee d)$, vertices $\langle C, 6\rangle, \langle C, 7\rangle$, and $\langle C, 8\rangle$ are connected to each other, and the following edges exist: $(\langle C, 3\rangle, \langle C, 6\rangle)$, $(\langle C, 4\rangle, \langle C, 7\rangle)$, $(\langle C, 5\rangle, \langle C, 8\rangle)$, as well as the edges $(a, \langle C, 3\rangle), (T, \langle C, 3\rangle), (b, \langle C, 4\rangle), (T, \langle C, 4\rangle)$, and $(d, \langle C, 5\rangle), (T, \langle C, 5\rangle)$.

In case anyone really wants to see them, here are the formulas written out:

$$\alpha(x_1,x_2,y_1,y_2) \equiv (x_2 = y_2 = 0) \wedge (x_1 \neq y_1) \wedge (x_1 \leq 2) \wedge (y_1 \leq 2),$$
$$\beta(x_1,x_2,y_1,y_2) \equiv (x_2 = 1 \wedge y_2 = 2 \wedge x_1 = y_1) \vee (x_1 = 2 \wedge x_2 = 0 \wedge (1 \leq y_2 \leq 2)),$$
$$\begin{aligned}\gamma(x_1,x_2,y_1,y_2) \equiv\ & (x_1 = x_2 = 0 \wedge (3 \leq y_1 \leq 5) \wedge (3 \leq y_2 \leq 5)\\ & \vee (x_1 = y_1 \wedge (3 \leq x_2 \leq 5) \wedge (y_2 = x_2 + 3))\\ & \vee [(x_2 = 1 \wedge y_2 = 3) \wedge P_1(x_1,y_1)] \vee [(x_2 = 2 \wedge y_2 = 3) \wedge N_1(x_1,y_1)]\\ & \vee [(x_2 = 1 \wedge y_2 = 4) \wedge P_2(x_1,y_1)] \vee [(x_2 = 2 \wedge y_2 = 4) \wedge N_2(x_1,y_1)]\\ & \vee [(x_2 = 1 \wedge y_2 = 5) \wedge P_3(x_1,y_1)] \vee [(x_2 = 2 \wedge y_2 = 5) \wedge N_3(x_1,y_1)]\\ & \vee (x_1 = y_1 \wedge x_2 \neq y_2 \wedge (6 \leq x_2 \leq 8) \wedge (6 \leq y_2 \leq 8)).\end{aligned}$$

4. Main theorem and proof.

THEOREM 4.1. *Let $\mathcal{C}$ be a nice complexity class, e.g.,* L, NL, P, NP, *etc. Let S and T be complete for $\mathcal{C}$ via first-order projections. Then S and T are isomorphic via a first-order isomorphism.*

To prove Theorem 4.1, we begin with the following lemma. Note the similarity between Lemma 4.2 and the proofs of Theorems 2.4 and 2.8. For simplicity, in this lemma we are assuming that I is a single fop that maps STRUC$[\sigma]$ to itself. The proof for the case with two fops and two vocabularies as in Lemma 4.3 is similar.

LEMMA 4.2. *Let I be an fop that is* 1:1 *and of arity greater than or equal to* 2 *(i.e., it at least squares the size). Then the following two predicates are first-order expressible concerning a structure $\mathcal{A}$:*

a. IE($\mathcal{A}$), *which means that $I^{-1}(\mathcal{A})$ exists*;

b. *#Ancestors*($\mathcal{A}, r$), *which means that the length of $\mathcal{A}$'s maximal ancestor chain is r.*

Proof. Let $I = \lambda_{x_1 \ldots x_d}\langle \varphi_1, \ldots, \varphi_r, t_1, \ldots, t_s\rangle$, where each φ_i is in the form of equation (3.1). To prove a, just observe that each bit of the relation R_i of $\mathcal{A}$ either (1) depends on exactly one bit of some preimage $\mathcal{B}$ (specified by an occurrence of a literal λ_{ij} in φ_i) or (2) does not depend on any bit of a preimage. In case (2), a given bit of $\mathcal{A}$ is either "right" or "wrong." Thus $\mathcal{A}$ has an inverse iff no bit of $\mathcal{A}$ is wrong, and no pair of bits from $\mathcal{A}$ are determined by the same bit of $\mathcal{A}$'s preimage in conflicting ways. We can check this in a first-order way by checking that for all pairs of bits from $\mathcal{A}$, $R_i(\bar{a})$ and $R_{i'}(\bar{b})$, either they do not depend on the same bit from $\mathcal{B}$, or the same value of that bit gives the correct answer for $R_i(\bar{a})$ and $R_{i'}(\bar{b})$. Furthermore, the preimage $\mathcal{B}$ if it exists can be described uniquely by a first-order formula that chooses the correct bits determined by entries of $\mathcal{A}$. *N.B.* Since we have assumed that I is 1:1, every bit of $I^{-1}(\mathcal{A})$ is determined by some bit of $\mathcal{A}$.

b. To express *#Ancestors*($\mathcal{A}, r$), we want to describe the existence of an ancestor chain:

$$\mathcal{A}_r \xrightarrow{I} \mathcal{A}_{r-1} \xrightarrow{I} \cdots \xrightarrow{I} \mathcal{A}_1 \xrightarrow{I} \mathcal{A}_0 = \mathcal{A}. \tag{4.1}$$

We will then assert that this is the maximal-length such chain, i.e.,

$$\neg\mathrm{IE}(\mathcal{A}_r) \wedge (\forall k < r)\mathrm{IE}(\mathcal{A}_k) \tag{4.2}$$

Equation (4.2) expresses the existence of the ancestor chain (4.1) inductively in the following sense. Once we know that $\mathcal{A}_k$ exists, we can ascertain the value $\mathcal{A}_k[[p_k]]$ of the bit at position p_k of $\mathcal{A}_k$ by exhibiting a certificate:

$$C(k,p_k) = \langle(\mathcal{A}_k[[p_k]],p_k),(\mathcal{A}_{k-1}[[p_{k-1}]],p_{k-1}),\ldots,(\mathcal{A}_0[[p_0]],p_0)\rangle.$$

We can say in a first-order sentence that $C(k, p_k)$ is internally consistent. That is, for all i with $k > i \geq 0$, bit p_{i+1} of $\mathcal{A}_{i+1}$ is determined correctly via I by bit p_i of $\mathcal{A}_i$.[4] Note that because each structure $\mathcal{A}_{i+1}$ is of size at most the square root of the size of $\mathcal{A}_i$, the certificate requires only $O(\log n)$ bits, i.e., a constant number of variables, to express.

Thus in equation (4.2), we refer to bit p_k of the structure $\mathcal{A}_k$ by existentially quantifying an internally consistent certificate $C(k, p_k)$. We know inductively that since $\mathrm{IE}(\mathcal{A}_{k-1})$, the bit value determined by $C(k, p_k)$ is unique and correct. □

LEMMA 4.3. *If S and T are interreducible via* 1:1 *fops I and J each of arity at least* 2*, then S and T are isomorphic via first-order isomorphisms.*

Proof. Let $\mathcal{A}$ be a structure in the vocabulary of S, and, as in the proof of Theorem 2.3, define the length of the ancestor chain of $\mathcal{A}$ to be the length of the longest sequence of the form $J^{-1}(\mathcal{A})$, $I^{-1}(J^{-1}(\mathcal{A}))$, $J^{-1}(I^{-1}(J^{-1}(\mathcal{A}))), \ldots$. The argument given in Lemma 4.2 shows that there is a formula $\#Ancestors(\mathcal{A}, r)$ that evaluates to true iff $\mathcal{A}$'s ancestor chain has length r. Lemma 4.2 also shows that there is a formula computing J^{-1}. The desired isomorphism is now the function b such that the ith bit of $b(\mathcal{A})$ is 1 iff the following first-order formula is true:

$$(\exists r)\left(\#Ancestors(\mathcal{A}, r) \wedge (\mathrm{BIT}(0, r) \wedge I(i)) \vee (\neg \mathrm{BIT}(0, r) \wedge J^{-1}(i))\right)$$

(Note that this first-order isomorphism b is not, strictly speaking, a first-order interpretation since it maps some inputs to strictly shorter outputs, which is impossible for an interpretation.) □

It now remains to show the following.

LEMMA 4.4. *Suppose that a problem S is complete via fops for a nice complexity class $\mathcal{C}$. Then S is complete for $\mathcal{C}$ via fops that are* 1:1 *and of arity at least* 2.

Proof. Of course, it remains to define "nice," but here is the proof. Every nice complexity class has a universal complete problem:

$$U_{\mathcal{C}} = \left\{M\$w\#^r \;\middle|\; M(w)\downarrow \text{ using resources } f_{\mathcal{C}}(r)\right\}. \tag{4.3}$$

Here $f_{\mathcal{C}}(r)$ defines the appropriate complexity measure, e.g., r nondeterministic steps for NP, deterministic space $\log r$ for L, space 2^r for EXPSPACE, etc.

We claim that $U_{\mathcal{C}}$ is complete for $\mathcal{C}$ via fops that are 1:1 and of arity at least 2. In order to make this claim, we need to agree on an encoding of inputs to $U_{\mathcal{C}}$ that allows us to interpret them as structures over some vocabulary. Since all of our structures are encoded in binary, we will encode \$ and # by 10 and 11, respectively, and the binary bits 0 and 1 constituting M and w will be encoded by 00 and 01, respectively. Now, as in, for example, [I87], we consider a binary string of length n to be a structure with a single unary predicate over a universe of size n. Now for any given problem $T \in \mathcal{C}$ accepted by machine M, we show that T is reducible to $U_{\mathcal{C}}$ via a fop that is 1:1 and of arity at least 2. The fop simply maps input w to the string $M\$w\#^r$, for an appropriate r which we can always take to be at least $|w|^2$. The fop checks that if $i \leq 2|M|$, then the odd-numbered bits are 0 and if i is even, then the ith bit is 1 iff the $i/2$nd bit of M is 1. Similarly, if $2|M| + 2 < i \leq 2(|M| + |w| + 1)$, then the odd-numbered bits are 0 and the even numbered bits are the corresponding bit of w, etc.

[4] The reader who is more familiar with bit hacking on Turing machines than with first-order formulas could instead convince herself that this can be done by an alternating Turing machine running in logarithmic time and making $O(1)$ alternations; first-order expressibility follows by [BIS].

To complete the proof of the lemma, let T be any problem in $\mathcal{C}$ and let S be as above. Then we reduce T to S via a 1:1 length-squaring fop as follows. First, reduce T to $U_{\mathcal{C}}$ as above. Next, reduce $U_{\mathcal{C}}$ to S via the fop promised in the statement of the lemma.

It is easy to verify that, using the encoding that we have chosen for $U_{\mathcal{C}}$, it holds that for every length n, for all $i \leq n$, there are two strings x and y of length n, differing only in position i, such that $x \in U_{\mathcal{C}}$ and $y \notin U_{\mathcal{C}}$.

Thus the fop from $U_{\mathcal{C}}$ cannot possibly ignore any of the bits in its input. However, an fop cannot process several bits into one; it can only either ignore a bit or copy it, or negate it, and this choice is made independently of the values of any of the bits.

It follows that the composition of these two fops is the 1:1 length-squaring fop that we desire. (Note that an fop by definition must have arity at least 1 and thus cannot be length-decreasing on Boolean strings.) □

From the above three lemmas, we have a first-order version of Theorem 2.3, and thus Theorem 4.1 follows.

We can inspect the proof of Lemma 4.4 to get a definition of "nice." A complexity class is "nice" if it has a universal complete problem via fops as in equation (4.3). It is easy to check that the following complexity classes, among many others, are nice and thus meet the conditions of Theorem 4.1.

PROPOSITION 4.5. *The following complexity classes are nice:* NC^1, L, NL, LOG(CFL), NC^2, P, NP, PSPACE, EXPTIME, and EXPSPACE.

Proof. This is immediate for the Turing-machine-based classes: L, NL, P, NP, PSPACE, EXPTIME, and EXPSPACE. It similarly follows for the other three classes using the definitions $\mathrm{NC}^i = \mathrm{ASPACE}[\log n] - \mathrm{TIME}[(\log n)^i]$ and $\mathrm{LOG(CFL)} = \mathrm{ASPACE}[\log n] - \forall\mathrm{TIME}[\log n]$. □

5. More on the relationship between isomorphisms and projections. There are several questions about isomorphisms among complete sets that can be answered in the setting of first-order computation but are open for general polynomial-time computation. It is not known whether one-way functions exist since their existence would imply that P $\neq$ NP. However, if one-way functions exist (i.e., if P $\neq$ UP), then there exists a one-way function f such that f(SAT) is polynomial-time isomorphic to SAT [Ga].

Here we can be more definitive: the bijection $f(x) = 3x \pmod{2^{|x|}}$ was shown in [BL] to be one-way for first-order computation, in the sense that f is first-order expressible but f^{-1} is not. (See also [Hås] for other examples.) However, it is not too hard to show that for this choice of f, f(SAT) is complete for NP under first-order projections, and thus it is first-order isomorphic to SAT.

The next result shows that the class of sets that are complete under first-order projections is not closed under first-order isomorphisms. (This also seems to be the first construction of a set that is complete for NP under first-order (or even poly-time) many–one reductions that is not complete under first-order projections.)

THEOREM 5.1. *There is a set first-order isomorphic to* SAT *that is not complete for* NP *under first-order projections.*

Proof. Let $g(x)$ be a string of $|x|^2$ bits, with bit $x_{i,j}$ representing the logical AND of bits i and j of x. Let $A = \{\langle x, g(x)\rangle : x \in \mathrm{SAT}\}$. By an extension of the techniques used in proving Theorem 4.1, it can be shown that A is first-order isomorphic to SAT. However, a direct argument shows that there cannot be any projection (even a nonuniform projection) from SAT to A. (*Sketch*: For all n, one can find bit positions i and j that are independent of each other and independent of every other bit position,

in the sense that for any setting b of bit j, there are two words that differ only in bit i, having b in position j, such that one of the words is in SAT and one is not. No projection reducing SAT to another language can "ignore" either i or j. However, since i and j are independent of all other bit positions, no projection can encode the AND of bits i and j.) □

A natural question that remains open is the question of whether every set that is complete for NP under first-order many–one reductions is first-order isomorphic to SAT. A related question is whether one can construct a set that is complete for NP under poly-time many–one reductions that is not first-order isomorphic to SAT. Since so many tools are available for proving the limitations of first-order computation, we are optimistic that this and related questions about sets that are complete under first-order reductions should be tractable.[5] Furthermore, we hope that insights gleaned in answering these questions will be useful in guiding investigations of the polynomial-time degrees.

Acknowledgments. The authors wish to thank the organizers of the 1992 Seminar on Structure and Complexity Theory at Schloß Dagstuhl, where this work was initiated. We also thank Richard Beigel, Jose Antonio Medina, and two anonymous referees for comments on an earlier draft.

REFERENCES

[AB] M. Agrawal and S. Biswas, *Polynomial isomorphism of* 1*-L-complete sets*, in Proc. 8th Annual Structure in Complexity Theory Symposium, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 75–79.

[A88] E. Allender, *Isomorphisms and* 1*-L reductions*, J. Comput. System Sci., 36 (1988), pp. 336–350.

[A89] E. Allender, *P-uniform circuit complexity*, J. Assoc. Comput. Mach., 36 (1989), pp. 912–928.

[AG91] E. Allender and V. Gore, *On strong separations from* AC^0, in Advances in Computational Complexity Theory, J.-Y. Cai, ed., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 13, AMS, Providence, RI, 1993, pp. 21–37.

[AG92] E. Allender and V. Gore, *A uniform circuit lower bound for the permanent*, SIAM J. Comput., 23 (1994), pp. 1026–1049.

[BIS] D. M. Barrington, N. Immerman, and H. Straubing, *On uniformity within* NC^1, J. Comput. System Sci., 41 (1990), pp. 274–306.

[BH77] L. Berman and J. Hartmanis, *On isomorphism and density of NP and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.

[BL] R. Boppana and J. Lagarias, *One-way functions and circuit complexity*, Inform. and Comput. 74 (1987), pp. 226–240.

[BH90] H.-J. Burtschick and A. Hoene, *The degree structure of* 1*-L reductions*, in Proc. Math. Foundations of Computer Science, Lecture Notes in Comput. Sci. 629, Springer-Verlag, Berlin, 1992, pp. 153–161.

[CSV] A. Chandra, L. Stockmeyer, and U. Vishkin, *Constant depth reducibility*, SIAM J. Comput., 13 (1984), pp. 423–439.

[Coo] S. Cook, *The complexity of theorem proving procedures*, in Proc. 3rd Annual ACM Symposium on the Theory of Computing, ACM, New York, 1971, pp. 151–158.

[Dah] E. Dahlhaus, *Reduction to NP-complete problems by interpretations*, in Logic and Machines: Decision Problems and Complexity, E. Börger, D. Rödding, and G. Hasenjaeger, eds., Lecture Notes in Comput. Sci. 171, Springer-Verlag, Berlin, 1984, pp. 357–365.

[5] One possible approach might be to attempt to construct a first-order analogue of the "scrambling" and "annihilating" functions studied in [KMR89]. However, we suspect that this particular approach is likely to be difficult since this would involve constructing sets that have a sort of "immunity" property relative to AC^0. Related problems (although not precisely this problem) were shown in [AG91] to imply the solution to some long-standing open questions in complexity theory.

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

[Fa] R. Fagin, *Generalized first-order spectra and polynomial-time recognizable sets*, in Complexity of Computation: Proc. SIAM–AMS Symposia, R. Karp, ed., Vol. 7, SIAM, Philadelphia, 1974, pp. 43–73.

[FSS] M. Furst, J. Saxe, and M. Sipser, *Parity, circuits, and the polynomial-time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13–27.

[Ga] K. Ganesan, *One-way functions and the isomorphism conjecture*, Theoret. Comput. Sci., 129 (1994), pp. 309–321.

[GJ] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman, San Francisco, 1979.

[Hås] J. Håstad, *One-way permutations in NC^0*, Inform. Process. Letters, 26 (1987), pp. 153–155.

[Har] J. Hartmanis, *On the logtape isomorphism of complete sets*, Theoret. Comput. Sci., 7 (1978), pp. 273–286.

[HIM] J. Hartmanis, N. Immerman, and S. Mahaney, *One-way log tape reductions*, in Proc. 19th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1978, pp. 65–72.

[HH] L. Hemachandra and A. Hoene, *Collapsing degrees via strong computation*, J. Comput. System Sci., 46 (1993), pp. 363–380.

[I87] N. Immerman, *Languages that capture complexity classes*, SIAM J. Comput., 16 (1987), pp. 760–778.

[I89] N. Immerman, *Descriptive and computational complexity*, in Computational Complexity Theory, J. Hartmanis, ed., Proc. Sympos. Appl. Math. 38, AMS, Providence, RI, 1989, pp. 75–91.

[IL] N. Immerman and S. Landau, *The complexity of iterated multiplication*, Inform. and Comput., 116 (1995), pp. 103–116.

[Jon] N. Jones, *Space-bounded reducibility among combinatorial problems*, J. Comput. System Sci., 11 (1975), pp. 68–85.

[JY] D. Joseph and P. Young, *Some remarks on witness functions for non-polynomial and non-complete sets in NP*, Theoret. Comput. Sci., 39 (1985), pp. 225–237.

[Kar] R. Karp, *Reducibility among combinatorial problems*, in Complexity of Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[Kel] J. L. Kelley, *General Topology*, Van Nostrand/Reinhold, New York, 1955.

[KMR89] S. Kurtz, S. Mahaney, and J. Royer, *The isomorphism conjecture fails relative to a random oracle*, in Proc. 21st ACM Symposium on the Theory of Computing, ACM, New York, 1989, pp. 157–166.

[KMR90] S. Kurtz, S. Mahaney, and J. Royer, *The tructure of complete degrees*, in Complexity Theory Retrospective, A. Selman, ed., Springer-Verlag, Berlin, 1990, pp. 108–146.

[Man] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison–Wesley, Reading, MA, 1989.

[MI] J. A. Medina and N. Immerman, *A syntactic characterization of NP-completeness*, in IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 241–250.

[Myh] J. Myhill, *Creative sets*, Z. Math. Logik Grundlag. Math., 1 (1955), pp. 97–108.

[SV] S. Skyum and L. Valiant, *A complexity theory based on Boolean algebra*, J. Assoc. Comput. Mach., 32 (1985), pp. 484–502.

[Ste] I. Stewart, *Using the Hamiltonian operator to capture NP*, J. Comput. System Sci., 45 (1992), pp. 127–151.

[Val] L. Valiant, *Reducibility by algebraic projections*, Enseign. Math., 28 (1982), pp. 253–268.

[You] P. Young, *Juris Hartmanis: Fundamental contributions to isomorphism problems*, in Complexity Theory Retrospective, A. Selman, ed., Springer-Verlag, Berlin, 1990, pp. 28–58.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 568–581, April 1997

012

GENERAL TECHNIQUES FOR ANALYZING RECURSIVE ALGORITHMS WITH APPLICATIONS*

RAKESH M. VERMA†

Abstract. The complexity of divide-and-conquer algorithms is often described by recurrences of various forms. In this paper, we develop general techniques and master theorems for solving several kinds of recurrences, and we give several applications of our results. In particular, almost all of the earlier work on solving the recurrences considered here is subsumed by our work. In the process of solving such recurrences, we establish interesting connections between some elegant mathematics and analysis of recurrences. Using our results and improved bipartite matching algorithms, we also improve existing bounds in the literature for several problems, viz, associative-commutative (AC) matching of linear terms, associative matching of linear terms, rooted subtree isomorphism, and rooted subgraph homeomorphism for trees.

Key words. analysis of algorithms, divide-and-conquer, recurrences, problem complexity, subtree isomorphism, associative-commutative (AC) matching, graph algorithms

AMS subject classifications. 68Q25, 11B37, 68T10, 68R10, 68Q35

PII. S0097539792240583

1. Introduction.

This paper investigates recurrences that arise frequently in the analysis of divide-and-conquer algorithms. Divide-and-conquer is an important and useful technique in the design of efficient sequential and parallel algorithms with innumerable applications. For example, much of the book by Aho et al. [1] consists of divide-and-conquer algorithms. The recurrences considered here are of three basic forms (with appropriate initial conditions):

$$T(n,m) = \sum_{i=1}^{k}\sum_{j=1}^{l} T(n_i, m_j) + h(k,l), \tag{1}$$

$$T(n,m) = \sum_{i=1}^{k}\sum_{j=1}^{l} T(n_i, m_j) + h(n,m), \tag{2}$$

$$T(n) = \sum_{i=1}^{k} a_i T(n/c_i) + f(n), \quad k \geq 2. \tag{3}$$

Our choice of the forms and dimensions (a recurrence in k variables, integer $k > 0$, will be called *k-dimensional*) in this paper is motivated by both the applications and the desire to minimize redundancy. If necessary, our results can be extended from the one-dimensional case to the two-dimensional case, or specialized in the opposite direction, for recurrences of the same form. In an earlier paper [22], we considered the recurrence $T(n) = aT(n/c)+f(n)$ and proved a general theorem on its analysis. These recurrences typically arise in connection with recursive algorithms described as follows: "To process an input of size n (or (n,m), etc.), spend an amount of computational effort $f(n)$ (or $h(n,m)$, etc.) and recursively solve derived instances of the same pro-

* Received by the editors November 30, 1992; accepted for publication (in revised form) June 7, 1995. This research was supported in part by NSF grants CCR-9010366 and CCR-9303011.
http://www.siam.org/journals/sicomp/26-2/24058.html

† Department of Computer Science, University of Houston, Houston, TX 77004 (rmverma@cs.uh.edu).

blem having size $c(n)$ (or n/c_i, (n_i, m_j), etc.)." Such recurrences also arise in other situations, e.g., as deterministic counterparts of probabilistic recurrences.

Earlier systematic efforts on solving recurrences have emphasized linear one-dimensional recurrences. Methods developed for them include differential-equation methods, operator methods, the generating-function approach, etc. Some effort has been given to nonlinear recurrences, but here the path is full of difficulties (see [11, 14] for a detailed discussion on linear and some nonlinear recurrences). Recently, some pioneering work has been done by Karp [15] on probabilistic recurrences. Much of this work, however, is difficult to apply to the recurrences considered here.[1] Earlier work known to us that has a direct bearing on our investigations in this and the earlier paper is as follows: a theorem on divide-and-conquer with equal parts (see [1, 3, 5] and [7, p. 62]) and the two theorems given below.

Let $G = (V_1, V_2, E)$ be a bipartite graph with $r = |V_1|$, $s = |V_2|$, and $r \leq s$. Let p and q be the two trees for rooted subtree isomorphism with sizes n and m, $n \leq m$ (p is to be mapped into q). In 1977 Reyner claimed and in 1988 Verma and Reyner [21, 25] proved the following.

THEOREM 1.1 ([25]). *Given an algorithm for bipartite matching that requires at most $O(rs^u)$ operations, where $u > 1$, the subtree algorithm requires at most $O(nm^u)$ operations.*

THEOREM 1.2 ([20]). *Given an algorithm for bipartite matching that requires at most $O(rs)$ operations, the subtree algorithm will require at most $O(nm \ln n)$ operations.*

In this paper, we present some techniques and several "master theorems" that can be used to obtain fairly tight upper bounds on the functions $T(n)$ and $T(n, m)$ of the above recurrences. In the process, we establish connections between some elegant mathematics (viz, theory of convex functions and inequalities) and analysis of recurrences, which appears to have been missed so far. An immediate motivation for analyzing these recurrences must also be mentioned. With progress in computer science, the computational expenditure f keeps decreasing, which forces us to reconsider the analysis and rederive the bounds on $T(n)$. For example, the complexity of bipartite matching was improved from $O(rs^{1.5})$ (see Hopcroft and Karp [13]) to $O((r+s)^{1.5}\sqrt{rs/\log s})$ in 1990 by Alt et al. [2] and further to $O(rs^{1.5}/\log s)$ in 1991 by Feder and Motwani [8] ($r \leq s$ are the sizes of the two vertex sets). Since these new bounds do not satisfy the hypotheses of Theorems 1.1 and 1.2, one is forced to reanalyze the rooted subtree isomorphism algorithms from scratch. This rework can be avoided if we can prove powerful theorems that can be applied to large classes of functions representing the computational expenditure for dividing the problem and merging the solutions.

1.1. Overview of our approach and results. To enhance the applicability of our results, we extract the salient features of a variety of situations as follows. We consider a homogeneous collection $\mathcal{C}$ of finite data structures partially ordered by inclusion and for which a suitable *size* function has been defined (see section 3). We then consider divide-and-conquer algorithms of a very general form with inputs from $\mathcal{C}$ in two kinds of situations. The first is the more general one in which the sizes of the

[1] Recurrence (3) can be transformed into a linear, nonhomogeneous, k-dimensional, $k \geq 1$, recurrence by the substitution $U(n_1, n_2, \ldots, n_k) = T(c_1^{n_1} \ldots c_k^{n_k})$. However, there are two problems in applying existing work to the resulting recurrence: nonhomogeneity and dimensions. Techniques are scarce and hard to apply for $k > 2$, and for $k \leq 2$ complicated summations must still be evaluated to obtain solutions by existing methods (see [14, 19]). This approach cannot be applied to recurrences (1) and (2).

derived problem instances may depend not just on the size of the original instance but on the instance itself, e.g., $T(n,m) = \sum\sum T(n_i, m_j) + h(k,l)$. In the second, the dependence is only on the size of the original instance and takes a certain known form, e.g., $T(n) = \sum a_i T(n/c_i) + f(n)$. We then prove the following general results.

Recurrence (1). Here we study the general situation in which the sizes of the derived instances depend on both the original instance and its size. In such situations, we assume that the *sum* of the sizes of the derived instances is less than the size of the original instance, which holds in a number of practical examples. We extend the definition of additivity (see section 2) (called superadditivity in the mathematics literature) of a univariate function to multivariate functions in a natural way. Then, under fairly weak assumptions, we obtain upper bounds for T when (i) h is biadditive, (ii) h is additive in one argument only, say the first, and there is a function f such that $h(\_, f(\_))$ is biadditive, (iii) h is additive in neither argument but there are f and g such that $h(f(\_), g(\_))$ is biadditive, and (iv) h is a continuous pseudoconvex function (section 3).

Recurrence (2). As in the previous paragraph, this is the general situation and we make the same assumption on the sum of the sizes of the derived instances. Then, under weak assumptions, we obtain upper bounds for T when h is biadditive and when h is continuous and pseudoconvex.

Recurrence (3). For simplicity and rigor, we consider recurrences over a real variable x instead of an integer variable n. We identify two crucial properties of a function, which we call g-star-shaped (this extends the definition of a star-shaped function in the mathematics literature) and g-co-star-shaped, and then prove master theorems, using the powerful principle of noetherian induction, under a variety of hypotheses (see section 5).

Our proofs make use of the basic properties of convex, pseudoconvex, and biadditive functions. These are included in section 2.1. We also give some conditions which imply that a function is biadditive, pseudoconvex, or convex, which we use in applying our results. These are (mostly) known and are included in section 4 to make the paper self-contained and to demonstrate the richness of these classes of functions.

Applications. In section 6, we present some of the applications of our theorems. Specifically, in the two-dimensional case, we prove that all three problems, rooted subtree isomorphism, associative-commutative (AC) matching of linear terms, and rooted subgraph homeomorphism on trees, have tight upper bounds of $O(nm^{1.5}/\log m)$. Our approach unifies the analysis of these problems and improves the existing upper bounds (given by Verma and Reyner [25], Verma and Ramakrishnan [24, 23], and Chung [6]) for these problems by a factor of $\log m$ in each case. In the process, we demonstrate the existence of a much tighter relationship between these three problems and bipartite matching than previously known. We also prove that all three problems above require time $O(nm)$ if there is a bipartite matching of time complexity $O(rs)$ and that associative matching of linear terms can be done in $O(nm)$ time. First, this improves by a factor of $\log n$ the existing upper bounds for (a) rooted subtree isomorphism (see Theorem 1.2 above) and (b) associative matching of linear terms, given by Verma and Ramakrishnan [24, 23]. Second, it implies that there are parallel algorithms for all three problems of time complexity $O(mn)$ using the parallel algorithms for bipartite matching [9, 10]. Finally, solutions of recurrences arising in [4] and [17] are corollaries of our theorems for the one-dimensional case.

Some implications of this work are as follows. First, much of the existing work on these recurrences, e.g., the popular theorem on analyzing divide-and-conquer algorithms with equal parts [1, 3, 5, 7, 14] (see Corollary 5.7) and the two theorems

given above, is subsumed by our results. Second, we establish interesting connections between some elegant mathematics and analysis of recursive algorithms. The power of our results means that considerable reworking may be avoided with future progress in computer science and upper bounds for new algorithms that fit our framework could be painlessly obtained. Finally, our work opens up an interesting area of research.

2. Preliminaries.

Notation. 1. All functions are defined on the nonnegative reals and take only nonnegative real values unless explicitly stated otherwise. 2. All variables p, q, x, and y and their subscripted versions take only nonnegative real values unless explicitly stated otherwise. Thus the phrase "for all x" means "for all nonnegative real values of x." 3. All variables m and n and their subscripted versions take only nonnegative integral values unless explicitly stated otherwise. 4. The restriction of a function f on the nonnegative reals to nonnegative integers is indicated by changing the arguments of f from real valued, e.g., x, to integer valued, e.g., n.

DEFINITION 2.1. *We say that a unary function f is* additive *iff $f(x) + f(y) \leq f(x+y)$ for all x and y. We say that a binary function h is* biadditive *iff $h(x_1, y) + h(x_2, y) \leq h(x_1 + x_2, y)$ and $h(x, y_1) + h(x, y_2) \leq h(x, y_1 + y_2)$ for all values of the arguments to h. Similarly, we say that a function is additive for $x > a$ if the additivity requirement is satisfied for all values of x greater than a. Analogously, we define biadditivity for $x > a$ and $y > b$.*

Remarks. An additive nonnegative function must be nondecreasing since by the additivity of f, $f(x_1) + f(x_2 - x_1) \leq f(x_2)$ for $x_1 \leq x_2$, which by the nonnegativity of f implies $f(x_1) \leq f(x_2)$ for $x_1 \leq x_2$. Similarly, a biadditive nonnegative function must be nondecreasing in x for each y and in y for each x. An easy induction shows that the biadditivity requirement on h implies that $\sum_{i=1}^{n} \sum_{j=1}^{m} h(a_i, b_j) \leq h(\sum_{i=1}^{n} a_i, \sum_{j=1}^{n} b_j)$.

DEFINITION 2.2. *A unary function f is* convex *iff $f((x+y)/2) \leq (f(x)+f(y))/2$ for all x and y. A binary function f on the nonnegative reals is* convex *iff $f((x_1 + x_2)/2, (y_1 + y_2)/2) \leq (f(x_1, y_1) + f(x_2, y_2))/2$ for all x_1, x_2, y_1, and y_2.* Notice that the definition of convexity for binary functions requires more than convexity in x and y separately. Functions that are convex in x for every y and in y for every x will be called pseudoconvex.

DEFINITION 2.3. *A binary function f is* pseudoconvex *iff $f((x_1 + x_2)/2, y) \leq (f(x_1, y) + f(x_2, y))/2$ for all x_1, x_2, and y and $f(x, (y_1 + y_2)/2) \leq (f(x, y_1) + f(x, y_2))/2$ for all x, y_1, and y_2.*

2.1. Properties of convex and pseudoconvex functions. We need some basic properties of convex and pseudoconvex functions to prove our theorems in the following sections.

LEMMA 2.4 ([12]). *Let f be a continuous convex function on the nonnegative reals. Then $f((px + qy)/(p + q)) \leq (pf(x) + qf(y))/(p + q)$.*

LEMMA 2.5. *Let f be a binary, continuous, pseudoconvex function on the nonnegative reals. We have the following:*

1. $f((px_1 + qx_2)/(p + q), y) \leq (pf(x_1, y) + qf(x_2, y))/(p + q)$.
2. $f(x, (py_1 + qy_2)/(p + q)) \leq (pf(x, y_1) + qf(x, y_2))/(p + q)$.
3. $f(x_1, y_1)+f(x_2, y_2) \leq f(x_1+x_2, y_1+y_2)+f(x_1+x_2, 0)+f(0, y_1+y_2)+f(0, 0)$.

$$\sum_{i=1}^{n} f(x_i, y_i) \leq f\left(\sum_{i=1}^{n} x_i, \sum_{i=1}^{n} y_i\right) + \sum_{j=1}^{n-1} \left(f\left(\sum_{i=1}^{j+1} x_i, 0\right) + f\left(0, \sum_{i=1}^{j+1} y_i\right)\right) + (n-1) f(0, 0).$$

4. $\sum_{i=1}^{n} f(x_i, y) \leq f(\sum_{i=1}^{n} x_i, y) + (n-1)f(0, y)$. *Similarly,* $\sum_{j=1}^{m} f(x, y_j) \leq f(x, \sum_{j=1}^{m} y_j) + (m-1)f(x, 0)$.

5. *Finally,* $\sum_{i=1}^{n} \sum_{j=1}^{m} f(x_i, y_j) \leq f(\sum_{i=1}^{n} x_i, \sum_{j=1}^{m} y_j) + (n-1)f(0, \sum_{j=1}^{m} y_j) + (m-1)f(\sum_{i=1}^{n} x_i, 0) + (m-1)(n-1)f(0, 0)$.

Proof.

1. For every y, we can apply Lemma 2.4 since f is convex in x for each y.

2. This is proved by the same reason as 1.

3. We first prove that $f(x_1, y) + f(x_2, y) \leq f(x_1 + x_2, y) + f(0, y)$, a very useful inequality. We will call this and the corresponding inequality in y the *addition inequalities* for pseudoconvex functions. Let $p = t_1$, $x_1 = t_1 + t_2$, $q = t_2$, and $x_2 = 0$ in property 1 of this lemma. Then

$$f(t_1, y) \leq t_1 f(t_1 + t_2, y)/(t_1 + t_2) + t_2 f(0, y)/(t_1 + t_2).$$

Now let $p = t_2$, $x_1 = 0$, $q = t_1$, and $x_2 = t_1 + t_2$. Then

$$f(t_2, y) \leq t_2 f(t_1 + t_2, y)/(t_1 + t_2) + t_1 f(0, y)/(t_1 + t_2)$$

Adding the two inequalities for f above, we have $f(t_1, y) + f(t_2, y) \leq f(t_1 + t_2, y) + f(0, y)$. Similarly, we have $f(x, t_1) + f(x, t_2) \leq f(x, t_1 + t_2) + f(x, 0)$. This proof of the addition inequalities is from [18].

Now we have $f(x_1, y_1) + f(x_2, y_1) \leq f(x_1 + x_2, y_1) + f(0, y_1)$. Since f is nonnegative, this implies $f(x_1, y_1) \leq f(x_1 + x_2, y_1) + f(0, y_1)$. Similarly, $f(x_2, y_2) \leq f(x_1 + x_2, y_2) + f(0, y_2)$. Adding the last two inequalities, we have $f(x_1, y_1) + f(x_2, y_2) \leq f(x_1 + x_2, y_1) + f(x_1 + x_2, y_2) + f(0, y_1) + f(0, y_2) \leq f(x_1 + x_2, y_1 + y_2) + f(x_1 + x_2, 0) + f(0, y_1 + y_2) + f(0, 0)$, which was what we wanted to prove.

For the generalization, use induction and the first part.

4. This is proved by induction on n using the addition inequalities.

5. This is proved by application of 4. □

3. Two-dimensional recurrences. Let $\mathcal{C}$ be any homogeneous class of finite data structures, partially ordered by inclusion. For example, $\mathcal{C}$ may be the class of all finite graphs partially ordered by the subgraph relation, or $\mathcal{C}$ may be the class of all finite rooted trees partially ordered by the subtree relation, etc. Further, let a *size* be defined for each member of $\mathcal{C}$ by a positive monotonic *size function* s.

DEFINITION 3.1. *$s : \mathcal{C} \to N$ is a size function if* (i) *$s(D) \geq 1$ for every $D \in \mathcal{C}$ and* (ii) *D included in E implies $s(D) \leq s(E)$ (the inequality being strict when the inclusion is proper).*

We wish to analyze the following type of recursive algorithms A, which return a yes/no answer represented by 1/0. The algorithm divides the input data structures D and E into n and m parts, respectively, where each part is also from the same class of data structures as D and E.

```
ALGORITHM A(D, E) /* D, E ∈ C. */
if s(D) ≤ threshold or s(E) ≤ threshold then /* threshold ≥ 2 is a constant. */
  B(D, E)
else
  begin
    for each pair (D_i, E_j), i ∈ [n], j ∈ [m] do A(D_i, E_j);
    /* The D_i's are included in D, and the E_j's are included in E. */
    Form an n × m matrix M with M_ij = A(D_i, E_j)
    return C(M)
  end
```

Let the time complexity of algorithm C be $O(h(n,m))$ when an $n \times m$ matrix is the input to C. We first consider the case where h is a biadditive function on the nonnegative reals. Suppose that $\sum_{i=1}^{n} s(D_i) \leq s(D) - 1$, $\sum_{j=1}^{m} s(E_j) \leq s(E) - 1$, and algorithm B takes a constant amount of time when the size of one of its inputs is at most a constant called *threshold*. Then we can show that the time complexity of algorithm A is $O(h(s(D), s(E)) + s(D)s(E))$.

Note that algorithm A needs the size of D, E, and their substructures on recursive calls. The determination of these sizes is the preprocessing cost of algorithm A. We will assume throughout this section that the preprocessing cost of the algorithm denoted T_{pre}, is bounded from above by the cost of the algorithm itself, denoted T, i.e., $T_{\text{pre}}(s(D), s(E)) \leq T(s(D), s(E))$, which holds in all of the applications that we discuss in this paper. Thus the total cost $T_{\text{pre}} + T \leq 2T$, so T will also stand for the total cost. This assumption will not be stated in our theorems.

3.1. Biadditive *h*. The following theorem essentially states that if the complexity of algorithm C is a biadditive function h of the input sizes, then the complexity of algorithm A is also the same function h.

THEOREM 3.2. *If*

1. *identification of each pair* (D_i, E_j) *takes* $O(1)$ *time,*

2. *algorithm* B *takes* $O(1)$ *time when the size of one of its inputs does not exceed the threshold,*

3. *the time complexity of algorithm* C *is* $O(h(n,m))$ *for an* $n \times m$ *input matrix, where* h *is biadditive,*

4. s *is a size function, and the sum of the substructure sizes on the recursive calls is less than the parent structure size,*

then the time complexity of algorithm $A(D,E)$ *is* $O(h(s(D), s(E)) + s(D)s(E))$.

Proof. We assume that comparing a number with a constant takes a constant of time. A simple proof by induction shows that the total number of recursive calls is less than $s(D)s(E)$. Therefore, by our assumption and hypothesis 1, the cost of all comparisons and identification of the substructure pairs for the recursive calls over all the recursive calls is $O(s(D)s(E))$. We now show that the cost of the remaining steps is as claimed above, thus completing the proof. The proof is by induction on $s(D)$ and $s(E)$. For the induction to succeed, we need to prove something slightly stronger, viz, the complexity of $A(D,E)$ is $O(h((s(D)-1),(s(E)-1)) + (s(D)-1)(s(E)-1))$.

Basis. If either $s(D)$ or $s(E)$ is equal to *threshold* and both are at least two, then the statement is trivially true by choosing the constant, say c, in the O notation large enough and at least equal to $\max(c_1, c_2)$, where c_1 is the positive constant in the time complexity for setting up matrix M and c_2 is the constant in the time complexity of algorithm C.

Induction step. The time complexity of $A(D,E)$ is bounded above by the time for the recursive calls, plus the time c_1nm for setting up the matrix M, and the time taken by algorithm C on an $n \times m$ input matrix. Therefore, by the inductive hypothesis, the complexity of $A(D,E)$ is bounded by

$$B = \sum_{i=1}^{n} \sum_{j=1}^{m} c[h((s(D_i)-1),(s(E_j)-1)) + (s(D_i)-1)(s(E_j)-1)] + ch(n,m) + c_1nm,$$

where c is the positive constant chosen above such that $c \geq c_1$ and $c \geq c_2$. Since h is biadditive,

$$\begin{aligned} B &\leq ch\left(\sum_{i=1}^{n}(s(D_i)-1), \sum_{j=1}^{m}(s(E_j)-1)\right) + c(s(D)-n-1)(s(E)-m-1) \\ &\quad + ch(n,m) + c_1 nm \\ &= ch\left(\sum_{i=1}^{n}(s(D_i)-1), \sum_{j=1}^{m}(s(E_j)-1)\right) + ch(n,m) + c(s(D)-1)(s(E)-1) \\ &\quad - cns(E) - cms(D) + (c+c_1)nm + c(n+m). \end{aligned}$$

Further, since $\sum_{i=1}^{n} s(D_i) \leq s(D) - 1$ and all of the $s(D_i)$'s are at least one (s is a size function), it follows that $s(D) > n$. Similarly, $s(E) > m$. Also, since $c \geq c_1$, we have $(c+c_1)nm + c(n+m) \leq cns(E) + cms(D)$. Therefore,

$$B \leq ch(s(D)-n-1, s(E)-m-1) + ch(n,m) + c(s(D)-1)(s(E)-1)$$

Again, by the additivity of h, we have

$$B \leq ch(s(D)-1, s(E)-1) + c(s(D)-1)(s(E)-1).$$

This completes the induction step. Now since h is nondecreasing in both arguments, $h(s(D)-1, s(E)-1) \leq h(s(D), s(E))$ and therefore $T(s(D), s(E)) = O(h(s(D), s(E)) + s(D)s(E))$. □

Remark 3.3. Note that it is sufficient for h to be biadditive over the positive integers. Furthermore, it is not necessary that h be biadditive for all positive n and m. It is sufficient that h be biadditive almost everywhere, i.e., with at most a finite number of (positive integral) exceptions. Also, it is possible to weaken the first two hypotheses by requiring that the time taken in each is $O(h(s(D), s(E))/(s(D)s(E)))$ instead of $O(1)$.

The following proposition merely states that the complexity of algorithm A cannot be reduced by using a more time consuming algorithm C.

PROPOSITION 3.4. *Let T and T' denote the time complexities of algorithm A when the time complexities of algorithm C are $ch(m,n)$ and $ch'(m,n)$ $(c > 0)$, respectively, and suppose that $h(m,n) \leq h'(m,n)$ for all m and n. Then for all values of $s(D)$ and $s(E)$, $T(s(D), s(E)) \leq T'(s(D), s(E))$.*

Proof. The proof is by induction on $s(D)$ and $s(E)$. □

3.2. h is additive in one argument only. We now consider the case when h is not biadditive, but $h(x,y)$ is additive in only one argument, say x. Additivity on the second argument, y, is handled similarly. Other things being equal, we prove that the time complexity of algorithm $A(D,E)$ is $O(h(s(D), f(s(E))) + s(D)s(E))$ if there is an additive function f such that $h(x, f(y))$ is biadditive and $h(x,y) \leq h(x, f(y))$ for all x and y.

THEOREM 3.5. *If all assumptions of Theorem* 3.2 *hold except that the time complexity of algorithm C for an $n \times m$ input matrix is $O(h(n,m))$, where h is additive in one argument only, say m, then the time complexity of algorithm $A(D,E)$ is $O(h(f(s(D)), s(E)) + s(D)s(E))$ provided there exists additive function f such that $h(f(x), y)$ is biadditive and $h(x,y) \leq h(f(x), y)$ for all x and y.*

Proof. The proof is similar to that of Theorem 3.2, except for minor changes and one extra step, where, using the additivity of f, we push the summation from outside f to inside, i.e., $\ldots \sum_{i=1}^{n} f(s(D_i)) \ldots$ to $\ldots f(\sum_{i=1}^{n} s(D_i)) \ldots$. □

3.3. h is additive in neither argument. Next, suppose that h is not additive in either argument. In this case, we prove that the time complexity of algorithm $A(D, E)$ is $O(h(f(s(D)), g(s(E))) + s(D)s(E))$ if there are additive functions f and g such that $h(f(x), g(y))$ is biadditive and $h(x, y) \leq h(f(x), g(y))$ for all x, y.

THEOREM 3.6. *If all assumptions of Theorem* 3.2 *hold except that the time complexity of algorithm C for an $n \times m$ input matrix is $O(h(n, m))$, where h is additive in neither argument, then the time complexity of algorithm $A(D, E)$ is $O(h(f(s(D)), g(s(E))) + s(D)s(E))$, provided there exist additive functions f and g such that $h(f(x), g(y))$ is additive and $h(x, y) \leq h(f(x), g(y))$ for all x and y.*

Proof. The proof is similar to that of Theorem 3.5 except for minor changes. □

Clearly the time complexity of algorithm $A(D, E)$ is $O(s(D)s(E))$ if $h(n, m) = O(nm)$. Also, if $h(n, m) \leq h'(n, m)$ and $h'(n, m)$ is biadditive, then algorithm A takes $O(h'(s(D), s(E)) + s(D)s(E))$ time by Proposition 3.4 and Theorem 3.2. With these results, we can often handle functions that are not biadditive and do not become biadditive for any additive functions f and g applied to n and m. Another useful generalization is the following theorem, which gives a biadditive bound on the complexity of algorithm A when the complexity of algorithm C is a pseudoconvex function.

THEOREM 3.7. *If all assumptions of Theorem* 3.2 *hold except that the time complexity of algorithm C for an $n \times m$ input matrix is $O(h(n, m))$, where h is a continuous pseudoconvex function, then the time complexity of algorithm $A(D, E)$ is $O(h(s(D), s(E)) + s(D)s(E)(1 + h(0, 0)) + s(D)h(0, s(E)) + s(E)h(s(D), 0))$.*

Proof. For $n \geq 1$ and $m \geq 1$, define $\phi(n, m) = h(m, n) + (n - 1)h(m, 0) + (m - 1)h(0, n) + (n - 1)(m - 1)h(0, 0)$. A simple computation using the properties of pseudoconvex functions given in Lemma 2.5 shows that ϕ is biadditive for all $n \geq 1$ and $m \geq 1$. Clearly, $\phi(n, m) \geq h(n, m)$ for all $n \geq 1$ and $m \geq 1$ since h is nonnegative. Now apply Proposition 3.4 and Theorem 3.2. Note that the theorem can also be proved from scratch by an induction similar to that of Theorem 3.2 using only the properties of pseudoconvex functions, but the proof is longer and involves more computations. □

3.4. Recurrence (2). Now we examine the case when the merging process requires more time. Specifically, we consider the case when the merging process takes time that depends on the sizes of D and E as well, i.e., $O(h(s(D), s(E)) + mn)$ time instead of $O(h(m, n) + mn)$ time.

THEOREM 3.8. *If all assumptions of Theorem* 3.2 *hold except that the time complexity of merging solutions is $O(h(s(D), s(E)) + mn)$, where h is biadditive, then the time complexity of algorithm $A(D, E)$ is $O((s(D) + s(E))h(s(D), s(E)) + s(D)s(E))$.*

Proof. The proof is similar to that of Theorem 3.2. □

If h is pseudoconvex, then the time complexity of algorithm A is $T(s(D), s(E)) = O((s(D) + s(E))\phi(s(D), s(E)) + s(D)s(E))$, where ϕ is given in the proof of Theorem 3.7.

4. Necessary and sufficient conditions. We now give relevant (some sufficient and some necessary and sufficient) conditions that ensure the additivity of a unary function, the biadditivity of a binary function, and pseudoconvexity.

LEMMA 4.1. *A unary function f is additive on the nonnegative reals if it satisfies one of the following conditions:* 1. *$f(x) = xg(x)$, where g is a monotonically increasing function.* 2. *f is twice differentiable, $f'' \geq 0$, and $f(0) = 0$.*

Proof. 1. A simple computation suffices for 1.

2. Note that if f'' is nonnegative, f' is nondecreasing. Therefore, $f'(t) \leq f'(t+y)$ for all $t, y \geq 0$. Integrating both sides, we have $\int_0^x f'(t)dt \leq \int_0^x f'(t+y)dt$, or $f(x) - f(0) \leq f(x+y) - f(y)$. Since $f(0) = 0$, we have $f(x) + f(y) \leq f(x+y)$. Since $f(0) + f(x) \leq f(0+x) = f(x)$, the requirement $f(0) = 0$ is also necessary for nonnegative f. If f is twice differentiable, then the requirement $f'' \geq 0$ is also necessary. □

Examples. Some examples of additive functions are as follows: for any $c \geq 0$, $cx(\log(x+1))^k$ $(k \geq 0)$, cx^u for all $u \geq 1$, etc.

LEMMA 4.2 ([12]). *A twice-differentiable unary function f is convex on the nonnegative reals iff $f'' \geq 0$.*

THEOREM 4.3. *A twice-differentiable binary function f is pseudoconvex iff $f''_{xx} \geq 0$ and $f''_{yy} \geq 0$ (the second partial derivatives of f with respect to x and y, respectively).*

Proof. The proof follows from the definition of a pseudoconvex function and the above lemma. □

Examples. $x + y$, xy, and $(x-y)^2$ are some examples of pseudoconvex functions.

THEOREM 4.4.

1. *$h(x,y) = f(x)g(y)$ is biadditive (pseudoconvex) if f and g are additive (convex) functions.*
2. *$h(x) = (f(x))^u$ is additive for $u \geq 1$ if f is additive.*
3. *For $p \geq 2$, $h(x,y) = p^{f(x)+g(y)}$ is biadditive for $x \geq 2$ and $y \geq 2$ if f and g are additive and $f(x) \geq 1$ and $g(y) \geq 1$ for all x and y.*
4. *$h(x,y) = xyg(x,y)$ is biadditive if $g(x,y)$ is a nondecreasing function of x for every y and of y for every x.*
5. *$h(x,y)$ is biadditive if $h(x,y)$ is pseudoconvex and $h(x,0) = 0$ for all x and $h(0,y) = 0$ for all y.*

Proof. 1. We verify only the biadditivity part. $h(u,x) + h(v,x) = f(u)g(x) + f(v)g(x) = (f(u) + f(v))g(x) \leq f(u+v)g(x) = h(u+v,x)$ by the additivity of f. Similarly, $h(x,u) + h(x,v) \leq h(x,u+v)$.

2. $h(x) + h(y) = (f(x))^u + (f(y))^u \leq (f(x) + f(y))^u$ since $u \geq 1$. Also, $(f(x) + f(y))^u \leq (f(x+y))^u = h(x+y)$ since f is additive and u is positive.

3. The proof follows from $x + y \leq xy$ for $x \geq 2$ and $y \geq 2$. $h(u,x) + h(v,x) = p^{f(u)+g(x)} + p^{f(v)+g(x)} = (p^{f(u)} + p^{f(v)})p^{g(x)} \leq p^{f(u)}p^{f(v)}p^{g(x)} = p^{f(u)+f(v)}p^{g(x)} \leq p^{f(u+v)+g(x)} = h(u+v,x)$. Similarly, $h(x,u) + h(x,v) \leq h(x,u+v)$.

4. A simple computation suffices.

5. The proof follows from the addition inequalities of a pseudoconvex function. □

5. One-dimensional recurrences. Let $T(x) = \sum_{i=1}^{k} a_i T(x/c_i) + f(x)$ for all reals $x > K$, $T(x) = b$ for all reals $1 \leq x \leq K$ for some real constants $a_i \geq 1$, $c_i > 1$ for $1 \leq i \leq k$, and $b > 0$, and function f be defined on the nonnegative reals. Also, let $K \geq \max_i\{c_i\}$ be an integer. In many applications, T is defined only for integral values using flooring and ceiling operations. The reason for defining T for all real values is that our analysis can easily be extended to the integral case. These details can be filled in easily.

We need the principle of noetherian induction for our proofs. Let $Q = \{x \in R \mid x \geq 1\}$. We define the relation R on Q by xRy iff $y > K$ and $x = y/c_i$ for some c_i in the recurrence given above (i.e., $T(x)$ appears on the right-hand side of the recurrence for $T(y)$). Let R^+ denote the transitive closure of R. Clearly, R is noetherian, i.e., there are no infinite descending chains in R. Let P be any predicate

on Q. We say that P is *R-complete* iff $\forall y \in Q[\forall x$ such that $xR^+yP(x)] \Rightarrow P(y)$. Our interest in noetherian relations is because of the following *principle of noetherian induction*. Let R be a noetherian relation and P be a R-complete predicate; then $\forall x \in Q\ P(x)$.

DEFINITION 5.1. *Let function g be given. We say that a function f is g-*star-shaped *iff for all $x \geq 1$ and $0 < t < 1$, $f(tx) \leq g(t)f(x)$. We say that f is g-co-star-shaped iff for all x and $0 < t < 1$, $f(tx) \geq g(t)f(x)$.*

The following theorem gives a tight bound on $T(x) = \Theta(f(x))$ if f is g-star-shaped and a certain sum of the subproblem sizes is smaller than 1 (which, for lack of a better term, we call the g weighted sum of the suproblem-size fractions). For convenience, we introduce the following notation.

Notation. Given function g, let $S(g,T)$ denote $\sum_{i=1}^{k} a_i g(1/c_i)$, where the a_i's and c_i's are as in the recurrence for T.

THEOREM 5.2. *If $f(x) \geq d$ over $[1,K]$ for some $d > 0$, there exists g such that f is g-star-shaped, and $S(g,T) < 1$, then $T(x) = \Theta(f(x))$.*

Proof. Clearly, $T(x) = \Omega(f(x))$. Therefore, it suffices to show that $T(x) = O(f(x))$. The proof is by noetherian induction. Choose $C = \max\{b/d, 1/(1 - S(g,T))\}$.

Basis. All the reals in $[1,K]$ are minimal with respect to R. The statement of the theorem is trivially true for all of these minimal elements since by our choice of C, $b \leq (b/d)d \leq Cf(x)$.

Induction step. Suppose $y > K$. By definition, $T(y) = \sum_{i=1}^{k} a_i T(y/c_i) + f(n)$. By definition of R, $(y/c_i)\ R\ y$. Therefore, combining the recurrence for $T(y)$ with our induction hypothesis for the $T(y/c_i)$'s, we have

$$\begin{aligned} T(y) &\leq \sum_{i=1}^{k} a_i C f(y/c_i) + f(y) \\ &\leq C \sum_{i=1}^{k} a_i g(1/c_i) f(y) + f(y) \\ &\leq Cf(y) \left(1/C + \sum_{i=1}^{k} a_i g(1/c_i) \right), \end{aligned}$$

which is at most $Cf(y)$ by our choice of C. □

Next, we consider the case when f is of the form $h(x)(\log x)^l$ for some h and $l \geq 0$. Here we give (i) an upper bound on $T(x)$ of $O(f(x)\log x)$ when h is g-star-shaped and the g weighted sum of the subproblem-size fractions is equal to 1 and (ii) a lower bound of $\Omega(f(x)\log x)$ when h is g-co-star-shaped and the g weighted sum of the subproblem-size fractions is at least 1.

THEOREM 5.3.

1. *If $f(x) = h(x)(\log x)^l + d$ $(x \geq 1)$ for some $l \geq 0$, $d > 0$, there is a g such that h is g-star-shaped, and $S(g,T) = 1$, then $T(x) = O(f(x)\log x)$.*

2. *If $f(x) = h(x)(\log x)^l$ $(x \geq 1,\ l \geq 0)$, there exists g such that h is g-co-star-shaped, and $S(g,T) \geq 1$, then $T(x) = \Omega(f(x)\log x)$.*

Proof.

1. First, we prove that $T(x) \leq Cf(x)(1 + \log_m x)$ for every $x \geq 1$. Here $m = \min_i\{c_i\}$, which is clearly greater than 1. Again, we use noetherian induction. Without loss of generality, we may assume that $f(x) = h(x)(\log_m x)^l + d$. Choose $C = \max\{1, b/d\}$.

Basis. The upper bound on $T(x)$ holds trivially for all reals in $[1, K]$ by our choice of C.

Induction step. Let $y > K$. Then $T(y) = \sum_{i=1}^{k} a_i T(y/c_i) + f(y)$. By our induction hypothesis for the $T(y/c_i)$'s, we have

$$\begin{aligned}
T(y) &\le \sum_{i=1}^{k} a_i C f(y/c_i)(1 + \log_m(y/c_i)) + f(y) \\
&= \sum_{i=1}^{k} a_i C(h(y/c_i)(\log_m(y/c_i))^l + d)(1 + \log_m(y/c_i)) + f(y) \\
&\le C \sum_{i=1}^{k} a_i g(1/c_i)(h(y)(\log_m y)^l + d)(1 + \log_m(y/c_i)) + f(y) \\
&\le C f(y) \sum_{i=1}^{k} a_i g(1/c_i)(1 + \log_m(y/m)) + f(y) \\
&\le C f(y)(1 + \log_m y)
\end{aligned}$$

since $S(g, T) = 1$ and $C \ge 1$. Now $Cf(x) \log_m x \ge Cf(x)$ for $x > m$, so $T(x) = O(f(x) \log_m x) = O(f(x) \log\ x)$.

2. Let $f(x) = h(x)(\log_p x)^l$ and let D be any constant such that $f(x) \le D$ over $[1, K]$. A similar proof by induction shows that $T(x) \ge Cf(x) \log_M x$, where $M = p(\max_i\{c_i\})$ and $C = \min\{1/S(g, T), b/(D \log_M K)\}$. □

By combining the two conditions of the above theorem, we have the following corollary, which gives a $\Theta(f(x) \log x)$ bound for T.

COROLLARY 5.4. *If $d \le f(x)$ over $[1, K]$ for some $d > 0$, $f(x) = \Theta(h(x)(\log\ x)^l)$ $(l \ge 0)$, there exists g such that $h(tx) = g(t)h(x)$ for all x and $0 < t < 1$, and $S(g, T) = 1$, then $T(x) = \Theta(f(x) \log\ x)$.*

The following useful proposition is easy to prove by noetherian induction.

PROPOSITION 5.5. *Let $f_1(x) \le f(x) \le f_2(x)$ for all $x \ge 1$ and let T_1, T, and T_2 denote the solutions to recurrences of form 3 corresponding to f_1, f, and f_2, respectively (initial conditions remain the same). Then for all $x \ge 1$, $T_1(x) \le T(x) \le T_2(x)$.*

The following theorem is useful when for every function g such that f is g-star-shaped, the g weighted sum of the subproblem-size fractions exceeds 1. Roughly speaking, we try to find a function F that dominates f and satisfies conditions similar to the ones imposed on f in the above theorems.

THEOREM 5.6. *Let F be any function such that $f(x) \le F(x)$ for all $x \ge 1$ and for some d $F(x) \ge d > 0$ over $[1, K]$.*

1. *$T(x) = O(F(x))$ if there exists G such that F is G-star-shaped and $S(G, T) < 1$.*

2. *$T(x) = O(F(x))$ if there is a $c > 0$ such that $f(x) + c \le F(x)$ for all $x \ge 1$, there exists G such that F is G-star-shaped, $S(G, T) \le 1$ (note the $\le$ sign as opposed to $<$), and there is a g such that f is g-co-star-shaped with $S(g, T) > 1$.*

3. *$T(x) = \Omega(F(x))$ if there exists G such that F is G-co-star-shaped and $S(G, T) \ge 1$.*

Proof.

1. Let $T'(x) = \sum_{i=1}^{k} a_i T'(x/c_i) + F(x)$ and apply Theorem 5.2 to T' and then Proposition 5.5 to get $T(x) = O(F(x))$.

2. Choose $C = \max\{b/c, 1/(S(g,T)-1)\}$. We need to prove something stronger, viz, $T(x) \le C(F(x) - f(x))$.

Basis. For $x \in [1, K]$, we have $T(x) = b \le Cc \le C(F(x) - f(x))$.

Induction step. By our induction hypotheses for the $T(y/c_i)$'s, we have

$$T(y) \le \sum_{i=1}^{k} a_i C(F(y/c_i) - f(y/c_i)) + f(y).$$

Since F is G-star-shaped and f is g-co-star-shaped, we have

$$\begin{aligned} T(y) &\le CF(y) \sum_{i=1}^{k} a_i G(1/c_i) - f(y) \left(C \sum_{i=1}^{k} a_i g(1/c_i) - 1 \right) \\ &\le C(F(y) - f(y)) \end{aligned}$$

by our choice of C and the assumption that $S(G,T) = \sum_{i=1}^{k} a_i G(1/c_i) \le 1$.

3. The proof is straightforward and follows by noetherian induction. □

We now show that the popular theorem on the analysis of divide-and-conquer algorithms with equal parts [1, 3, 5, 14], [7, p. 62] emerges as a special case of our theorems above.

COROLLARY 5.7. *Let $\epsilon > 0$ and $q = \log_c a$. The solution to $T(x) = aT(x/c) + f(x)$ $(a, b > 0,\ c > 1,\ T(x) = b$ for $x \le c)$ is* (i) $T(x) = \Theta(x^q)$ *if* $f(x) = O(x^{q-\epsilon})$, (ii) $T(x) = \Theta(f(x) \log_c x)$ *if* $f(x) = \Theta(x^q (\log_c x)^l)$, *and* (iii) $T(x) = \Theta(f(x))$ *if* $f(x) = \Omega(x^{q+\epsilon})$.

Proof. (iii) By Proposition 5.5, it suffices to consider $f(x) = dx^p$ $(d > 0,\ p \ge 0)$ and $g(t) = t^p$. Then $f(tx) = d(tx)^p = g(t)f(x)$. If $q < p$, then $ag(1/c) = a(1/c)^p = c^{q-p} < 1$. Therefore, by Theorem 5.2, we have $T(x) = \Theta(x^p)$. (i) Again, it is sufficient to let f be as in part (iii). If $p < q$, then $ag(1/c) > 1$. Let $F(x) = (d+1)x^q$ so that there is an $e > 0$ such that $f(x) + e \le F(x)$. F is G-star-shaped, where $G(t) = t^q$, and $aG(1/c) = 1$. Since f is g-co-star-shaped and $ag(1/c) > 1$, by Theorem 5.6, we have $T(x) = \Theta(x^q)$. (ii) The proof follows from Theorem 5.3. □

Note that Brassard and Bratley [5] have a slight generalization of the above corollary, which is a consequence of our work reported in [22].

6. Applications. We present some applications of the above theorems.

COROLLARY 6.1. *Let n and m denote the sizes of the two rooted trees, $n \le m$. The following problems can be solved in $O(nm^{1.5}/\log\ m)$ time:*

1. *rooted subtree isomorphism* [25];
2. *AC matching of linear terms* [23];
3. *rooted subgraph homeomorphism on trees* [6].

Proof. The function $h(x,y) = cxy^{1.5}/\log\ y$ is biadditive for every constant $c \ge 0$ (part 1 of Theorem 4.4). Since the algorithms for all of these problems are of the form given above with an algorithm for bipartite maximum matching in place of C, and since there is a bipartite matching algorithm of complexity $O(rs^{1.5}/\log\ s)$ [8], where $r \le s$ are the sizes of the vertex sets, by Theorem 3.2 we have the stated result. In each case, all of the assumptions of the theorem are satisfied. □

Remarks. Clearly, this corollary does not follow from Theorem 1.1 given above. Moreover, Theorem 1.1 is a corollary of Theorem 3.2.

COROLLARY 6.2. *Associative matching of linear terms can be done in $O(nm)$ time, where n and m are the sizes of the input trees representing the terms.*

Proof. The function $h(x, y) = cxy$ is biadditive for every constant $c \geq 0$. Since there is an ordered bipartite matching (see [23] for the definition of ordered bipartite matching) algorithm of time complexity $O(rs)$, where $r < s$ are the sizes of the vertex sets, and since the algorithm for associative matching of linear terms is of the form given above [23], by Theorem 3.2 we have the stated result. □

Remark. This improves the bound given by Verma and Ramakrishnan [23] by a factor of log n, and the corollary below improves Theorem 1.2 proved by Reyner in [20] by a factor of $\ln n$.

COROLLARY 6.3. *If there is a bipartite matching algorithm of time complexity $O(rs)$, where $r < s$ are the sizes of the vertex sets, then the algorithms for rooted subtree isomorphism, AC matching of linear terms, and rooted subgraph homeomorphism on trees are of time complexity $O(nm)$.*

COROLLARY 6.4. *There are* parallel *algorithms for rooted subtree isomorphism, AC matching of linear terms, and rooted subgraph homeomorphism on trees of time complexity $O(nm)$.*

Proof. The proof follows from Corollary 6.3 and the parallel bipartite matching algorithms of time complexity better than $O(rs)$ [9, 10]. Note that everything else in algorithm A is being done sequentially except for a parallel algorithm for C. (This means that faster algorithms using more processors can be designed.) □

COROLLARY 6.5.

1. *The Select algorithm of Blum et al.* [4] *is of time complexity $O(n)$, where n is the input size.*

2. *The solution to $T(n) \leq T(n/2) + T(n/4) + cn^a$, $c > 0$ [17], where $a < b = \log_2(1 + \sqrt{5}) - 1$, is $O(n^b)$.*

Proof.

1. The recurrence for the algorithm is $T(x) = T(x/5) + T(3x/4) + cx$ for some $c > 0$. Here $f(x) = cx$. Let $g(t) = t$; then $f(tx) = ctx \leq g(t)f(x)$ and $S(g, T) = \sum_i a_i g(1/c_i) = 1(1/5) + 1(3/4) < 1$. Therefore, by Theorem 5.2, we have the stated result.

2. The proof follows from part 2 of Theorem 5.6. □

7. Conclusion. In this paper, we have presented some general techniques and master theorems for three kinds of recurrences frequently occurring in the analysis of divide-and-conquer algorithms. Much of the existing work on the recurrences considered here is subsumed by our results. In the process, we established interesting connections between some elegant mathematics and the analysis of recurrences. We then gave several applications of our theorems, thus improving existing bounds in the literature for several problems. This paper is an invitation to an exciting area for further research, viz, general techniques and theorems for other frequently occurring recurrences, which is obviously of considerable importance.

Acknowledgments. The author thanks the referees for their constructive comments and suggestions. After this paper was submitted for publication, P. Kilpelainen sent me a copy of his thesis [16], which contains a weaker form of Theorem 3.2. Subsequently, we also received a paper [26] from E. Reingold which analyzes a different (min-max) recurrence. Thanks also go to both of them.

REFERENCES

[1] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[2] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, *Computing a maximum cardinality matching in a bipartite graph in time* $O(n^{1.5}\sqrt{m/logn})$, Inform. Process. Lett., 37 (1991), pp. 237–240.

[3] J. Bentley, D. Haken, and J. Saxe, *A general method for solving divide-and-conquer recurrences*, SIGACT News, 12 (1980), pp. 36–44.

[4] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1972), pp. 448–461.

[5] G. Brassard and P. Bratley, *Algorithmics: Theory and Practice*, Prentice–Hall, Englewood Cliffs, NJ, 1988.

[6] M. Chung, $O(n^{2.5})$ *time algorithms for the subgraph homeomorphism problem on trees*, J. Algorithms, 8 (1987), pp. 106–112.

[7] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press/McGraw–Hill, Cambridge, MA, New York, 1990.

[8] T. Feder and R. Motwani, *Clique partitions, graph compression and speeding-up algorithms*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1991, pp. 123–133.

[9] H. Gabow and R. Tarjan, *Almost-optimal speedups of algorithms for matching and related problems*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1988, pp. 514–527.

[10] A. Goldberg, S. Plotkin, and P. Vaidya, *Sublinear-time parallel algorithms for matching and related problems*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 174–185.

[11] D. Greene and D. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, 1982.

[12] G. Hardy, J. Littlewood, and G. Polya, *Inequalities*, Cambridge University Press, Cambridge, UK, 1952.

[13] J. Hopcroft and R. Karp, *An* $n^{5/2}$ *algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[14] P. Purdom Jr. and C. Brown, *The Analysis of Algorithms*, Oxford University Press, Oxford, UK, 1985.

[15] R. Karp, *Probabilistic recurrence relations*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1991, pp. 190–197.

[16] P. Kilpelainen, *Tree matching problems with applications to databases*, Ph.D. thesis, University of Helsinki, Helsinki, 1992.

[17] M. van Kreveld, M. Overmars, and P. Agarwal, *Intersection queries in sets of disks*, in Proc. Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comput. Sci. 447, Springer-Verlag, Berlin, 1990, pp. 393–403; BIT, 32 (1992), pp. 268–279.

[18] D. Mitrinovic, *Analytic Inequalities*, Springer-Verlag, Berlin, 1970.

[19] L. Monier, *Combinatorial solutions of multidimensional divide-and-conquer recurrences*, J. Algorithms, 1 (1980), pp. 60–74.

[20] S. W. Reyner, *An analysis of a good algorithm for the subtree problem*, SIAM J. Comput., 6 (1977), pp. 730–732.

[21] R. M. Verma, *An error in Reyner's "An analysis of a good algorithm for the subtree problem,"* Technical Report 88/03, Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY, 1988.

[22] R. M. Verma, *A general method and a master theorem for divide-and-conquer recurrences with applications*, J. Algorithms, 16 (1994), pp. 67–79.

[23] R. M. Verma and I. V. Ramakrishnan, *Tight complexity bounds for term matching problems*, Inform. and Comput., 101 (1992), pp. 33–69.

[24] R. M. Verma and I. V. Ramakrishnan, *Some complexity theoretic aspects of AC Rewriting*, in Proc. Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 349, Springer-Verlag, Berlin, 1989, pp. 407–420.

[25] R. M. Verma and S. W. Reyner, *An analysis of a good algorithm for the subtree problem, corrected*, SIAM J. Comput., 18 (1989), pp. 906–908.

[26] L. Zhiyuan and E. Reingold, *Solution of a divide-and-conquer maximin recurrence*, SIAM J. Comput., 18 (1989), pp. 1188–1200.

SIAM J. COMPUT.
Vol. 26, No. 2, pp. 582–603, March 1997

013

POTENTIALS IN UNDIRECTED GRAPHS AND PLANAR MULTIFLOWS*

ANDRÁS SEBŐ†

Abstract. The duality relation between shortest paths and potentials in *directed graphs* and the significance of both of these in the theory of network flows is well known. In this paper, we work out the analogous *undirected* notions, which neither are contained in nor contain their directed counterpart. They are more related to matching theory than to network flows: the corresponding min-path-max-potential theorem can be considered a weighted generalization of the Gallai–Edmonds structure theorem for matchings.

In our earlier work [*J. Combin. Theory Ser. B*, 49 (1990), pp. 10–39], the corresponding theorems are proved in the special case of ±1 bipartite weightings, and this special case already contains the main points of the general proof. The goal of the present paper is to extrapolate from this ±1-weighted bipartite special case the arbitrarily weighted general min-path-max-potential theorem and to show some algorithmic consequences related to planar multiflows, the Chinese postman problem, the weighted and unweighted matching structure, etc. In order to make this paper self-contained, we also include a compact, revised variant of earlier proofs, adapted to the present context. In addition to good characterization theorems and polynomial algorithms, efficient (logarithmic polynomial) parallel algorithms follow for some of these problems.

Key words. T-joins, T-cuts, multicommodity flows, Chinese postman, matching, structure, parallel algorithm

AMS subject classifications. 05C38, 05C45, 90B10

PII. S0097539790186704

1. Introduction.

In this section, we explain the background of the paper and introduce the main tools that we will use. In particular, we present potentials in the ±1-weighted bipartite special case, which was developed in Sebő [1990]. Since this constitutes the kernel of our results, we fully include a compact proof of the main theorem concerning this case.

Then in section 2, we define potentials in arbitrary weighted undirected graphs and prove (extrapolate from the bipartite special case) a minimax theorem on minimum weight paths and maximum potentials. In section 3, we point out the algorithmic consequences of our results and apply them to, for example, planar multiflows.

If G is a graph and $w : E(G) \to \mathbb{R}$, define the *distance* of $x, y \in V(G)$ as

$$\lambda_{G,w}(x,y) = \lambda_w(x,y) = \lambda(x,y) = \min\{w(P) : P \text{ is an } (x,y) \text{ path}\}.$$

In this paper, *paths* are considered to be sets of edges, or subgraphs. (For instance, $V(P)$ will denote the set of vertices of the path P.) They can have a repetition of vertices, but no repetition of edges is allowed. An (x,y) path is a path whose endpoints are $x, y \in V(G)$. The definition of $\lambda_{G,w}(x,y)$ is meant to be ∞ if x and y are not in the same component of G.

A path without repetition of vertices will be called *simple*. If the two endpoints of a (simple) path coincide, it is a *cycle* (*circuit*). $w(P)$ denotes the sum $\sum_{e\in P} w(e)$. A *shortest* (*w-shortest*) *path* is an (a,b) path P with $w(P) = \lambda_w(a,b)$. If $a, b \in V(P)$,

* Received by the editors August 20, 1990; accepted for publication (in revised form) June 21, 1995. This research was partly supported by the Alexander von Humboldt Foundation and project TEMPRA, Région Rhône-Alpes.

http://www.siam.org/journals/sicomp/26-2/18670.html

† CNRS, LEIBNIZ-IMAG, 46 Avenue Félix Viallet, 38031 Grenoble cedex 1, France (andras.sebo@imag.fr).

$P(a,b)$ denotes a simple subpath of P joining a and b. (If P is simple, $P(a,b)$ is uniquely determined.) For $X \subseteq V(G)$, $\delta(X)$ will denote the set of edges with exactly one endpoint in X. We will also use the notation $E^- := \{e \in E(G) : w(e) < 0\}$, $E^+ := \{e \in E(G) : w(e) > 0\}$.

A graph G with a weighting $w : E(G) \to \mathbb{R}$ is called *conservative* if $w(C) \geq 0$ for every circuit $C \subseteq E(G)$. Conservative graphs are characterized as follows.

(1.1) *(G,w) is conservative if and only if for all $x, y \in V(G)$, $\lambda_w(x,y) = \min\{w(P) :$ P is a simple (x,y) path$\}$.*

In particular, for connected graphs, $\lambda_w(x,y)$ is finite for all $x, y \in V(G)$.

Recall that $\lambda_w(x,y)$ can be computed via matching techniques in various well-known ways (see Edmonds [1965a] or Lawler [1976]); one of these will be explained in section 3. On the other hand *it cannot be reduced to the well-known shortest-path algorithms* because the two directed edges corresponding to an undirected edge of negative weight constitute a negative directed cycle; moreover, subpaths of shortest paths are not necessarily shortest and distances do not satisfy the triangle inequality. Thus the notion of potentials and the related theory are also different in the undirected case.

The behavior of undirected potentials is determined by the following theorem, as will be explained in section 2:

If (G,w) is conservative and $x_0 \in V(G)$, we call each set $V^i = V^i(\lambda) := \{x \in V(G) : \lambda_w(x_0,x) \leq i\}$ $(i = 0, \pm 1, \pm 2, \ldots)$ a *level set* of (G,w,x_0); we denote by G^i the graph induced by V^i, and we call it the *level graph*. If the weights are ± 1, then edges go between neighboring levels; that is, we have the following.

(1.2) *If $w : E(G) \to \{-1,1\}$ is conservative, then for all $xy \in E(G)$, $|\lambda(x_0,x) - \lambda(x_0,y)| \leq 1$. If, in addition, G is bipartite, then this inequality is satisfied with equality for every edge.*

Indeed, we can assume without loss of generality that $\lambda(x_0,x) \geq \lambda(x_0,y)$. Let P be a simple shortest (x_0,y) path. If $xy \in P$, then $xy \in E^-$ follows, and $\lambda(x_0,x) \leq w(P \setminus \{xy\}) = w(P) + 1$; if $xy \notin P$, then $xy \in E^+$, and $\lambda(x_0,x) \leq w(P \cup \{xy\}) = w(P) + 1$. Thus $\lambda(x_0,y) \leq \lambda(x_0,x) \leq \lambda(x_0,y) + 1$. If G is bipartite, then in addition $\lambda(x_0,x) \neq \lambda(x_0,y)$, whence $\lambda(x_0,x) = \lambda(x_0,y) + 1$, and (1.2) is proved.

For an arbitrary conservative weighting, the inequality $|\lambda(x_0,x) - \lambda(x_0,y)| \leq |w(xy)|$ can be checked in the same way (and also follows easily from (1.2); see the proof of Lemma 2.1(a)).

THEOREM 1.1. *Let G be a bipartite graph and $w : E(G) \to \{-1,1\}$ such that (G,w) is conservative. Furthermore, let $x_0 \in V(G)$ be arbitrary and D be the vertex set of a component of G^i $(i \in \{0, \pm 1, \pm 2, \ldots\}$, $V^i \neq \emptyset)$. Then $|\delta(D) \cap E^-| = 1$ provided that $x_0 \notin D$ and $|\delta(D) \cap E^-| = 0$ provided that $x_0 \in D$.*

If $\mathcal{D}$ denotes the family of sets occurring as the vertex set of a component of a G^i (as a D in the theorem), where G is bipartite, then because of (1.2), $\{\delta(D) : D \in \mathcal{D}\}$ partitions $E(G)$; the theorem states that the $D \in \mathcal{D}$ with $x_0 \notin D$ partition E^- into singletons.

Theorem 1 contains Seymour's minimax theorem on T-joins and T-cut packings, the Berge–Tutte theorem, and the Gallai–Edmonds structure theorem. It actually implies the generalization of this structure theorem to T-joins and weighted matchings (see Sebő [1990] and section 3 below).

Figure 1 illustrates the components of the level sets of a ± 1-weighted conservative graph. The thick edges are those of weight -1.

Proof of Theorem 1.1. Let (G,w) be conservative. We prove the theorem by

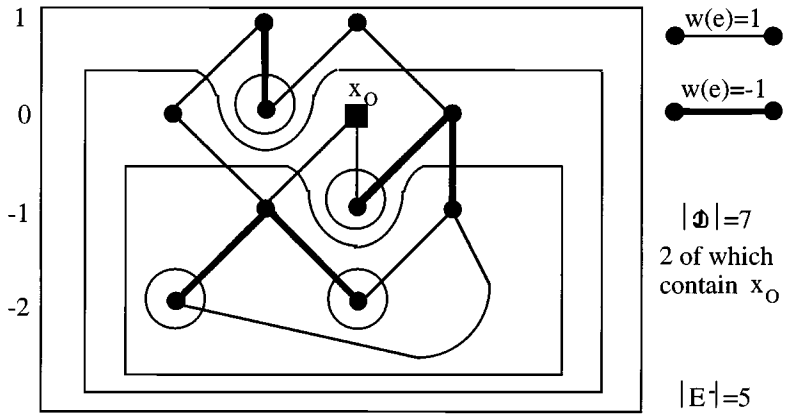


FIG. 1.

induction on $|V(G)|$. Let b be a vertex that satisfies

$$\lambda_w(x_0, b) = m := \min\{\lambda_w(x_0, x) : x \in V(G)\}$$

and let P be a simple (x_0, b) path, with $w(P) = m$. (P exists because of (1.1).) In addition, we assume that b is chosen among all possible choices so that $|P|$ is minimum. (This last assumption is not really essential; the claims that we will prove for b are true without it. However, it will be useful for technical simplicity.)

Let us first check the statement for $\{b\}$: since G is bipartite, V^m does not induce edges, whence $\{b\}$ is one of its components. The statement to check is the following.

CLAIM 1.

$$|\delta(b) \cap E^-| = 1 \quad \textit{provided } b \neq x_0,$$

$$|\delta(b) \cap E^-| = 0 \quad \textit{provided } b = x_0;$$

Moreover, if $b \neq x_0$, then P contains the unique negative edge adjacent to b.

If $b = x_0$, then because of $m = \lambda_w(x_0, b) = 0$, there cannot be a negative edge adjacent to $x_0 = b$. Now suppose that $b \neq x_0$.

Since P is simple, it has exactly one edge adjacent to b. That edge is negative because otherwise, if we delete it from P, we get a path which is shorter than $w(P) = m$. Suppose indirectly that there exists a negative edge $e \in \delta(b) \setminus P$: $P \cup e$ is also a path and $w(P \cup e) < w(P) = m$, and this contradiction proves Claim 1.

Let G^* be the graph that we obtain after contracting $\delta(b)$ or, equivalently, after identifying the vertices adjacent to b and deleting b. (By the choice of b and because of (1.2), all neighbors of b are at level $m - 1$.) We consider that the vertices and edges of G^* are the same as those of $G - b$; w^* is defined as the restriction of w to $E(G - b) = E(G^*)$. (Parallel edges can be replaced by one edge whose weight is the minimum of the weights.)

CLAIM 2. *Suppose $b \neq x_0$. Then (G^*, w^*) is conservative, and for every $x \in V(G^*)$, $x \neq b$, $\lambda_{G^*,w^*}(x_0, x) = \lambda_{G,w}(x_0, x)$.*

A circuit or (x_0, x) path ($x \in V(G)$, $x \neq b$) K^* of G^* is a circuit or (x_0, x) path of G or can be made into one by adding two edges of $\delta(b)$. Denote this corresponding circuit or path of G by K. Since $|\delta(b) \cap E^-| = 1$, $w(K^*) = w(K) - 2$ or $w(K^*) = w(K)$.

If K is not a zero-weight circuit or shortest (x_0, x) path, then since G is bipartite, $w(K) \geq 2$ or $w(K) \geq \lambda_w(x_0, x)+2$, respectively, and $w(K^*) \geq 0$ or $w(K^*) \geq \lambda_w(x_0, x)$ follow.

Therefore, let $w(K) = 0$ or $w(K) = \lambda_w(x_0, x)$, respectively. Claim 2 clearly follows now from the following: either $K \cap \delta(b) = \emptyset$ or $K \cap \delta(b) = \{e_1, e_2\}$, $e_1 \in E^-$, $e_2 \in E^+$.

Suppose indirectly that $K \cap \delta(b) = \{e_1, e_2\}$, $e_1, e_2 \in E^+$.

Case 1. $K \cap P = \emptyset$. If K is a zero-weight circuit, then $P' := (P \cup K) \setminus e_1$ is a path, with $w(P') = w(P) + w(K) - 1 = w(P) + 0 - 1 < m$, a contradiction. Therefore, suppose that K is a w-shortest (x_0, x) path. By (1.1), we can suppose that K is simple. Since the only edge of $K(x_0, b)$ adjacent to b is positive, by Claim 1, $K(x_0, b)$ is not a w-shortest path, that is, $w(K(x_0, b)) > w(P(x_0, b))$. Then, however, $w(P(x_0, b) \cup K(b, x)) = w(P(x_0, b)) + w(K(b, x)) < w(K(x_0, b)) + w(K(b, x)) = \lambda_w(x_0, x)$, a contradiction because $P(x_0, b) \cup K(b, x)$ contains an (x_0, x) path.

Case 2. $K \cap P \neq \emptyset$. Then $b \neq x_0$ (otherwise $P = \emptyset$), and by the last part of Claim 1, the edge of P adjacent to b is negative, so it is different from the edges of K adjacent to b. Thus $V(K) \cap V(P)$ cannot consist of b only. Walking on P from b towards x_0, let $a \neq b$ be the first vertex of K we meet. $K(a, b) \cap P(a, b) = \emptyset$. $K' := (K \setminus K(a, b)) \cup P(a, b)$ is also a circuit (if K is a circuit) or an (x_0, x) path (if K is so), whence $w(K(a, b)) \leq w(P(a, b))$; otherwise, K' would be shorter than K. Moreover, $w(P(a, b)) \leq 0$ because otherwise $w(P(x_0, a)) = w(P(x_0, b)) - w(P(a, b)) < m$. On the other hand, by the conservativeness of (G, w), $w(K(a, b) \cup P(a, b)) \geq 0$, whence we have equality throughout; in particular, $w(K(a, b)) = w(P(a, b)) = 0$ and $w(P(x_0, a)) = m$. $|P(x_0, a)| < |P(x_0, b)|$, contradicting the choice of b. Claim 2 is proved.

Theorem 1.1 now follows by induction in a straightforward way:

• If b can be chosen to be different from x_0, then the level sets of G^* are the same as those of G by Claim 2, except that $\{b\}$ is no longer a level set and its neighbors are identified.

• If $b = x_0$ is the only possible choice for b, then all distances are positive from x_0. In particular, there is no negative edge adjacent to x_0, and in contracting $\delta(x_0)$, all distances from x_0 decrease exactly by one; again, the level sets of G^* are the same as those of G, except that $\{x_0\}$ is no longer more a level set and its neighbors are identified.

In both cases, the components of the level graphs of G that are different from $\{b\}$ are exactly the components of the level graphs of G^*. We apply Claim 1 to $\{b\}$ and apply the induction hypothesis to G^*, and the theorem follows. □

In the remaining part of this section, we explore some applications of our results. For the moment, we restrict ourselves to the ± 1-weighted special case.

If $T \subseteq V(G)$, then $F \subseteq E(G)$ is called a *T-join* if T is the set of odd-degree vertices of F. $X \subseteq V(G)$ is said to be T-odd if $|X \cap T|$ is odd. $C \subseteq E(G)$ is a T-cut if $C = \delta(X)$ for some $X \subseteq V(G)$ and X is T-odd. It is an easy exercise to show that a T-join F and a T-cut C have an odd and, in particular, nonempty intersection, and it follows that the minimum cardinality of a T-join is at least as much as the maximum cardinality of a family of pairwise-disjoint T-cuts.

To give a first, typical example of how Theorem 1 relates T-joins and T-cuts, let us show how Theorem 1.1 implies Seymour's [1981] following well-known theorem.

(1.3) *If G is bipartite, then the minimum cardinality of a T-join is equal to the maximum cardinality of a family of pairwise-disjoint T-cuts.*

It is an easy exercise to show that a T-join F has minimum cardinality if and only if, upon defining the weight of the edges in F to be -1 and the weight of the other edges to be 1, we get a conservative weighting (a remark of Guan [1962]). Apply Theorem 1.1 to this conservative weighting and to an arbitrary $x_0 \in V(G)$. If D is a component of a level graph for which $x_0 \notin D$, then $|\delta(D) \cap F| = 1$; it follows that $\delta(D)$ is a T-cut. According to (1.2) applied to the bipartite graph G, every edge leaves some level graph, whence by Theorem 1.1, the number of T-cuts $\delta(D)$, $x_0 \notin D$, is $|F|$, and these are pairwise disjoint. We have thus found a family of pairwise-disjoint T-cuts which has the same cardinality as the minimum T-join, and Seymour's theorem is proved. In fact, the constructed set of disjoint T-cuts has the particular form of the packings presented by the following theorem of Frank, Sebő, and Tardos [1984].

(1.4) *Given a bipartite graph with a* ± 1 *conservative weighting, both classes of the bipartition can be partitioned into classes* $X_1, \dots, X_k$ *so that* $\delta(C)$*, where* C *is a component of* $G - X_i$ $(i = 1, \dots, k)$*, contains at most one negative edge.*

Indeed, according to Theorem 1.1, the vertices x, $\lambda_w(x_0, x) = i$, in the components of G^i with i odd, $i = \pm 1, \pm 3, \dots$ (or of G^i with i even), constitute the classes of a partition which has the claimed property.

(1.3) and (1.4) easily imply half-integer minimax theorems valid for arbitrary graphs. (The half-integer version of (1.3) is a result of Lovász [1975].)

The generalization of theorems on T-joins or conservative weightings to the weighted case is straighforward via subdivision of edges. For instance, given a weight function $w : E(G) \to \mathbb{N}$, the minimum weight of a T-join is at least as much as the maximum cardinality of a w-packing of T-cuts, where a *w-packing* is a multiset which covers edge $e \in E(G)$ at most $w(e)$ times; Seymour's theorem ((1.3)) states that there is equality here for bipartite weightings. (For the sharpening series of integer minimax theorems that have been developed (including weighted variants) and the blocking pair of T-join and T-cut polyhedra, see Lovász and Plummer [1986b]; for a survey of more recent results, see, for instance, Frank [1990] or other recent publications on T-joins mentioned in the reference list. We will state some of these in section 3.) The weighting $w: E(G) \to \mathbb{Z}$ is called *Eulerian* if $w(C)$ is even for every cut C, and it is *bipartite* if it is even for every circuit C. A graph is Eulerian or bipartite if the identically 1 function on its edges is Eulerian or bipartite.

Let us also note that the distances *do not depend on the choice of the minimum T-join F*. (For an easy exercise, see, for example, Sebő [1990].)

The weighted generalization of (1.4) is somewhat artificial. On the other hand, Theorem 1.1 itself can be straightforwardly generalized to the weighted case, and this can be used for various purposes, as we will show in the following sections.

We finish this introduction by showing how Theorem 1.1 already applies to unweighted multiflows, that is, edge-disjoint path problems. We assume that the reader is familiar with the (easy) relation between some notions in planar graphs and those in the planar dual: the dual of the dual of a graph G is equal to G; Eulerian and bipartite graphs or weight functions correspond to each other; disjoint unions of circuits correspond to sets of the form $\delta(X)$ $(X \subseteq V(G))$; etc.

If G is planar, in the dual graph, Theorem 1.1 has the following more apparent meaning.

We are given an Eulerian graph (dual of "bipartite") G embedded in the plane and $R \subseteq E(G)$ so that the *cut condition*

$$|C \cap R| \leq |C \setminus R| \quad \text{for every cut } C$$

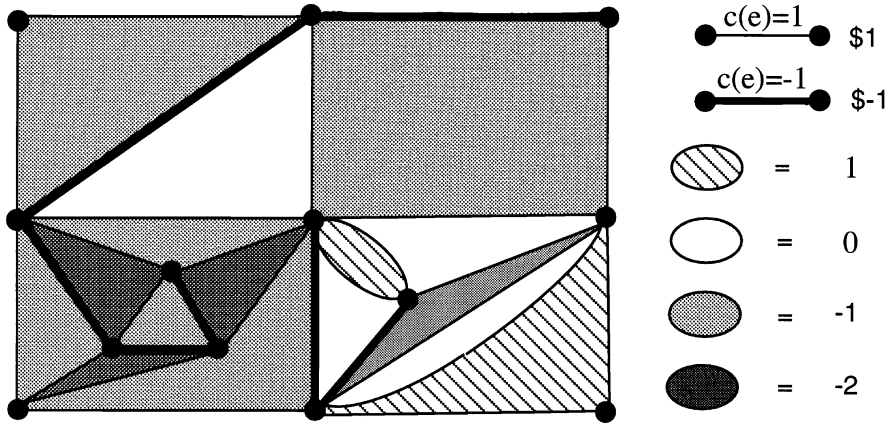


FIG. 2.

is satisfied for (G, R). (This condition is equivalent to the conservativeness of the dual graph. Set weights -1 on (duals of) edges in R.) Assign to every face φ of G the following *magic numbers*:

$$\lambda(\varphi) := \min\{|P \setminus R| - |P \cap R| : P \text{ dual path from the infinite face to } \varphi\}.$$

A "dual" path is a path of the dual graph and can be imagined as going from face to face, crossing edges. $\lambda(\varphi)$ can be interpreted as the minimum cost of reaching face φ if we must pay 1\$ for crossing an edge in $|E(G) \setminus R|$ and -1\$ for crossing an edge of R. (Negative costs correspond to incomes.)

This definition of $\lambda(\varphi)$ corresponds to a natural choice of x_0 in Theorem 1.1. Let x_0 be the vertex corresponding to the infinite face of the dual graph. Let us apply Theorem 1.1 to the dual of G with this choice of x_0.

The union of any set of faces of a planar graph can be partitioned into (topologically) connected regions, where each region is the union of faces corresponding to the components of the dual graph. Consider the regions determined by the faces φ with $\lambda(\varphi) \leq i$ (Figure 2). Among these, those whose territory is bounded (equivalently, which do not contain the infinite face) will be called *patches* (see Figure 2).

THEOREM 1.2. *Let G be an Eulerian graph embedded in the plane, and let $R \subseteq E(G)$ be such that (G, R) satisfies the cut condition. Then the boundaries of patches contain exactly one edge of R each, they are pairwise disjoint, and every $e \in R$ is contained in one of them.*

Proof. Apply Theorem 1.1 to the dual of G, where the edge weights are -1 for the duals of edges in R and 1 otherwise; note that the dual of G is bipartite; define x_0 to be the vertex of the dual graph corresponding to the infinite face of G. Clearly, there is a one-to-one correspondence between the patches of G and those components of the level graphs of the dual graph which do not contain x_0. Because of (1.2), the boundaries of these patches are disjoint; the rest of the statement can be extrapolated from Theorem 1.1. □

In other words, if G is a planar Eulerian graph and $R \subseteq E(G)$ is such that (G, R) satisfies the cut condition, then Theorem 1.2 provides a uniquely determined integer "multiflow" (see the definition below), which will be called a *patch flow*. Similarly, the packing of odd cuts defined after Theorem 1.1 will be called a *patch packing* for (G, T, x_0).

The reader may find it amusing to translate the proof of Theorem 1.1 to give a direct proof of Theorem 1.2 and, in fact, of the following sharpening.

The necessity of the cut condition for the existence of paths in $G - R$ between the endpoints of the edges in R is trivial. The sufficiency is just the "planar dual" of (1.3). That is, we have shown a constructive proof (see the corresponding polynomial algorithm in section 3) of the following theorem of Seymour [1981].

(1.5) *Let G be a planar Eulerian graph and $R \subseteq E(G)$. Then there exist edge-disjoint paths in $G - R$ between the endpoints of the edges of R if and only if the cut condition is satisfied for (G, R).*

We conclude this section by defining multiflows precisely and listing a sequence of known results about them. Let G be a graph and $c: E(G) \to \mathbb{R}$. c will mean capacity on the positive edges and demand on the negative edges. We define the *demand* of $e \in E(G)$, $c(e) < 0$, to be $-c(e)$. If $c(e) < 0$, e is called a demand edge.

Given a weight function in a graph, let us denote the set of negative edges by E^- and the set of nonnegative edges by E^+. A *multiflow* for (G, c) is a set of circuits $\mathcal{C}$ with "multiplicities" $f: \mathcal{C} \to \mathbb{R}$ such that

$$|C \cap E^-| = 1 \quad \text{for all } C \in \mathcal{C},$$

$$\sum_{e \in C} f(C) \leq c(e) \quad \text{if } e \in E^+, \quad \text{and}$$
$$\sum_{e \in C} f(C) = -c(e) \quad \text{if } e \in E^-.$$

If $f(e)$ is (half-) integer for every $e \in E(G)$, we say that the flow is (*half-*) *integer*. If in this definition we replace "circuit" by "cut," we say that $(\mathcal{C}, f)$ is a *dual multiflow*. The planar special case of dual multiflows is the panar multiflow problem, but dual multiflows have the advantage that the theorems concerning them are valid for arbitrary graphs.

The (*integer*) *multiflow problem* is the problem whose instances are (G, c) pairs, where G is a graph and $c: E(G) \to \mathbb{R}$, and the question is to decide the existence of an (integer) multiflow. In the *planar multiflow problem*, we consider only instances where G is planar.

Statements about capacitated multiflow problems can be reduced to uncapacitated ones by replacing e by $|c(e)|$ parallel copies of e. For instance, the cut condition becomes

$$c(\delta(X)) \geq 0 \quad \text{for every } X \subseteq V(G).$$

The results of Edmonds and Johnson [1973], Lovász [1975], Barahona [1980], and Korach [1982] proved that in a planar graph, a half-integer flow or a violating cut can be found in polynomial time. The best complexity was attained by Barahona [1989]. The closest predecessor to our approach is in the work of Matsumoto, Nishizeki, and Saito [1986]. They decreased c along a face for all possible choices of faces and checked the cut condition for each choice. The proof of Theorem 1.1 indicates explicitly that the face to be chosen is φ with $\lambda(\varphi)$ minimum; moreover, Theorem 1.1 foresees the entire multiflow, the same one which would be the result of alternatively determining φ and deleting its boundary.

Various results about integral (dual) multiflows have been obtained by Seymour [1977, 1981], Korach and Penn [1992], Frank [1990], Sebő [1987a, b], and, more recently, Frank and Szigeti [1995] and Ageev, Kostochka, and Szigeti [1995]. These

integer multiflows can also be obtained with the help of the "magic numbers"—some of them with considerable additional work, but also with the algorithmic advantages that this represents (see section 3).

The planar multiflow problem in general was proved by Middendorf and Pfeiffer [1989] to be NP-complete.

In section 2, we will extrapolate the weighted generalization of the results in this section, and in section 3, we will apply the results that we obtain.

2. Potentials. Potentials in directed graphs are defined in the following way:

Let G be a directed graph, $w: E(G) \to \mathbb{R}$, and $x_0 \in V(G)$. (G, w) is said to be *conservative* if $w(C) \geq 0$ for every directed circuit C. $\pi : V(G) \to \mathbb{R}$ is called a *potential* (centered at x_0) if

$$\pi(x_0) = 0 \quad \text{and}$$

$$\pi(y) - \pi(x) \leq w(x, y) \quad \text{for every directed edge } xy.$$

The role of potentials is apparent from, for instance, the following well-known proposition.

(2.1)

(a) (G, w) *is conservative if and only if there exists a potential.*

(b) *If* π *is a potential centered at* x_0, *then for all* $x \in V(G)$,

$$\lambda_w(x_0, x) \geq \pi(x).$$

(c) *If* (G, w) *is conservative, the function defined by* $\pi(x) := \lambda_w(x_0, x)$ *is a potential centered at* x_0.

(b) and the *if* part of (a) are trivial; (c) is also easy and proves the *only if* part of (a). In other words, potentials centered at x_0 give an apparent proof of conservativeness and a lower bound for the distances from x_0. (c) claims the a priori surprising fact that $\max\{\pi(x) : \pi \text{ is a potential centered at } x_0\}$ is attained by one and the same potential for every $x \in V(G)$, that is, by the distance function λ, $\lambda(x) := \lambda_w(x_0, x)$ $(x \in V(G))$.

Now let G be undirected. Suppose first that G is bipartite, $w: E(G) \to \{-1, 1\}$, and $x_0 \in V(G)$. $\pi: V(G) \to \mathbb{Z}$ will be called a *potential* centered at x_0 if

(i) $\pi(x_0) = 0$ and

(ii) $|\pi(y) - \pi(x)| = 1$ for every $xy \in E(G)$.

If D is a component of the graph $G^i = G^i(\pi)$ induced by the level set $V^i = V^i(\pi)$ $:= \{x \in V(G) : \pi(x) \leq i\}$ $(i = 0, \pm 1, \pm 2, \ldots)$, then

(iii)

$$|\delta(D) \cap E^-| = 1 \quad \text{provided that } x_0 \notin D,$$

$$|\delta(D) \cap E^-| = 0 \quad \text{provided that } x_0 \in D.$$

Theorem 1.1 can now be restated in the following way.

(2.2) *If* G *is* ± 1*-weighted and bipartite, then* (2.1) *holds.*

The *if* part of (2.1a) and all of (2.1b) can be proved in a straightforward way. (For details, see Sebő [1990].) To prove (2.1c), which also implies the *only if* part of (2.1a), we have to check (i), (ii), and (iii): (i) is trivial, (ii) is easy (we have already proved it; see (1.2)), and (iii) is Theorem 1.1.

We have arrived at the main purpose of this section: we will generalize potentials for arbitrary undirected graphs with arbitrary weights. Of course, we have to satisfy two constraints: it should be easy to check whether a given function is a potential (like the inequality for directed graphs or (i), (ii), and (iii) for bipartite ±1-weighted graphs); (2.1) should be true.

For the sake of simplicity, we first do this work only for integer weights, and then we will observe that the theorems hold for arbitrary real weights almost without change and that the extension is straightforward.

If w is not bipartite, we associate with the pair (G,w) a bipartite graph $\hat{G}$ and a weight function $\hat{w}\colon E(\hat{G}) \to \{-1,1\}$ in the following way:

Contract the zero-weight edges of G and replace each edge $e \in E(G), w(e) \neq 0$, by $2|w(e)|$ edges in series. (Divide e into $2|w(e)|$ edges by $2|w(e)|-1$ new points.) We think of $V(G)$ as a subset of $V(\hat{G})$. If $\hat{e} \in E(\hat{G})$ is an element of the subdivision of $e \in E(G)$, let $\hat{w}(\hat{e}) = -1$ if $w(e) < 0$ and let $\hat{w}(\hat{e}) = +1$ if $w(e) > 0$.

Clearly, the natural correspondence between paths of G and paths of $\hat{G}$ simply doubles the weights. Thus

$$\forall x, y \in V(G), \quad \lambda_{\hat{G},\hat{w}}(x,y) = 2\lambda_{G,w}(x,y).$$

This shows that we shall have an easy task. We know that the potentials in $(\hat{G},\hat{w})$ are the functions for which (i), (ii), and (iii) hold. On the other hand (e.g., for applications), we need theorems that consider only (G,w) directly; potentials should be defined in terms of (G,w). We will face no obstacles in obtaining this direct definition because it turns out that a potential $\hat{\pi} : V(\hat{G}) \to \mathbb{Z}$ is *already determined by its restriction to* $V(G)$.

LEMMA 2.1. *Suppose $\hat{\pi} : V(\hat{G}) \to \mathbb{Z}$ is a potential in $(\hat{G},\hat{w})$. Then $\hat{\pi}$ is even on $V(G)$, and if $\pi : V(G) \to \mathbb{Z}$ denotes the restriction of $\hat{\pi}/2$ to $V(G)$, then for every $ab \in E(G)$,*

(a) $|\pi(a) - \pi(b)| \leq |w(ab)|$.

Denote the path of $\hat{G}$, replacing $ab \in E(G)$ by $Q = \{p_0,\ p_1, \dots, p_{t-1},\ p_t\}$ ($p_0 = a,\ p_t = b,\ t = 2|w(ab)|$).

(b) *π uniquely determines $\hat{\pi}$, namely, if $w(ab) \neq 0$, then in $(\hat{G},\hat{w})$,*

$$\hat{\pi}(p_i) := \begin{cases} 2\pi(a) + i & \text{provided that } 0 \leq i \leq i_0, \\ 2\pi(b) + |2w(ab)| - i & \text{provided that } i_0 \leq i \leq t, \end{cases}$$

where $i_0 = \pi(b) - \pi(a) + |w(ab)|$.

(c) $\max_{0\leq i\leq t}\ \hat{\pi}(p_i) = \hat{\pi}(p_{i_0}) = \pi(a) + \pi(b) + |w(ab)|$.

Remark. Lemma 2.1 gives the value of $\hat{\pi}(p_i)\,(0 \leq i \leq t)$; it even gives two definitions for $\hat{\pi}(p_{i_0})$, but both of them define the value $\hat{\pi}(p_{i_0}) = \hat{\pi}(a) + \hat{\pi}(b) + |\hat{w}(ab)|$.

Lemma 2.1 is illustrated in Figure 3.

Proof of Lemma 2.1. First, we prove that $\hat{\pi}$ is even on $V(G)$. Indeed, it follows from (ii) that $\hat{\pi}$ has different parity on the two endpoints of every edge, and hence its parity is fixed on each class of the bipartition of $\hat{G}$. Since one of these classes contains $V(G)$, and since $\hat{\pi}(x_0) = 0$, we get that $\hat{\pi}$ is even on $V(G)$.

We now prove (a). If $w(ab) = 0$, then $\hat{\pi}(a) = \hat{\pi}(b)$, and (a) is trivial.

Suppose that $w(ab) \neq 0$. According to (ii), $|\hat{\pi}(p_i) - \hat{\pi}(p_{i-1})| = 1$, whence

$$2|\pi(a) - \pi(b)| = |\hat{\pi}(a) - \hat{\pi}(b)| = \left|\sum_{i=1}^{t} \hat{\pi}(p_i) - \hat{\pi}(p_{i-1})\right| \leq e\sum_{i=1}^{t} |\hat{\pi}(p_i) - \hat{\pi}(p_{i-1})|$$

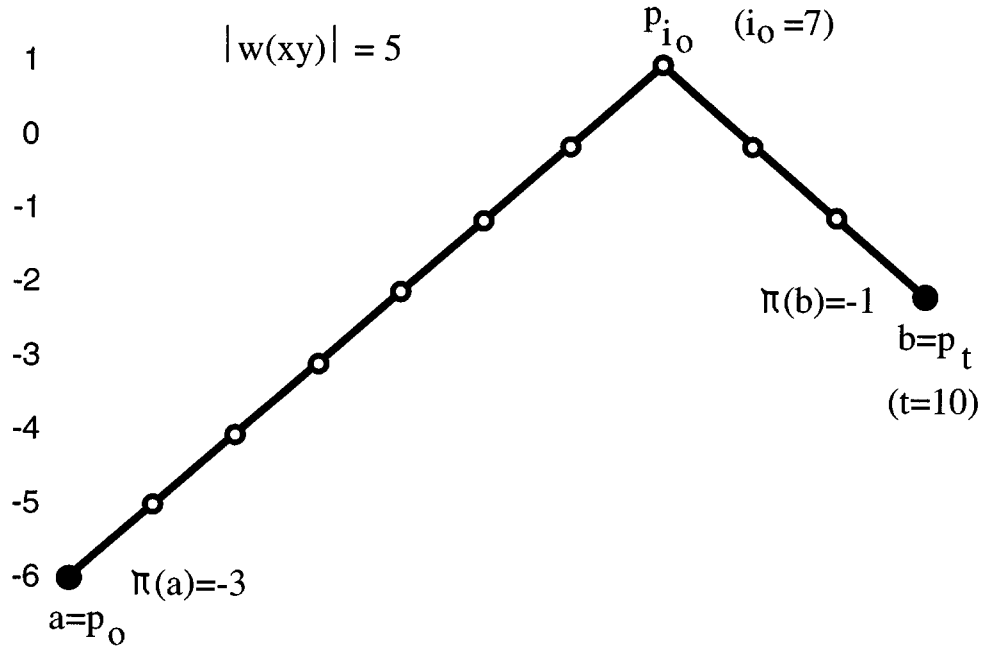


FIG. 3.

$$= t = |2w(ab)|,$$

and (a) is proved.

If $\hat{\pi}(p_i) = \hat{\pi}(p_{i-1}) - 1$, then we call the edge $p_{i-1}p_i$ *descending*, and if $\hat{\pi}(p_i) = \hat{\pi}(p_{i-1}) + 1$, then we call it *ascending*. By (ii), every edge is either descending or ascending. If $p_{i-1}p_i$ is descending, $i < t$, then p_ip_{i+1} cannot be ascending; otherwise, $\{p_i\} \in D$, which contradicts (iii) because $\delta(p_i)$ consists of two positive or two negative edges. From this it follows that there exists an $i_0 \in \mathbb{N}$ such that $p_{i-1}p_i$ is ascending if $1 \le i \le i_0$ and descending if $i_0 < i \le t$. Consequently, with this i_0, $\hat{\pi}(p_i) = \hat{\pi}(p_0) + i$ if $0 \le i \le i_0$ and $\hat{\pi}(p_i) = \hat{\pi}(p_t) + (t - i)$ if $i_0 \le i \le t$. Hence $\hat{\pi}(p_0) + i_0 = \hat{\pi}(p_{i_0}) = \hat{\pi}(p_t) + (t - i_0)$, that is, $i_0 = (\hat{\pi}(b) - \hat{\pi}(a) + 2|w(ab)|)/2$. Thus we have proved (b). (c) is the immediate consequence of (b). □

We call a function $\pi : V(G) \to \mathbb{Z}$ a *potential centered at* $x_0 \in V(G)$ if the function 2π is the restriction to $V(G)$ of a potential in $\hat{G}$ centered at $x_0 \in V(\hat{G})$. Given G, w, and a potential π, we will keep the notation $\hat{G}$, $\hat{w}$, $\hat{\pi}$; these are uniquely determined (see Lemma 2.1(b)). The properties of $\hat{\pi}$ can be "translated" into properties of π:

Let $\pi : V(G) \to \mathbb{Z}$ and define $G^i_w = G^i_w(\pi)$ by $V(G^i_w) := \{x \in V(G) : \pi(x) \le i\}$ and $E(G^i_w) := \{xy \in E(G) : (\pi(x) + \pi(y) + |w(xy)|)/2 \le i\}$ ($i = 0, \pm 1/2, \pm 1, \pm 3/2, \ldots$). We also introduce the notation

$$\mathcal{D} := \{D \subseteq V(G)\text{: } D \text{ is a component of } G^i_w \text{ for some } i\}.$$

LEMMA 2.2. *Let G be an arbitrary undirected graph, $w : E(G) \to \mathbb{Z}$, $x_0 \in V(G)$. $\pi : V(G) \to \mathbb{Z}$ is a potential centered at x_0 if and only if the following hold:*

(i) $\pi(x_0) = 0$.

(ii) $|\pi(y) - \pi(x)| \le |w(x, y)|$ *for all* $xy \in E(G)$.

(iii) *If D is a component of $G^i_w(\pi)$, then*

- *all negative edges induced by D in G are in $E(G^i_w)$ and*
- (iii) *holds for D.*

Proof. To prove the *only if* part, suppose that $\pi : V(G) \to \mathbb{Z}$ is a potential centered at x_0, that is, 2π is the restriction to $V(G)$ of a potential $\hat{\pi}$ of $(\hat{G}, \hat{w})$ centered at x_0. We must prove that (î), (îî), and (îîî) hold for π provided that (i), (ii), and (iii) hold for $\hat{\pi}$. (î) is obvious and (îî) is just Lemma 2.1(a).

To prove (îîî), let $\hat{\imath} \in \mathbb{Z}$ and note the following: the level graph $\hat{G}^{\hat{\imath}}(\hat{\pi})$ of $\hat{G}$ contains exactly those vertices $v \in V(G)$ for which $\hat{\pi}(v) = 2\pi(v) \leq \hat{\imath}$; by Lemma 2.1(c), it entirely contains the subdivision of exactly those edges $ab \in E(G)$ for which $\pi(a) + \pi(b) + |w(ab)| \leq \hat{\imath}$. That is, the components of $\hat{G}^{\hat{\imath}}$ are determined by those vertices $v \in V(G)$ for which $\pi(v) \leq \hat{\imath}/2$ and those edges $ab \in E(G)$ for which $(\pi(a) + \pi(b) + w(ab))/2 \leq \hat{\imath}/2$ ($\hat{\imath} = 0, \pm 1, \pm 2, \ldots$). Defining $i := \hat{\imath}/2$, we see that these vertices and edges are exactly those of $E(G^i_w(\pi))$. Thus there is a one-to-one correspondence between the components of $\hat{G}^{\hat{\imath}}(\hat{\pi})$ and those of $G^i_w(\pi)$. If D is a component of the latter, we will denote the corresponding component of the former by $\hat{D}$.

(îîî) now follows easily. If a negative edge ab induced by D is not in $E(G^i_w)$, then $(\pi(a) + \pi(b) + |w(ab)|)/2 > i$, and it follows from Lemma 2.1 that there are two negative edges in $\delta(\hat{D})$. (Follow on Figure 3: since $p_{i_0} \notin \hat{D}$ but $a, b \in \hat{D}$, $\delta(\hat{D})$ contains one negative edge from each of the paths (a, p_{i_0}) and (p_{i_0}, b).) Therefore, since by assumption $\hat{\pi}$ satisfies (ii), such an edge ab does not exist, and the first part of (îîî) is proved. Furthermore, it now follows that the negative edges of $\hat{D}$ are in one-to-one correspondence with those entering D, so the second part of (îîî) holds as well.

To prove the *if* part, suppose that (î), (îî), and (îîî) hold for π. Define $\hat{\pi}$ to be equal to 2π on $V(\hat{G}) \cap V(G)$, and extend it to the entire $V(\hat{G})$ in the unique way dictated by Lemma 2.1 (b). By definition, $\hat{\pi}$ satisfies (i) and (ii). If D is as in (îîî), then by the same correspondence as in the proof of the *only if* part, D correponds to a component $\hat{D}$ of $\hat{G}^{\hat{\imath}}(\hat{\pi})$, and because of the first part of (îîî), there is a one-to-one correspondence between the negative edges of $\delta(\hat{D})$ and those of $\delta(D)$. Now the second part of (îîî) implies that (iii) holds for $\hat{D}$. □

The following theorem is a straightforward reformulation of Theorem 1.1 (that is, of (2.2)) for the weighted case.

THEOREM 2.1 *Let G be an arbitrary undirected graph, and let $w : E(G) \to \mathbb{Z}$ such that (G, w) is conservative*; *furthermore, $x_0 \in V(G)$. Then $\lambda : V(G) \to \mathbb{Z}$, where $\lambda(x) := \lambda_{G,w}(x_0, x)$ $(x \in V(G))$ is a potential centered at x_0. Furthermore,* (2.1) *holds.*

Proof. Apply Theorem 1.1 and then (2.2) to the conservative bipartite $(\hat{G}, \hat{w})$. Then apply Lemma 2.2 to obtain the result for (G, w). □

The reader may find it to be a useful exercise to give a direct proof of the trivial *if* part of (2.1a) and all of (2.1b), where the definition of potentials is given in (î), (îî), and (îîî).

The main point of Theorem 2.1 is that (îîî) holds for the distances from x_0. Since this statement will be often used in what follows, let us restate it separately in a slightly different form for later convenience.

(2.3) *Let G be a undirected graph, $w : E(G) \to \mathbb{Z}$ such that (G, w) is conservative, and $x_0 \in V(G), \lambda(x) := \lambda_{G,w}(x_0, x)$, $G^i_w := G^i_w(\lambda)$.*

(a) *If D is a component of G^i_w,*

$$|\delta(D) \cap E^-| = 1 \quad \textit{provided that } x_0 \notin D,$$

$$|\delta(D) \cap E^-| = 0 \quad \textit{provided that } x_0 \in D.$$

(b) *Let $y(D) := \max\{i : D \text{ is a component of } G^i\} - \min\{i : D \text{ is a component of } G^i\}$. Then for all $e \in E^-$, $\sum_{D\in\mathcal{D}, e\in\delta(D)} y(D) = |w(e)|$, and for all $e \in E(G)$, $\sum_{D\in\mathcal{D}, e\in\delta(D)} y(D) \le |w(e)|$.*

(2.3) shows how the magic numbers $\lambda(x_0, x)$ generate a special dual flow (packing of T-cuts). This will be the basis of the applications in the following section.

In the rest of the paper, G^i will denote the graph defined in (2.3) (given that (G, w) is conservative and $x_0 \in V(G)$). The family $\mathcal{D}$ can be split into the union of $\mathcal{D}'$ and $\mathcal{D}''$, where $\mathcal{D}'$ is the family of all components of G^i with i as an integer and $\mathcal{D}''$ is the family of all components of G^i with i as a noninteger (but, of course, as a half-integer). Each element of $\mathcal{D}''$ is partitioned by some elements of $\mathcal{D}'$ because $V(G^i) = V(G^{i+1/2})$, and $E(G^i) \subseteq E(G^{i+1/2})$. The following remark will be useful in the construction of integer (dual) flows or packings.

(2.4) *Let G be an undirected graph, and let $w : E(G) \to \{-1, 1\}$ so that (G, w) is conservative. Then $\{\delta(D) : D \in \mathcal{D}'', x_0 \notin D\}$ is a set of disjoint cuts each of which contains one negative edge, and $xy \in E(G)$ is in none of these cuts if and only if $\lambda(x) = \lambda(y)$.*

Indeed, let $xy \in E(G)$, and suppose without loss of generality that $\lambda(x) \le \lambda(y)$. Then either $\lambda(x) = \lambda(y)$ or $\lambda(x) = \lambda(y) - 1$. In the latter case, xy is in $\delta(D)$, where D is the component of $G^{\lambda(x)+1/2}$ containing x; xy is in none of the other sets of the form $\delta(X)$, $X \in \mathcal{D}''$. In the former case, $(\lambda(x)+\lambda(y)+w(xy))/2 = \lambda(x)+1/2$, that is, $xy \in E(G^{\lambda(x)+1/2}$. It follows that x and y are in the same component of $G^{\lambda(x)+1/2})$, whence none of the sets $\delta(D)$ $(D \in \mathcal{D}'')$ contains it, and (2.4) is proved.

The dual flow defined by

$$\mathcal{V} := \{\delta(D) : D \in \mathcal{D},\, x_0 \notin D\}$$

with multiplicities $y(D)$ will be called the *patch dual flow* of (G, w, x_0). It is, in fact, a dual multiflow. A patch dual flow becomes a multiflow in the dual of a planar graph (and if we dualize so that x_0 is the infinite face, this is the uniquely determined patch flow); applying it to the conservative graph corresponding to a minimum T-join, we get a *patch packing* of T-cuts (see section 1 for the unweighted case), which is a maximum packing of T-cuts.

Note that the results can be *generalized to arbitrary $w: E(G) \to \mathbb{R}$*, where the potentials can be defined by (î), (ĩî), and (ĩĩî). *The distances still form a potential*; *furthermore,* (2.1) *still holds.*

Indeed, let $\varepsilon \le (1/2n^2)\mu$, where μ is the minimum difference between *different* path lengths. The function $w_\varepsilon(e) := \lceil w(e)/\varepsilon \rceil$ $w_\varepsilon: E(G) \to \mathbb{N}$ has the property that for two paths P and Q (not necessarily between the same pair of vertices), $w(P) < w(Q)$ implies $w_\varepsilon(P) < w_\varepsilon(Q)$, whence the w_ε-shortest paths are also w-shortest paths.

Since $w(e)/\varepsilon \le w_\varepsilon \le 1 + w(e)/\varepsilon$, we have $w(e) \le \varepsilon w_\varepsilon \le \varepsilon + w(e)$. Thus, choosing an arbitrary series $\varepsilon_n \to 0$, $\varepsilon_n w_{\varepsilon_n} \to w$ holds (uniformly on the paths). It follows that, applying (2.1) to w_{ε_n}, the same follows for $\varepsilon_n w_{\varepsilon_n}$. Since, as we noticed, the w_{ε_n}-shortest paths are also w-shortest, the distances according to the weight function $\varepsilon_n w_{\varepsilon_n}$ converge to λ_w. Now since in (î), (ĩî), and (ĩĩî) we only have linear functions

of convergent series, these linear functions also converge to the same linear function of the limits, and (2.1) follows for arbitrary real weights in a straightforward way.

Patch flows and patch packings (without using this term) have been applied to prove integer-path or cut-packing theorems. See, for instance, Sebő [1990], Frank and Szigeti [1995], or Ageev, Kostochka, and Szigeti [1995].

3. Algorithms and applications. It is well known that an algorithm for minimum-weight T-joins has been deduced from weighted matching algorithms in various ways (see Edmonds [1965b] and Lawler [1976]). The same sources reduce the shortest-undirected-path problem of conservative graphs to matching problems with similar gadget-type reductions. However, these and other "Waterloo folklore solutions" do not give a satisfactory answer to the dual of the minimum-weight T-join problem (though it is often needed in applications; see below). That is why there were later several attempts at the solution of the primal and dual problem at the same time; see Barahona [1980, 1989], Edmonds and Johnson [1973], and Korach [1982]. Barahona [1989] gave a clear presentation; Barahona and Cunningham [1989] showed an elegant way to provide an integer dual solution in the bipartite case.

Let n stand for $|V(G)|$ and m stand for $|E(G)|$ in the rest of the paper. In this section, we will show that a dual flow or a maximum w-packing of T-cuts can be found by solving n independent matching problems (which can also be carried out in parallel). Furthermore, we always get a half-integer solution, and if for all circuits $w(C)$ is even, then we automatically get an integer solution. Then we collect applications of weighted and unweighted potentials, the complexity of which is determined by our algorithm.

In the following, we shall need a subroutine that determines a minimum-weight T-join. For this we shall use Edmonds and Johnson's [1973] reduction to matchings, which we shall describe now. It is the following obvious fact that makes possible the use of this method.

LEMMA 3.1. *Suppose that (G,w) is conservative, $a,b \in V(G)$, and $T := \{x \in V(G) : d_{E^-}(x)$ is odd$\}$. If F is a $|w|$-minimum $T\triangle\{a,b\}$-join, then in $F\triangle E^-$, any (a,b)-path is w-shortest.*

Remark. Since $F\triangle E^-$ is an $\{a,b\}$-join, an (a,b)-path contained in it is trivial to find. Thus Lemma 3.1 reduces the shortest-path problem to finding a minimum T-join for nonnegative weights. Since, moreover, the zero-weight edges can be contracted, in the following, we can concentrate on T-joins in graphs with positive weights.

We also remark that the converse of Lemma 3.1 is also true (but irrelevant here): if P is a w-shortest (a,b)-path, then $P\triangle E^-$ is a $|w|$-minimum $T\triangle\{a,b\}$-join.

Proof of Lemma 3.1. Clearly, $F\triangle E^-$ is an $\{a,b\}$-join, and

$$\begin{aligned} w(F\triangle E^-) &= w(F \setminus E^-) + w(E^- \setminus F) \\ &= |w|(F \setminus E^-) + |w|(F \cap E^-) - |w|(E^- \cap F) + w(E^- \setminus F) \\ &= |w|(F) + w(E^-). \end{aligned}$$

Thus $|w|(F)$ and $w(F\triangle E^-)$ differ only in a constant independent of F. F is a $|w|$-minimum $T\triangle\{a,b\}$-join $\Leftrightarrow$ $F\triangle E^-$ is a w-minimum (a,b)-path $\Leftrightarrow$ in $F\triangle E^-$, an arbitrary circuit has weight 0. □

Now let $w : E(G) \to \mathbb{N}$, and on the edges of the complete graph on H with $V(H) := T$, define the weighting $c(x,y) := \min\{w(P) : P \subseteq E(G),\ P$ is an (x,y) path$\}$

> 0. ($c(x, y)$ can, for instance, be computed by Dijkstra's algorithm; see, for example, Lawler [1976].) The following lemma of Edmonds and Johnson [1973] relates the w-minimum T-joins of G to the c-minimum matchings of H.

LEMMA 3.2. *Let G be a connected graph, $T \subseteq V(G)$, where $|T|$ is even, and $w : E(G) \to \mathbb{N}$, and from these let us define H and c in the above way. Furthermore, let $k := |V(H)|/2$. If $\{x_i y_i : i = 1, \dots, k\}$ is a c-minimum perfect matching in H and P_i is a shortest (x_i, y_i)-path $(i = 1, \dots, k)$, then $P_i \cap P_j = \emptyset$ $(i \neq j)$ and $\bigcup_{i=1}^{k} P_i$ is a w-minimum T-join in G.*

Remark. Applying both Lemmas 3.1 and 3.2, the shortest paths of conservative graphs can be determined using any matching algorithm (see Algorithm 1 below).

Proof of Lemma 3.2. Let $M := \{x_i y_i : i = 1, \dots, |V(H)|/2\}$ be a c-minimum perfect matching and P_i be a w-minimum (x_i, y_i) path $(i = 1, \dots, k)$. Thus $P_i \cap P_j = \emptyset$ since if $P_i \cap P_j \neq \emptyset$, then $w(P_i \triangle P_j) < w(P_i) + w(P_j)$, and $P_i \triangle P_j$ contains two (edge-) disjoint paths between two disjoint pairs of points in $\{x_1, y_1, x_2, y_2\}$, which contradicts the minimality of M. □

The converse of this lemma is also easy. Given nonnegative weights, a minimum-weight T-join (which is a forest) is easy to split into edge-disjoint paths between pairs in $V(H)$, and no matter how we carry this out, the pairs will create a c-minimum matching of H.

We now have the means at our disposal to describe the algorithm based on Theorem 2.1 (section 2).

ALGORITHM 1.

Input: graph G, $x_0 \in V(G)$, and $w : E(G) \to \mathbb{Z}$.

Output: either a negative circuit in (G, w) or a feasible w-packing which is furthermore the uniquely existing patch dual flow belonging to (G, w, x_0).

0. Contract the zero-weight edges and with the help of Lemma 3.2 above, determine a $|w|$-minimum T-join F, $T := \{x \in V(G) \ : \ d_{E^-}(x) \text{ is odd}\}$.

• If $|w|(F) < |w|(E^-)$, then a negative circuit can easily be found in $F \triangle E_-$. STOP.

• If $|w|(F) = |w|(E_-)$, then GOTO 1.

1. With the help of Lemmas 3.1 and 3.2, determine the weight of a w-shortest (x_0, x) path for every $x \in V(G), x \neq x_0$. Let this number be denoted by $\lambda(x)$. GOTO 2.

2. Let the function $\lambda : V(G) \cup E(G) \to \mathbb{Z}$ be the following:

$$\lambda(x) := \begin{cases} \lambda(x) & \text{if } x \in V(G), \\ \dfrac{\lambda(u) + \lambda(v) + w(uv)}{2} & \text{if } x = uv \in E(G). \end{cases}$$

• For the value of $\lambda(x)$ $(x \in V(G))$ and $\lambda(xy)$ $(xy \in E(G))$, define the components of the graph G^i, that is, the set system $\mathcal{D}$. (In the case of bipartite weightings, λ takes only integer values because $\lambda(x)$ and $\lambda(y)$ have the same parity if $w(xy)$ is even and have different parities if it is odd.) The multiplicities $y(D)$ $(D \in \mathcal{D})$ are easily seen to be computable in the following way:

$$y(D) \leftarrow \max_{e \in E^- \cap \delta(D)} \lambda(e) - \min_{e \in E^- \cap E(D)} \lambda(e).$$

(See (2.3b); for a proof, use (îîî).)

$$\mathcal{V} \leftarrow \{\delta(D) : D \in \mathcal{D}, x_0 \notin D\},$$

where we mean the multiplicity of $C \in \mathcal{V}, C = \delta(D)$ $(D \in \mathcal{D})$, to be $y(C) := y(D)$.

Comments.

• The first step of Algorithm 1 associates the execution of a matching algorithm with each point $x \neq x_0$ (see Lemma 3.1) of step 0. The matching subroutine carried out in step 0 can be associated with x_0, so a matching algorithm has been associated with each $x \in V(G)$.

In fact, there is no real assymmetry between steps 0 and 1, in other words, between x_0 and the other vertices of G. The assymetry disappears as soon as we consider a somewhat more general object than graphs (see "towers" in Sebő [1990]).

• For arbitrary real weights, Algorithm 1 works without any change; the only difference is that the function λ defined in step 2 will not be an integer function. We prefer to assume that the weights are integer and often that they are bipartite because then λ is also an integer function and the integrality results that we obtain are included in a natural way.

THEOREM 3.1.

(a) *$\mathcal{V}$ with multiplicities $y(V)$ $(V \in \mathcal{V})$ is a dual flow.*

(b) *$\mathcal{V}$ and $y(V)$ $(V \in \mathcal{V})$ can be determined by first computing the distances in $(G, |w|)$, then using n parallel running and noncommunicating matching subroutines, and then executing at most $n + m$ subroutines that find the connected components of a graph. (These can also be executed in parallel and without communication.) The inputs of the subroutines are graphs on at most n points, and in the input of the matching algorithm, every weight is the sum of the weights $w(e)$ of at most n edges $e \in E(G)$.*

We will assume throughout the paper that the complexity of computing the distances (in parallel or nonparallel) in a graph with nonnegative weights does not exceed that of the (parallel or nonparallel) matching algorithms, whence it can be neglected.

Proof of Theorem 3.1. (a) can immediately be seen from (2.3) (that is, Theorem 2.1), and it is an immediate consequence of Lemma 3.1 and 3.2 that Algorithm 1 satisfies the properties described in (b). □

COROLLARY 1. *Suppose that we have a weighted matching algorithm whose running time is $t(n)$ for an input of n points, and suppose that it uses $p(n)$ processors. Then either a dual multiflow or a negative circuit can also be determined in $t(n)$ time and with $np(n)$ processors.*

According to the results of Mulmuley, Vazirani, and Vazirani [1986], a maximum matching can be determined in $O(\log^2 n)$ time with a random parallel algorithm. The same article solves weighted matching problems for particular weights, but we do not know of any general efficient parallel algorithm that solves this problem. However, Mulmuley, Vazirani, and Vazirani note that the general weighted problem is also in RNC^2 if the encoding of the weights is unary. Wein [1991] developed a Las Vegas RNC algorithm for minimum-weight perfect matchings which is logarithmic polynomial if the encoding of the weights is unary. Through Theorem 2.1, we get the same results for dual multiflows and all of their applications. (Wein's [1991] result itself uses Theorem 2.1.)

In Algorithm 1, Theorem 3.1, and Corollary 1, instead of "dual flow," we could of course have written "maximum T-cut packing" for a (G, T) and a weighting $w : E(G) \to \mathbb{Z}_+$. Furthermore, Algorithm 1 always defines *the uniquely existing patch packing.* Thus for every application of patch packings, we have the following corollary:

In planar networks, with the "dualization" described in section 1 and within the time limits in Corollary 1, Algorithm 1 either finds a cut that violates the cut condition

or finds a feasible flow. The "dual distances" (see section 1) taken from the infinite face can be determined by Algorithm 1, and in this way our result will be the uniquely existing patch flow. The same holds for the Ising model (see Barahona [1980]). Thus for these problems as well, a structural decomposition is implied. In the case of an Eulerian graph, this will automatically be an integer flow.

Of course, to get the best complexity results, one should substitute into Corollary 1 the best possible complexity results that exploit planarity. Finding a logarithmic polynomial parallel matching algorithm is still an open problem. The best known complexity for matchings in planar graphs is by Lipton and Tarjan [1980]: $O(n^{3/2}\log n)$. It follows that the sequential complexity of finding a patch flow in planar graphs—and of all the planar problems mentioned above—is $O(n^{5/2}\log n)$.

Nishizeki, Matsumoto, and Saito [1986] also solved planar flow problems within this time limit via n matching algorithms; however, these must be successive. In their algorithm, the input of every matching subroutine depends on the output of all subroutines carried out previously, and the output of the whole algorithm depends on the choices made during the running of the algorithm.

Note that the best sequential complexity for the planar Chinese postman problem has thus far been reached by Barahona [1989] with a direct algorithm that has the same complexity as the best planar matching algorithm at present, $O(n^{3/2}\log n)$.

We now mention some applications that require more than a simple use of Theorems 2.1 and 3.1.

3.1. Integer packings in graphs without odd K_4.

(3.1) (Seymour [1977]) *Let G be a graph and suppose that $T \subseteq V(G)$, where $|T|$ is even. If $V(G)$ cannot be partitioned into four T-odd parts so that each induces a connected graph, and if there exists an edge between any two of the four parts, then the maximum cardinality of a family of disjoint T-cuts is equal to the minimum cardinality of a T-join.*

Since the constraint of (3.1) remains true after the usual subdivision (or contraction) of edges, the weighted generalization can be straightforwardly deduced. The condition of (3.1) is not only sufficient but also necessary to have a maximum w-packing of T-cuts that is integer for an arbitrary nonnegative integer weight function w.

(3.1) is an important special case of Seymour's general characterization of binary clutters with the strong max-flow min-cut property. It is closely related to the following result. (Again, for the sake of simplicity, we state only the cardinality case, which implies the general case through the usual subdivision of edges.)

If $\mathcal{P}$ is a partition of $V(G)$ into T-odd parts each of which induces a connected graph, then the set of edges whose endpoints are in different parts of $\mathcal{P}$ is called a *T-border*. The *value* of this T-border is $|\mathcal{P}|/2$. A T-border B is called *bicritical* if, upon contracting all the edges of $E(G) \setminus B$ (that is, shrinking all the classes of $\mathcal{P}$), we get a bicritical graph, that is, a graph that has a perfect matching, and according to the weight function that is -1 on a perfect matching and 1 everywhere else, the distance between any two points is -1. (It is an easy exercise to show that this definition does not depend on the chosen matching.)

(3.2) (Sebő [1988]) *Let G be a graph and $T \subseteq V(G)$, where $|T|$ is even. The minimum cardinality of a T-join of G is equal to the maximum sum of the values of a set of edge-disjoint bicritical T-borders.*

Again, (3.2) implies its own weighted generalization in the usual way. (3.2) implies (3.1) using the following.

(3.3) (Sebő [1988]) *The vertices of a bicritical graph can be partitioned into four classes of odd cardinality so that, upon contracting all edges with both endpoints in the same class, we get* K_4.

The original proof of (3.3) reduces the statement to Seymour's theorem and points out that an elementary proof would generate a simple proof of Seymour's theorem via (3.2). A simple proof of (3.3) was given by Lovász through the ear decomposition of nonbipartite matching-covered graphs and by Gerards [1987] in an elementary way. For an elegant, elementary proof of (3.3), see Frank and Szigeti [1994].

The original proof (see Sebő [1987a]) of (3.2) used Theorem 1.1, and for the sake of the algorithmic consequences, this is what we must follow here. (In Sebő [1988], the same proof was described in a self-contained way by substituting the proof of Theorem 1.1 instead of using it. Frank and Szigeti [1994] replaced Theorem 1.1 by using (1.4), which combined with their proof of (3.3) is the shortest variant.)

Proof of (3.2). It is trivial that the minimum is greater than or equal to the maximum. To prove equality, let F be a minimum T-join and define $w(e) := -1$ if $e \in F$ and $w(e) := 1$ if $e \in E(G) \setminus F$. By Guan's remark (see section 1), (G, w) is conservative. We can suppose without loss of generality that G is connected. Let $x_0 \in V(G)$ be arbitrary. Define $\mathcal{D}$, $\mathcal{D}'$, and $\mathcal{D}''$ as we did immediately after (2.3), and apply (2.3).

As we noticed after (2.3), each $D \in \mathcal{D}''$ is partitioned by some elements $D_1, \dots, D_k$ of $\mathcal{D}'$. If $x_0 \notin D$, then $x_0 \notin D_1 \in \mathcal{D}, \dots, x_0 \notin D_k \in \mathcal{D}$, and by (2.3a), $\delta(D_i)$ contains exactly one negative edge for $i = 1, \dots, k$, and since $\delta(V(G) \setminus D) = \delta(D)$, so does $\delta(V(G) \setminus D)$. $G - D$ may be disconnected, but for one of its components—denote it by D_0—$\delta(D_0)$ contains the unique negative edge of $\delta(D)$. Add each component of $G - D$ except D_0 to one of the D_i's ($i = 1, \dots, k$) adjacent to it. (Since G is connected, there exists at least one such D_i for every component of $G - D$.) In this way, we get a partition $\mathcal{P}(D) := \{D_0, D'_1, \dots, D'_k\}$ whose classes are matched by $(k+1)/2$ edges of F, and there is no other edge of F in the set $B(D)$ of edges joining different classes of $\mathcal{P}(D)$. Thus B(D) is a T-border.

Moreover, $\mathcal{B} := \{B(D) : D \in \mathcal{D}'', x_0 \notin D\}$ is a set of disjoint T-borders whose sum of values is equal to $|F|$. This proves (3.2) without proving that these T-borders are bicritical. If some of the T-borders $B \in \mathcal{B}$ are not bicritical, we will *find a larger number of disjoint* T*-borders with the same value.* This will complete the proof of (3.2) because the number of disjoint T-borders is bounded by the number of edges, so it has a maximum, and then all T-borders are bicritical.

Let B^* be the graph that we obtain by contracting $E(G) \setminus B$ (that is, by shrinking the classes of the underlying partition). $F \cap B$ is a perfect matching of B^*, and the weighting w becomes w^*: -1 on F and 1 elsewhere. (B^*, w^*) is conservative. Suppose that B^* is not bicritical; let $x_0^* \neq x^* \in V(B^*)$ and $\lambda_{B^*,w^*}(x_0^*, x^*) \geq 0$.

Apply to (B^*, w^*) the argument that we used for (G, w) to find a set of disjoint $V(B^*)$-borders of maximum sum of values. We get a set $\mathcal{B}^*$ of disjoint T-borders of total value $|B \cap F|$, the same as the value of B. The T-border in $\mathcal{B}^*$ containing the negative edge e adjacent to x_0^* contains only edges adjacent to vertices at distance -1, whence it does not contain the negative edge adjacent to x^*. Thus $|\mathcal{B}^*| \geq 2$, as claimed. □

The algorithm that follows from this proof for finding the integer packing of (3.2) is the following.

ALGORITHM 2.

Input: a graph G and $w : E(G) \to \mathbb{N}$.

Output: a w-packing of T-borders (T-cuts if the condition of (3.1) is satisfied).

1. Find a patch dual flow (see Algorithm 1).

2. Find a w-packing of T-borders using the guidelines of the first part of the proof of (3.2).

Comment. Of course, the proof must be applied to the graph where each $e \in E(G)$ is subdivided into a path of $w(e)$ edges, and if the algorithm really does the subdivision, the polynomial bound is lost. However, from the multiplicities of the cuts in the dual flow provided by Algorithm 1, it is straightforward to compute the multiplicities of the T-borders in the packing without actually doing the subdivision.

3. Decompose each T-border B of the constructed packing into disjoint bicritical T-borders using the guidelines of the second part of the proof of (3.2), and replace B with the elements of the decomposition, assigning its multiplicity to each. Continue this procedure with each of the newly constructed T-borders until there is no more proper decomposition, that is, until the distance between any two vertices in all of the T-borders is -1 (according to the weight function defined in the proof of (3.2)), i.e., until they are all bicritical.

Comment. This step is the same in the weighted and unweighted cases. The decomposition of a T-border into bicritical T-borders is an *unweighted* problem.

THEOREM 3.2. *The integer packing of odd cuts in* (3.2) *and* (3.1) *can be found by solving* $O(mn^2)$ *matching algorithms on minors of* G (*graphs that arise with the contraction and deletion of some edges from* G) *and performing some additional steps whose order of magnitude is smaller.*

Proof. To execute steps 1 and 2, the number of matching algorithms to be solved is n. After the execution of step 3, the number of bicritical T-borders that decompose a bicritical T-border is at most the number of edges of the latter. Thus m is an upper bound for the number of times Algorithm 1 must be applied to the subsequent T-borders. On each of these T-borders, we have to execute n^2 matching algorithms to prove that they are bicritical or decompose them. □

It follows that the maximum integral dual solution to the minimization problem on a T-join polyhedron described with a minimal totally dual integral (TDI) description (see Sebő [1988]) can also be computed within the same time limit.

However, note that the bounds in Theorem 3.2 are weaker than those in Corollary 1. We do not see how our parallel complexity estimates could be saved for this case.

3.2. Integer flows—almost. Let (G, w) be ± 1-weighted and conservative. Of course, E^- forms a forest. Korach and Penn [1992] proved that there exist pairwise-disjoint cuts, each of which contains one negative edge, so that every negative edge is in some of the cuts except perhaps at most one edge of each component of E^-; in fact, for one component of E^-, one can require every edge to be in some of the cuts. Equivalently, for arbitrary weights, Korach and Penn proved the following theorem.

(3.4) *If* (G, w) *is conservative, then there exists* $F \subseteq E^-$ *with* $|F \cap C| \leq 1$ *for every connected component* C *of* E^- *such that for the weight function* w'*, there exists an integer dual flow, where* $w'(e) := w(e)$ *if* $e \in E \setminus F$ *and* $w'(e) := w(e) - 1$ *if* $e \in F$.

We give the proof of Sebő [1990], which provides an algorithm for finding this dual flow and the multiflow of Corollary 2 below via Theorem 2.1, with the time limits of Theorem 3.1.

Proof of (3.4). The usual subdivision of edges leads to a ± 1-weighted conservative graph, so suppose that (G, w) is already one. Let $x_0 \in V(G)$ be arbitrary and $\lambda(x) := \lambda_w(x_0, x)$ $(x \in V(G))$. Denote the components of E^- by $E_0, \ldots, E_k$.

CLAIM 3. *For every $x \in V(E_i)$, there exists at most one $y \in V(E_i)$ with $xy \in E_i$ such that $\lambda(y) \geq \lambda(x)$ $(i = 1, \ldots, k)$.*

Indeed, let $xy_1, xy_2 \in E_i$, $y_1 \neq y_2$. A shortest simple (x_0, x) path P contains at most one of these edges—say, it does not contain the edge xy_2. Then, however, $P \cup xy_2$ is an (x_0, y_2) path and $\lambda(y_2) \leq w(P \cup xy_2) = w(P) - 1 = \lambda(x) - 1$.

CLAIM 4. *The maximum of $\lambda(x)$ on $V(E_i)$ is reached on one vertex or the two endpoints of an edge $(i = 1, \ldots, k)$. For every edge $uv \in E_i$ but this one, $|\lambda(u) - \lambda(v)| = 1$.*

Indeed, if there are two nonadjacent vertices of E_i with $\lambda(x)$ maximum, then these two vertices are joined by a subpath of E_i, which, if it consists of at least two edges, must contain a vertex, contradicting the claim. If $|\lambda(u) - \lambda(v)| = 0$, then in exactly the same way, we get from Claim 3 that $\lambda(u) = \lambda(v)$ are the only maxima of $\lambda(x)$, $x \in V(E_i)$.

Thus each E_i $(i = 1, \ldots, k)$ contains at most one edge xy with $\lambda(x) = \lambda(y)$. According to (2.4), $\{\delta(D) : D \in \mathcal{D}'', x_0 \notin D\}$ is a set of disjoint cuts which contains every negative edge except those with $\lambda(x) = \lambda(y)$, that is, there is at most one exception in each E_i $(i = 1, \ldots, k)$ (see Claim 4). □

The following statement of Korach and Penn [1992] is an immediate corollary.

COROLLARY 2. *If (G, c) is an instance of the planar multiflow problem and the cut condition is satisfied, then upon decreasing all but one of the demands by 1 (that is, increasing all the negative capacities by 1), there exists an integer multiflow.*

Korach and Penn also noticed that one can, in fact, require $F \cap E_0 = \emptyset$ for any given component E_0 of E^-. Indeed, in the above proof as well, the choice $x_0 \in V(E_0)$ makes sure that in Claim 4, $\lambda(x_0) = 0$, and for every other vertex of E_0, $\lambda(x) < 0$. In E_0 there is no edge xy with $\lambda(x) = \lambda(y)$.

Frank and Szigeti [1995] generalized (3.4) in the following way.

(3.5) *Let G be a graph, $w : E(G) \to \mathbb{Z}$, and $E_0, E_1, \ldots, E_k, E_{k+1}, \ldots, E_{k+l}$ be the components of E^-. Suppose that $w(C) \geq s(C)$ holds for every circuit C of G, where $s(C) := |\{i \in \{1, \ldots, k\} : E_i \cap C \neq \emptyset\}|$.*

Then there exists a set $F \subseteq E_{k+1} \cup \cdots \cup E_{k+l}$ such that $|F \cap E_i| \leq 1$ if $i = k+1, \ldots, k+l$, with the property that, upon decreasing the demand of $e \in E_i \cap F$ by 1 $(i = k+1, \ldots, k+l)$, there exists a dual multiflow.

The proof of Frank and Szigeti consists of splitting every vertex $v \in E_1 \cup \cdots \cup E_k$ into two vertices v_1 and v_2, where v_1 is incident to the positive edges and v_2 is incident to the negative edges of $\delta(v)$. v_1 and v_2 are joined by an edge of weight $-1/2$. If the condition of (3.5) is satisfied, then the constructed graph with the constructed weighting is conservative, and applying the proof of (3.4) to this graph with this weight, we can obtain the dual multiflow stated in (3.5).

Frank and Szigeti [1995] also observed the following immediate corollary.

COROLLARY 3. *If (G, w) is conservative and $w(\delta(X)) + w(\delta(X) \cap E^-) \geq 0$, then there exists an integer dual multiflow.*

Theorem 3.1 then implies the following:

The integer (dual) multiflow in (3.4), (3.5), and Corollary 3 can be found within the same time limits as those in Corollary 1. Of course, the same holds for integer multiflows in planar graphs.

3.3. Packing cuts exactly. Finally, let us comment on the complexity of integer odd cut packings. Let G be a graph and $T \subseteq V(G)$, where the number of vertices of T in each component of G is even. (G, T) is said to have the *Seymour property* if the minimum cardinality of a T-join is equal to the maximum cardinality of a set

of disjoint T-cuts. Accordingly, (G, w) $(w : E(G) \to \mathbb{Z})$ has the Seymour property if there exists an integer dual flow. Middendorf and Pfeiffer [1989] proved that it is NP-complete to test the Seymour property, even in planar graphs.

Therefore, instead of the Seymour property, our interest turns towards Seymour graphs. G is called a Seymour graph if for arbitrary $w : E(G) \to \{-1, 1\}$ for which the cut condition holds, there exists an integer dual multiflow. Through the subdivision of edges, the weighted generalization is the following: G is a Seymour graph with respect to the weight function $w : E(G) \to \mathbb{N}$ if for an arbitrary $w' : E(G) \to [-|w(e)|, w(e)]$ such that $w'(e) \equiv w(e) \bmod 2$ and the cut condition holds, there exists an integer dual multiflow. (This is the class of weight functions that one gets by independently signing the parts of a subdivided edge. In the following, we will restrict ourselves to unweighted Seymour graphs since the generalization is trivial and somewhat artificial.) It is not known whether the problem of deciding whether a graph G is a Seymour graph is polynomially solvable or coNP-complete. (It is not trivial but true that it is in coNP, as we shall see later.) However, the following somewhat weaker results are known:

As Seymour's result (1.3) implies that bipartite graphs are Seymour graphs, so does (3.1) imply that series-parallel graphs are Seymour graphs. (The latter statement can also be easily proved directly without using (3.1).) Gerards [1992] proved a common generalization of these two results. A sharpening of this that provides a coNP characterization of Seymour graphs was conjectured by Sebő [1991] and proved by Ageev, Kostochka, and Szigeti [1995]. The following simplified version still provides a polynomially checkable obstacle, and it suffices for our purposes.

(3.6) *G is not a Seymour graph if and only if there exists a signing of the edges such that the union of zero-weight circuits* (*in fact, of only two zero-weight circuits*) *is nonbipartite.*

The proof of the *if* part of this conjecture is a straightforward exercise.

The proof of the *only if* part given by Ageev, Kostochka, and Szigeti is based on Theorem 1.1 in such a way that Algorithm 1 can be straightforwardly substituted in it, and the corresponding polynomial bounds follow easily:

Given the graph G and $w : E(G) \to \{-1, 1\}$, there is a polynomial algorithm which either (a) finds an integer dual multiflow in (G, w) or a negative circuit or (b) exhibits a weight function $w' : E(G) \to \{-1, 1\}$ for which (G, w') is conservative but no integer dual multiflow exists. In the latter case, a "good certificate" (a nonbipartite graph and two zero-weight circuits covering all the edges) is provided.

As a consequence, *the unweighted dual multiflow problem is polynomially solvable in Seymour graphs*, and all of the consequences provided by the Corollary 1 also hold. Let us finally state the specialization to planar graphs (after dualization):

Given the graph $G = (V, E)$ and $R \subseteq E$, there is a polynomial algorithm which either (a) finds an integer multiflow in (G, R) or a violated cut or (b) exhibits $R' \subseteq E$ for which the cut condition is satisfied (certificate: a half-integer multiflow) but no integer multiflow exists (certificate: two zero-weight cuts whose union is odd).

As a consequence, *the edge-disjoint paths problem is polynomially solvable in dual Seymour planar graphs.* The main open problem of whether Seymour graphs are in NP and whether they can be recognized in polynomial time remains.

3.4. Weighted and canonical matching and T-join structure. If $V(G)$ is odd, a matching M will be called *perfect* if it leaves exactly one vertex uncovered. The maximum- (or minimum-) weight matching problem can be easily reduced to the problem of finding minimum-weight perfect matchings. If $w : E(G) \to \mathbb{R}$, we will

denote by $\tau(G, w)$ the minimum weight of a perfect matching in G.

It is now easy to deduce the consequences of Theorems 2.1 and 3.1 to weighted matchings:

Add a large number N (say, N is the sum of the absolute values of the weights) to the weight of every edge. Add a new vertex x_0 to G, and join it to every vertex with an edge of weight N. Let G' denote this new graph and w' denote the weight function that we defined on its edges. It is easy to see that the w'-minimum $T := V(G')$-joins (or $T := V(G') \setminus \{x_0\}$-joins, depending on which of the two is even) of G' are exactly the w-minimum perfect matchings of G (after deleting the edge adjacent to x_0 if $|V(G)|$ is odd).

Choose a minimum-weight perfect matching of G or G' (depending on whether $|V(G)|$ or $|V(G')|$ is even), and change the sign of the weights of the edges in it.

It is now easy to see that the distance of the vertex $x \in V(G)$ from x_0 is the number $\tau(G - x, w) - \tau(G, w)$. Note that it is independent of the particular w-minimum matching chosen.

From a maximum 2-packing of T-cuts in G', a maximum odd cut packing of G can also be reconstructed. That is, Theorem 1.1 can be adapted to weighted matchings. The "magic" numbers should be defined to be $\tau(G - x, w) - \tau(G, w)$. These numbers will be equal to the distances with the chosen weights; these independently computable numbers determine a packing of odd cuts, which could be called a "patch packing" and which generalizes the Gallai–Edmonds structure of maximum matchings. (See more about the relation of Theorem 1.1 and the Gallai–Edmonds structure theorem in Sebő [1990].)

Algorithm 1 and Lemmas 3.1 and 3.2 can be applied to other problems involving integer packings in a similar way. We mention some results whose proofs involve Theorem 1.1 (or 2.1) so that they can be accompanied by a polynomial algorithm that combines Lemmas 3.1 and 3.2 and Algorithm 1.

A generalization of the Kotzig–Lovász theorem (Sebő [1987a, b]) provides a "canonical partition" (implying new results on integer feasible flows or the computation of the dimension of T-join polyhedra or multiflows). This canonical partition can be computed via the distances, that is, with Lemmas 3.1 and 3.2. Thus the integer packings and flows of Sebő [1990], or perhaps a negative cut or an odd circuit consisting of tight edges, or a set violating the cut condition or some stronger condition (see Sebő [1987b]) can also be found in polynomial time.

Acknowledgments. I am thankful to András Frank for his encouragement and to Zoltán Szigeti and an anonymous referee for a lot of helpful advice.

REFERENCES

A. Ageev, A. Kostochka, and Z. Szigeti [1995], *A characterization of Seymour graphs*, J. Graph Theory, to appear.

F. Barahona [1980], *Application de l'optimisation combinatoire á certains modéles de verres de spin: Complexité et simulations*, thèse de docteur ingénieur, Université Scientifique et Médicale de Grenoble, Institut National Polytechnique de Grenoble, Grenoble, France.

F. Barahona [1989], *Planar multicommodity flows: Max cut and the Chinese postman problem*, in Polyhedral Combinatorics, W. Cook and P. Seymour, eds., DIMACS Series in Discrete Mathematics and Computer Science, Vol. 1, AMS, Providence, RI, pp.189–202.

F. Barahona and W. Cunningham [1989], *On dual integrality in matching problems*, Oper. Res. Lett., 8, pp. 245–249.

J. Edmonds [1965a], *Paths, trees and flowers,* Canad. J. Math., 17, pp. 449–467.

J. Edmonds [1965b], *The Chinese postman problem,* Oper. Res., 13 (supplement 1), p. 373.

J. Edmonds and E. L. Johnson [1973], *Matching, Euler tours and the Chinese postman*, Math. Programming, 5, pp. 88–124.
A. Frank [1990], *Packing paths, circuits and cuts: A survey,* in Paths, Flows and VLSI-Layout, B. Korte, L. Lovász, H. Prömel, and A. Schrijver, eds., Springer-Verlag, Berlin, Heidelberg, pp. 47–100.
A. Frank, A. Sebő, and É. Tardos [1984], *Covering directed and odd cuts,* Math. Programming Stud., 22, pp. 99–112.
A. Frank and Z. Szigeti [1994], *On packing T-cuts*, J. Combin. Theory Ser. B, 61, pp. 263–271.
A. Frank and Z. Szigeti [1995], *A note on packing paths in planar graphs*, Math. Programming, 70, pp. 201–209.
A. Gerards [1987], private communication.
A. Gerards [1992], *On shortest T-joins and packing T-cuts*, J. Combin. Theory Ser. B, 55, pp. 73–82.
M. Guan [1962], *Graphic programming using odd or even points*, Chinese J. Math., 1, pp. 273–277.
E. Korach [1982], *Packing of T-cuts, and other aspects of dual integrality,* Ph.D. thesis, Waterloo University, Waterloo, ON, Canada.
E. Korach and M. Penn [1992], *Tight integral duality gap in the Chinese postman problem,* Math. Programming, 55, pp. 183–191.
E. Lawler [1976], *Combinatorial Optimization, Networks and Matroids,* Holt, Rinehart, and Winston, New York.
R. J. Lipton and R. E. Tarjan [1980], *Applications of a planar separator theorem,* SIAM J. Comput., 9, pp. 615–627.
L. Lovász [1975], 2*-matchings and* 2*-covers of hypergraphs,* Acta Math. Acad. Sci. Hungar., 26, pp. 433–444.
L. Lovász and M. Plummer [1986a], *On bicritical graphs*, in Infinte and Finite Sets, A. Hajnal, R. Rado, V. T. Sós, et al., eds., Colloq. Math. Soc. János Bolyai, North–Holland, Amsterdam, pp. 1051–1079.
L. Lovász and M. Plummer [1986b], *Matching Theory*, Akadémiai Kiadó, Budapest.
K. Matsumoto, T. Nishizeki, and N. Saito [1986], *Plane multicommodity flows, maximum matchings and negative cycles*, SIAM J. Comput., 15, pp. 495–510.
M. Middendorf and F. Pfeiffer [1989], *On the complexity of the disjoint path problem*, in Polyhedral Combinatorics, W. Cook and P. Seymour, eds., DIMACS Series in Discrete Mathematics and Computer Science, Vol. 1, AMS, Providence, RI, pp. 171–178.
K. Mulmuley, U. V. Vazirani, and V. V. Vazirani [1987], *A parallel algorithm for matching,* Combinatorica, 7, pp. 105–113.
A. Schrijver [1986], *Theory of Linear and Integer Programming*, John Wiley, Chichester, UK.
A. Sebő [1987a], *The factors of graphs: Structures and algorithms,* Ph.D. thesis, Eötvös Loránd University, Budapest.
A. Sebő [1987b], *Dual integrality and multicommodity flows,* in Infinite and Finite Sets, A. Hajnal and V. T. Sós, eds., Colloq. Math. Soc. János Bolyai, North–Holland, Amsterdam, pp. 453–469.
A. Sebő [1988], *The Schrijver system of odd join polyhedra*, Combinatorica, 8, pp. 103–116.
A. Sebő [1990], *Undirected distances and the postman-structure of graphs,* J. Combin. Theory Ser. B, 49, pp. 10–39.
A. Sebő [1991], *On two multiflow problems*, lecture, Graph Minors Meeting, Seattle.
P. D. Seymour [1977], *The matroids with the max-flow min-cut property*, J. Combin. Theory Ser. B, 23, pp. 189–222.
P. D. Seymour [1981], *On odd cuts and plane multicommodity flows*, Proc. London. Math. Soc., 42, pp. 178–192.
J. Wein [1991], *Las Vegas RNC algorithm for weighted matchings*, Inform. Process. Lett., 40, pp. 161–167.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 605–633, June 1997

001

BOOLEAN CIRCUITS, TENSOR RANKS, AND COMMUNICATION COMPLEXITY *

PAVEL PUDLÁK†, VOJTĚCH RÖDL‡, AND JIŘÍ SGALL§

Abstract. We investigate two methods for proving lower bounds on the size of small-depth circuits, namely the approaches based on multiparty communication games and algebraic characterizations extending the concepts of the tensor rank and rigidity of matrices. Our methods are combinatorial, but we think that our main contribution concerns the algebraic concepts used in this area (tensor ranks and rigidity). Our main results are following.

(i) An $o(n)$-bit protocol for a communication game for computing shifts, which also gives an upper bound of $o(n^2)$ on the contact rank of the tensor of multiplication of polynomials; this disproves some earlier conjectures. A related probabilistic construction gives an $o(n)$ upper bound for computing all permutations and an $O(n \log\log n)$ upper bound on the communication complexity of pointer jumping with permutations.

(ii) A lower bound on certain restricted circuits of depth 2 which are related to the problem of proving a superlinear lower bound on the size of logarithmic-depth circuits; this bound has interpretations both as a lower bound on the rigidity of the tensor of multiplication of polynomials and as a lower bound on the communication needed to compute the shift function in a restricted model.

(iii) An upper bound on Boolean circuits of depth 2 for computing shifts and, more generally, all permutations; this shows that such circuits are more efficient than the model based on sending bits along vertex-disjoint paths.

Key words. circuit, tensor rank, communication complexity, random graph

AMS subject classifications. 68Q15, 68Q25, 68R10

PII. S0097539794264809

1. Introduction.

The problem of proving superlinear lower bounds on the size of circuits for an explicitly defined sequence of Boolean functions is perhaps the most persistent problem in complexity theory. Attempts to solve it have led to several weaker problems which are often of independent interest. The problem is still open even if we look for functions of n input bits with n outputs and impose an additional restriction that the depth of the circuit is $O(\log n)$.

The main motivation for this paper is a related problem in which the computation is restricted to *linear circuits* over a field $\mathbf{F}$, i.e., circuits which have linear functions as gates. The question is to find an explicit linear function which cannot be computed by a linear circuit of size $O(n)$ and depth $O(\log n)$. Clearly, such a function must have a nonconstant number of output bits since every one-output linear function can be computed by a balanced tree with linear gates. If the field is GF_2, this is a variant of the problem for general Boolean circuits. However, strictly speaking, the problem

* Received by the editors March 18, 1994; accepted for publication (in revised form) June 6, 1995. http://www.siam.org/journals/sicomp/26-3/26480.html

† Mathematical Institute, Academy of Sciences of the Czech Republic, Žitná 25, 11567 Praha (Prague) 1, Czech Republic (pudlak@mbox.cesnet.cz). Part of this research was done while this author was visiting the Department of Mathematics and Computer Science, Emory University, Atlanta, GA, 30322.

‡ Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322 (rodl@mathcs.emory.edu). Part of this research was done while this author was visiting the Mathematical Institute AV ČR, Žitná 25, 11567 Praha (Prague) 1, Czech Republic.

§ Mathematical Institute, Academy of Sciences of the Czech Republic, Žitná 25, 11567 Praha (Prague) 1, Czech Republic (sgallj@mbox.cesnet.cz). The research of this author was done at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 and partially supported by AV ČR grant A119107 and US–Czechoslovak Science and Technology Program grant 93 025.

about linear circuits over GF_2 is not a weaker problem since the class of computable functions is smaller.

Many natural functions are not linear in all of the inputs, but only in some subset of them. For example, matrix multiplication is a bilinear function, which means that whenever one of the matrices is fixed, it is a linear function of the entries of the other matrix. A natural extension of linear circuits which includes circuits for such functions are *semilinear circuits*. Suppose that a function $F(p, \vec{x})$ is linear for every fixed parameter p. We say that a circuit C (with $\vec{x}$ as an input) is a *semilinear circuit* for $F(p, \vec{x})$ if for any fixed parameter p, we can assign linear functions to the gates of C so that we get a linear circuit for $F(p, \vec{x})$. We want to prove a lower bound on the size of a semilinear circuit computing an explicit $F(p, \vec{x})$.

Thus far, superlinear lower bounds have been proved only for infinite fields. Shoup and Smolensky proved a lower bound of $\Omega(n \log n / \log \log n)$ for linear circuits of poly-logarithmic depth [30]; however, their proof works only for functions with very large values. For finite fields, no explicit functions not computable by linear or semilinear circuits of size $O(n)$ and depth $O(\log n)$ are known.

Some superlinear lower bounds have been proved for a much more restricted model of constant-depth circuits with linear gates of unbounded fan-in. (This is again nontrivial only for multioutput functions.) Some of these bounds use results on the complexity of communication networks [13, 24, 26]. The bounds can be extended to bounded-depth Boolean circuits with arbitrary Boolean functions as gates, which proves, for instance, that addition and multiplication cannot be computed by a constant-depth circuit of size $O(n)$.

We are interested in various algebraic and combinatorial concepts related to circuit complexity. First, we review several such concepts that have been devised for proving circuit lower bounds.

The algebraic approach dates back to Strassen, who introduced the concept of the *rank of a tensor* [31]. This rank characterizes up to a constant factor the number of multiplications needed to compute an explicit bilinear function, which is called multiplicative complexity. It is a major open problem in algebraic complexity to prove a superlinear lower bound on the rank of an explicit tensor. Valiant defined the *rigidity of a matrix* [32]. Sufficiently large lower bounds on the rigidity of the matrix defining a linear function would imply superlinear lower bounds on the size of linear circuits of depth $O(\log n)$. However, thus far, the best bounds for explicit matrices are too small [14]. Later, Razborov considered a modification of the tensor rank, called the *contact rank* [27]. This rank characterizes the complexity of certain restricted algebraic circuits (where only multiplications by a variable or a scalar are allowed). He proved a lower bound of $\Omega(n^{3/2})$ on the contact rank of the tensor of multiplication of polynomials and on the contact rank of the tensor of matrix multiplication. He also used the contact rank to prove a lower bound on the rigidity of certain matrices. The combinatorial approach (which can be also characterized as information-theoretic) leads to the *multiparty communication complexity* introduced by Chandra, Furst, and Lipton [16], which was used to prove lower bounds on circuit complexity in various situations [7, 15, 20], and to the concept of computation with *common bits* introduced by Valiant [32, 33].

In this paper, we generalize the concept of the rigidity of a matrix to the *rigidity of a tensor*, where a tensor is essentially a set of matrices. This gives a tool for proving lower bounds on semilinear circuits in the same way as the original concept does for linear circuits. We also define another variant of tensor rank, which we call the

rigidity rank because of its relation to the rigidity of tensors. Furthermore, we define certain refinements of the multiparty communication complexity and computation with common bits and show that they are closely related to the rigidity rank. Thus some bounds can be transferred from communication complexity to the algebraic framework and vice versa.

We consider the complexity of computing all n cyclic shifts of input bits. We prove an $o(n)$ *upper bound* on the corresponding three-party communication game, thus giving a negative answer to a communication complexity question of Nisan and Wigderson motivated by the problem of circuits of depth $O(\log n)$ and size $O(n)$ [20]. A similar problem about computation with common bits was also posed by Valiant [33]. Since each shift is a linear function, computing all shifts can be presented as a problem about semilinear circuits or, equivalently, as a problem about the rigidity of the diagonal tensor, which is the corresponding tensor. We get an upper bound of $o(n^2)$ on the rigidity rank of this tensor. This disproves a conjecture of Razborov [27] that the contact rank of this tensor is $\Omega(n^2)$, since the rigidity rank is at least as large as the contact rank.

This piece of information is quite important since it shows that certain direct approaches to the problem of superlinear lower bounds for Boolean circuits do not work. We have to use more refined notions, like the rigidity of a tensor instead of its contact and rigidity ranks, or special kinds of protocols in the case of communication complexity.

We generalize this upper bound in two ways. First, we consider all permutations instead of just shifts. Using probabilistic methods, we prove that the upper bounds of $o(n)$ on the communication complexity and $o(n^2)$ on the rigidity rank of the corresponding tensor hold even for this much harder problem. Second, for the communication complexity, we consider any constant number of players instead of just three, computing the composition of several shifts. In such a case, we prove that the communication complexity decreases exponentially with the number of players. These results do not exclude the possibility of applying the communication complexity and related combinatorial tools to the problem of proving a superlinear lower bound for Boolean circuits. However, they show that the usual intuition—that for simple functions the natural protocols are essentially the best possible—can be wrong.

Furthermore, we prove a *lower bound* on the size of circuits computing all shifts with common bits. In the algebraic language, this gives a lower bound on the rigidity of the diagonal tensor. As a consequence, we improve the current best lower bound on Boolean circuits of depth 2 with arbitrary gates computing multiplication of two integers (written in binary). We also give a slightly larger lower bound on the rigidity rank of the tensor of multiplication of polynomials than is known for contact rank. However, this bound is not strong enough to prove a lower bound for circuits of logarithmic depth.

Another restricted model of computation used for computing permutations of input bits are networks in which the inputs are routed through the network to the corresponding outputs along vertex disjoint paths [21, 23]. This can be viewed as a special case of semilinear circuits if we allow only projections as the gates instead of general linear functions. We consider the gap between the complexity of *conservative computation* by such networks and *nonconservative computation* by general semilinear circuits. We prove that it is possible to compute all permutations by a semilinear circuit which can be divided into two parts, one with a sublinear number of vertices and the other with $o(n^{3/2})$ edges. Using only routing along vertex-disjoint paths, such

networks cannot compute even all shifts. It is not surprising that circuits are more efficient, but as far as we know, this is the first proof of such a fact for the computation of a set of permutations.

One of our technical tools is Theorem 4.4 concerning *coloring random graphs*, which extends a famous estimate of the chromatic number of the random graph due to Bollobás [10] and may be of independent interest. We also use other powerful probabilistic techniques such as *martingales* and *Janson's inequality*.

Even though our tools are combinatorial and we obtain some lower bounds on the complexity of certain restricted computations, we think that the main contributions of this paper are the refinement of the algebraic concepts used in this area (tensor ranks and rigidity) and results obtained about them, together with the upper bounds on the multiparty communication complexity and on the size of depth-2 circuits.

This paper expands the results of the extended abstract [25] and the note [28] and contains some new results.

Independently of us, Babai, Kimmel, and Lokam [6] proved an $o(n)$ upper bound on the communication complexity of a game corresponding to computing certain set of n permutations (instead of n shifts). Very recently (during the refereeing process of this paper), Ambainis [4] improved some of our results on communication complexity (Theorems 4.2 and 4.9). In another recent paper, Damm, Jukna, and Sgall [11] prove bounds on the k-party communication complexity of the pointer-jumping function in a restricted model not directly corresponding to the models considered here.

Here is an *overview of the paper*. In section 2, we give the basic definitions and facts. Section 3 summarizes the relations between ranks of tensors, size of circuits, and communication games. In section 4, we prove the upper bounds on the communication complexity of computing all permutations and the iterated shift function, as well as the corresponding bound on tensor rigidity. The lower bounds on computation of shifts with common bits and the corresponding bounds on the rigidity and rigidity rank of the diagonal tensor are proved in section 5. In section 6, we prove the gap between conservative and nonconservative computation of shifts. The proof of the estimate on the chromatic number of a random graph is given in section 7.

2. Basic definitions and facts. We index all vectors and tensors starting from 0 since this is convenient for the modular arithmetic. An exception are some tensors that are indexed by a general parameter, which is not necessarily a number, in one coordinate.

2.1. Boolean functions. For circuit bounds, we consider *multioutput functions*, whose values are vectors of n bits. The *shift function*, $Shift^n : \{0,\dots,n-1\}\times\{0,1\}^n \to \{0,1\}^n$, is defined by

$$Shift^n(s,\vec{x}) = \vec{y}, \quad \text{where } y_i = x_{(i+s) \bmod n}.$$

This is a very important function since it can be reduced to many naturally occurring functions, e.g., multiplication of binary numbers, convolution, etc. We believe that linear-size circuits of logarithmic depth cannot compute the shift function, and hence cannot compute any of the functions to which it can be reduced.

If we use general permutations in place of shifts, we obtain the *permutation function*,

$$Perm^n(\pi,\vec{x}) = \vec{y}, \quad \text{where } y_i = x_{\pi(i)}$$

and π is a permutation of the set $\{0,\dots,n-1\}$. Note that the input size for $Perm^n$ is of the order of $n\log n$, while it is only $n+\lceil\log n\rceil$ for $Shift^n$.

Any multioutput function can be transformed into a *one-output function* with one extra argument that indexes the output bits. For any function $F(p, \vec{x}) : Y \times X \to \{0,1\}^n$, we define its one-output variant, $bit_F : \{0, \ldots, n-1\} \times Y \times X \to \{0,1\}$, by

$$bit_F(i, p, \vec{x}) = (F(p, \vec{x}))_i.$$

We use lower-case letters for one-output functions and capitalize the names of corresponding multioutput functions. It is easily seen that we can get a circuit for bit_F by adding only $O(n)$ new gates to the circuit for F so that the depth increases only by $O(\log n)$. As we will see later, the communication complexity of a one-output function is related to the circuit complexity of the corresponding multioutput function.

We define

$$shift^n = bit_{Shift^n}, \quad \text{i.e.,} \quad shift^n(i, j, \vec{x}) = x_{(i+j) \bmod n};$$

$$perm^n = bit_{Perm^n}, \quad \text{i.e.,} \quad perm^n(i, \pi, \vec{x}) = x_{\pi(i)}.$$

These two functions can be generalized to several shifts or permutations. We define

$$shift_k^n(s_1, \ldots, s_{k+1}, \vec{x}) = x_{(s_1 + \cdots + s_{k+1}) \bmod n};$$

$$perm_k^n(i, \pi_1, \ldots, \pi_k, \vec{x}) = x_{\pi_k \ldots \pi_1(i)}.$$

Note that $shift^n(i, j, \vec{x}) = shift_1^n(i, j, \vec{x})$ and $perm^n(i, \pi, \vec{x}) = perm_1^n(i, \pi, \vec{x})$.

All of the functions introduced above are linear in $\vec{x}$. We call such functions semilinear. More precisely, a function $F(p, \vec{x})$ is *semilinear* in $\vec{x}$ if for every fixed parameter p_0, the function $F(p_0, \vec{x})$ is a linear function of $\vec{x}$.

2.2. Circuits. The main motivation for our work is to study methods for proving superlinear lower bounds on the size of circuits with depth $O(\log n)$, where n is the number of inputs and all gates have fan-in 2. No such bounds are known even for functions with n outputs. In a Boolean circuit, the gates are arbitrary binary functions. We also consider algebraic circuits, in which case the gates are arbitrary polynomials over the given field.

Algebraic circuits over the field GF_2 and Boolean circuits are closely related, but there is a difference. In an algebraic circuit, we actually compute in $GF_2(x_1, \ldots, x_n)$, i.e., in GF_2 extended by indeterminates $x_1, \ldots, x_n$, while in a Boolean circuit we use only the elements of GF_2. A Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be identified with a multilinear polynomial p of $GF_2[x_1, \ldots, x_n]$. Then an algebraic circuit for p is a Boolean circuit for f. The converse, however, is not true. A polynomial p' obtained from a Boolean circuit C for f is equal to p only after factorizing by the ideal defined by the equations $x_1^2 = x_1, \ldots, x_n^2 = x_n$.

We are interested mainly in linear and semilinear functions. For that purpose it is natural to consider circuits in which the gates are linear functions. In the case of Boolean circuits, this means that the only gates allowed are parity, projections, and their negations. For linear circuits, the distinction between Boolean and algebraic circuits disappears.

In a circuit for a semilinear function $F(p, \vec{x})$, we consider p to be a parameter of each gate; therefore, it is natural to use the following definition. A *semilinear circuit* for a function $\vec{y} = F(p, \vec{x})$ is a directed acyclic graph with sources labeled by the

variables $\vec{x}$ and sinks labeled by the variables $\vec{y}$ such that for every fixed p, we can assign linear gates to the nodes so that the resulting circuit computes $F(p,\vec{x})$. We measure the dependence of the circuit size on $n = |x|$.

Every one-output semilinear function has a semilinear circuit of size n and depth $\log n$. However, for most multioutput linear functions $F : \{0,1\}^n \to \{0,1\}^n$, the size of a circuit computing F is $\Omega(n^2/\log n)$, and the same is true for semilinear functions and semilinear circuits. It is an open problem to prove that some explicitly defined Boolean semilinear function has no semilinear circuit of size $O(n)$ and depth $O(\log n)$.

An important example of semilinear circuits are algebraic circuits for bilinear functions, which include matrix multiplication, multiplication of polynomials, and convolution. Let $F(\vec{y},\vec{x}) = \sum a_{ij}y_ix_j$ be a bilinear function, let C be an algebraic circuit for $F(\vec{y},\vec{x})$. If we substitute arbitrary constants for $\vec{y}$, the circuit C computes a linear function. It is well known and easy to prove that it is possible to convert a general algebraic circuit for a linear function into a linear circuit computing the same function in such a way that the underlying graph is unchanged. (Let us stress that this does not hold for Boolean circuits; see [24].) Thus C, or, more precisely, the underlying graph, is a semilinear circuit for $F(\vec{y},\vec{x})$.

An important tool for studying the circuits of size $O(n)$ and depth $O(\log n)$ is a reduction discovered by Valiant [32]. We use the terminology that he introduced in [33].

Let F be a multioutput Boolean function with input variables $\vec{x} = (x_0,\dots,x_{n-1})$ and outputs $\vec{y} = (y_0,\dots,y_{n-1})$. Let G be a bipartite graph with nodes $x_0,\dots,x_{n-1}$ and $y_0,\dots,y_{n-1}$. We say that F *can be computed by the graph G with r common bits* if there exist Boolean functions $h_1,\dots,h_r$ and $g_0,\dots,g_{n-1}$ such that

$$y_i = (F(\vec{x}))_i = g_i(h_1(\vec{x}),\dots,h_r(\vec{x}),\vec{x}^{(i)}),$$

where $\vec{x}^{(i)}$ is the substring of input variables adjacent to y_i in G. In other words, F can be computed by a circuit of depth 2 where there are some direct connections from inputs to outputs given by the graph G and some connections through an intermediate level of r gates of unbounded fan-in. By the *degree* of the graph G, we mean the maximal degree of the nodes y_i in the graph G. By the *common bits*, we mean the functions $h_1,\dots,h_r$. We are interested in bounding the degree of G and the number of common bits.

In the analogy with semilinear circuits, we are especially interested in the case where the common bits and the functions g_i are semilinear; we then say that the computation by the graph *uses only semilinear functions*. In this restricted case, we can also use the same notion for algebraic circuits over a general field $\mathbf{F}$; the functions $h_1,\dots,h_r$ and $g_0,\dots,g_{n-1}$ are then required to be semilinear functions with values in $\mathbf{F}$, instead of Boolean functions.

The reduction is based on the following graph-theoretic fact proved in [32, Theorem 5.3].

THEOREM 2.1 (Valiant [32]). *For every $\varepsilon > 0$, c, and d, there exists K such that for any directed acyclic graph C with cn nodes and depth $d\log n$, there exists a set S of $Kn/\log\log n$ edges such that every directed path of length $\varepsilon\log n$ in C contains an edge from S.*

The consequence that is important for us is the following.

THEOREM 2.2 (Valiant [32, 33]). *For every $\varepsilon > 0$, c, and d, there exists K such that if a function F can be computed by a circuit of size cn and depth $d\log n$, then it can be computed by a graph G of degree at most n^ε with $Kn/\log\log n$ common bits.*

Moreover, if the original circuit is semilinear, then the computation by the graph G uses only semilinear functions.

Proof. Let K and S be as in Theorem 2.1. Define the common bits to be the functions computed at the edges from S by the circuit C. From Theorem 2.1, it follows that each output can be expressed as a circuit of depth at most $\varepsilon \log n$ with inputs from S and from the original inputs. Construct the graph G by connecting each output to these inputs. Because the gates have fan-in 2, this circuit depends on at most n^ε original inputs, and hence the degree of G is at most n^ε. The semilinearity is preserved since any function computed in a semilinear circuit is semilinear. □

Let us state explicitly what this reduction means for the shift function Shift^n. No superlinear lower bounds are known for computing explicitly given functions by semilinear circuits. In particular, it is an open problem whether it is possible to compute shifts by a graph G of degree n^ε, $\varepsilon < 1$, with $o(n)$ common bits using only semilinear functions. We show that it is possible to compute by a graph of degree $o(n)$ with $o(n)$ common bits not only all n shifts but in fact all $n!$ permutations. Since the degree of our graph is $\Theta(n \log\log n / \log n)$, which is much larger than n^ε, this does not solve the above open problem. However, it shows that it is possible to compute all permutations by a significantly smaller graph than was expected.

Valiant's reduction suggests that we consider Boolean circuits with gates of unbounded fan-in and constant depth, where the gates are arbitrary Boolean functions or arbitrary linear functions. The size of such a circuit is defined to be the number of edges. (This is a trivial notion for one-output functions.) For this case, some superlinear lower bounds for explicitly given Boolean functions are known; see [13, 21, 22]. These bounds are based on some graph properties of circuits computing particular Boolean functions. For instance, a function can easily be defined so that every circuit for the function is a superconcentrator. Similar graph-theoretic arguments can also be used for the circuits of unbounded fan-in and small depth computing the function Shift^n [26]. For example, for depth 2, this gives the bound of $\Omega(n \log n)$. However, these bounds are very small, and we do not get anything at all for depth $O(\log n)$.

An even more restricted model for computing Shift^n that has been considered requires that the bits are routed along vertex-disjoint paths. A graph with n sources $\{x_0, \ldots, x_{n-1}\}$ and n sinks $\{y_0, \ldots, y_{n-1}\}$ is an *n-shifter* if for every s there exist n vertex disjoint paths $x_{0+s} \to y_0$, $x_{1+s} \to y_1$, $\ldots$, $x_{n-1+s} \to y_{n-1}$, where all indices are computed modulo n; see [23]. Obviously, any shifter is a semilinear circuit for Shift^n, but this condition seems to be much stronger. The lower bounds known for the shifters are much larger than for general semilinear circuits, e.g., shifters of depth 2 have size $\Omega(n^{3/2})$, and general shifters have size $\Omega(n \log n)$ (actually, these bounds are tight). These bounds can be applied to monotone circuits since any circuit with only monotone gates that computes the shift function is a shifter; see [22]. However, for the monotone basis, there are well-known even exponential lower bounds for other explicit functions. The reader interested in further applications of these methods to circuit complexity should consult [21, 24].

It is well known that a general circuit for the shift function is not necessarily a shifter [22]. It is an open problem whether the shift function can be computed more efficiently using general circuits than using circuits that contain shifter graphs. We give a partial result in this direction below by proving that certain circuits of depth 2 are more efficient than corresponding shifters; see Theorems 6.1 and 6.2.

2.3. Multiparty communication complexity. In the most common model of multiparty communication, a function $f(\vec{x}_1, \ldots, \vec{x}_k)$ is computed in the following way.

There are k players; we denote them by Player 1, ..., Player k. Player i knows $\vec{x}_1, \ldots, \vec{x}_{i-1}, \vec{x}_{i+1}, \ldots, \vec{x}_k$. They send messages consisting of binary strings, and the game ends when one of the players knows the answer $y = f(\vec{x}_1, \ldots, \vec{x}_k)$. The communication complexity is the minimal number of bits of communication needed in any protocol which computes $f(\vec{x}_1, \ldots, \vec{x}_k)$ correctly. For more background see, e.g., [19]. We omit the general definition of a protocol because in this paper we consider only some restricted models explained below. In particular, we require that each player sends only one message and that all players send their messages simultaneously. This means that the message can depend only on the input available to the player, not on the previous communication. A lower bound for such a restricted model should be easier to prove, while it would still imply the circuit lower bound.

Now we define our models more precisely. In the *simultaneous* model for computing $f(\vec{x}_1, \ldots, \vec{x}_k)$, there are $k+1$ players, Player 0, Player 1, ..., Player k. The additional Player 0 does not have access to any input, while the others have the same access as in the general model. There is only one round of communication, when each of the Players $1, \ldots, k$ sends a piece of information to Player 0. Then Player 0 produces the answer. A slight modification is an *almost simultaneous protocol*: there are k Players, and the communication has one round, in which Players $1, \ldots, k-1$ independently send a message to Player k, who produces the answer. The *communication complexity* is the total number of bits that has to be communicated for the inputs of given length. By $\mathrm{SCC}(f)$ and $\mathrm{ASCC}(f)$, we denote the simultaneous and almost simultaneous communication complexity of f, respectively. Let us observe that $\mathrm{ASCC}(f)$ differs from $\mathrm{SCC}(f)$ by at most the sum of the sizes of inputs $\vec{x}_1, \ldots, \vec{x}_{k-1}$ since Player k can just send all the inputs he has access to Player 0, which changes an almost simultaneous protocol into a simultaneous protocol. Therefore, if the size of the first inputs is small, the difference can be disregarded. This is true, for example, for shift_k^n, where the size of the first two inputs is only $\log n$.

A *semilinear protocol* is an almost simultaneous protocol such that for every fixed $\vec{x}_1, \ldots, \vec{x}_{k-1}$, the message sent by each Player i consists of a vector of fixed linear functions of $\vec{x}_k$. A *restricted protocol* is an almost simultaneous protocol for three players such that for any fixed $\vec{x}_1$, the message sent by Player 2 consists of a fixed substring of $\vec{x}_3$. A *restricted semilinear protocol* satisfies both conditions, i.e., Player 1 sends a vector determined by linear functions of $\vec{x}_3$ and Player 2 sends a substring of $\vec{x}_3$.

We study the communication complexity of the functions shift_k^n and perm_k^n.

Let us first examine our intuition about restricted protocols for $\mathit{shift}^n(i, j, \vec{x})$. If Player 2 does not send anything, the only knowledge of Player 3 about $\vec{x}$ comes from Player 1. Therefore, Player 1 has to send n bits about $\vec{x}$ because without knowledge of j he cannot determine which of the bits is relevant. (More precisely, for any two $\vec{x} \neq \vec{x}'$, the message sent by Player 1 has to be different.) Player 2 has to select a subset of bits without the knowledge of i; therefore, if he sends only a small number of bits, in most cases he does not send the bit x_{i+j}. It would appear then that his message can hardly be of any significant help in determining the output, and therefore Player 1 has to send a long message even in this case. A natural conjecture is that the total number of communicated bits for shift^n must be $\Omega(n)$. As we shall see, this would yield a proof that there is no circuit of size $O(n)$ and depth $O(\log n)$ computing Shift^n. However, we disprove this conjecture by showing that even for the more general function perm^n, there exists a restricted semilinear protocol with a total communication of only $O(n \log\log n / \log n)$ bits; see Corollary 4.8.

Clearly, the multiparty communication complexity of these functions can only decrease with increasing k in any of the above models. It is a major open problem to study the growth rate of the communication complexity when k increases. It seems beyond the present means to prove any good bounds for increasing k. For the function $shift_k^n$, we have a partial result in that direction. We construct a restricted semilinear protocol with a total communication of only $O(n(\log\log n/\log n)^k)$ bits for any constant k.

The best known lower bound for $shift_k^n$ is given in the following proposition. It is a straightforward generalization of the bound for $k = 1$ by Nisan and Wigderson [20].

PROPOSITION 2.3. *For every constant k,* $\mathrm{SCC}(shift_k^n) \geq \Omega(n^{\frac{1}{k+1}})$.

Proof. Assume that the numbers $s_1, \ldots, s_{k+1}$ are written as $(k+1)$-digit numbers in the basis of $b = \lceil n^{\frac{1}{k+1}} \rceil$. We restrict the domain of $shift_k^n(s_1, \ldots, s_{k+1}, \vec{x})$ so that in s_1 only the first digit may be nonzero, in s_2 only the second digit may be nonzero, etc. (Thus for $k = 1$, s_1 ranges over $0, \sqrt{n}, 2\sqrt{n}, \ldots, n$ and s_2 ranges over $0, 1, \ldots, \sqrt{n}$.) Each s_i can have only b different values, but $s_1 + \cdots + s_{k+1}$ can be any number between 0 and $b^{k+1} \geq n$.

Now consider any simultaneous protocol for $shift_k^n$, and for a fixed x, write down the information communicated by each player on all inputs $s_1, \ldots, s_{k+1}$ from our restricted range. The total amount of communication is at most $O(n \cdot \mathrm{SCC}(shift_k^n))$ bits. Because the protocol is simultaneous, the information communicated by each player depends only on the inputs available to him. For every input of a given player, there are b different values of the input that he does not see, and therefore the same output appears at b predetermined positions. Thus the information contained in all communication is only $O(n\mathrm{SCC}(shift_k^n)/b)$ bits. Given this information, it is possible to reconstruct the whole vector $\vec{x}$ because we can reconstruct any bit x_i by choosing inputs such that $s_1 + \cdots + s_k = i$ and following the protocol. Thus for any two different vectors $\vec{x}$, this information must be different, and hence it has at least n bits for some $\vec{x}$. This gives us $\mathrm{SCC}(shift_k^n) \geq \Omega(b) \geq \Omega(n^{\frac{1}{k+1}})$. □

The best known lower bound on multiparty communication complexity is $\Omega(n/c^k)$ for the generalized inner product [7]. This means that we have no lower bounds at all for $k = \Omega(\log n)$. If we could extend the lower bounds on the almost simultaneous communication complexity to $k = \mathrm{polylog}(n)$, it would yield a lower bound on ACC circuits, which is a major open problem; see [8, 15, 34].

One function often considered in this context is pointer jumping in a directed acyclic graph with one source and k additional levels with n vertices on each level. (The out-degree of each vertex is 1, except for the last level.) The inputs are divided between the players so that every player sees everything except for one level. Intuitively, the most difficult instances should be those in which all vertices in the graph are reachable, which means that on each level, the mapping given by the pointers is one to one. This restriction of pointer jumping is essentially a variation of our function $perm_k^n$. We define

$$jump_k^n(i, \pi_1, \ldots, \pi_k) = \pi_k \ldots \pi_1(i).$$

We can use our bound for the function $perm_k^n$, Corollary 4.8, to improve the obvious upper bound of $O(n \log n)$ on the communication complexity of $perm_k^n$ by a factor of $\log n/\log\log n$.

PROPOSITION 2.4. *For all $k \geq 2$,* $\mathrm{ASCC}(jump_k^n) \leq O(n \log\log n)$.

Proof. We can compute the $\lceil \log n \rceil$ bits of $jump_k^n(i, \pi_1, \ldots, \pi_k)$ by computing the value of $perm_{k-1}^n(i, \pi_1, \ldots, \pi_{k-1}, Perm^n(\pi_k, \vec{x}))$ for $\log n$ vectors $\vec{x}$ as follows. Define

vectors $\vec{x}^{(j)}$, $1 \le j \le \lfloor \log n \rfloor$, so that $\vec{x}_i^{(j)}$, $0 \le i < n$, is the jth digit in the binary representation of i. By the definitions of $Perm^n$, $perm_{k-1}^n$, and $jump_k^n$, the jth bit of $jump_k^n(i, \pi_1, \ldots, \pi_k)$ is $perm_{k-1}^n(i, \pi_1, \ldots, \pi_{k-1}, Perm^n(\pi_k, \vec{x}^{(j)}))$. We can run the $\lceil \log n \rceil$ protocols for $perm_{k-1}^n$ in parallel since they are completely independent; note that all players have the necessary information since the input is divided in the same way for $jump_k^n$ as for all the instances of $perm_{k-1}^n$. The total communication is $\lceil \log n \rceil \mathrm{ASCC}(perm_{k-1}^n) = O(n \log \log n)$ by Corollary 4.8. □

Our results do not exclude the function $jump_k^n$ as a candidate for lower bounds, but we think that they give some insight into the difficulties that are encountered in the attempted proofs.

2.4. Tensors. By a *tensor* over a field $\mathbf{F}$, we simply mean a three-dimensional matrix or, equivalently, a finite sequence of matrices of the same size with entries from $\mathbf{F}$. We use the following notation to denote the *slices* of tensors in different directions. For a tensor T, the symbol $T_{i,*,*}$ denotes the matrix consisting of all entries of T with the first coordinate i; the matrix is indexed in the same way as the remaining two coordinates of the tensor. Similarly, $T_{i,j,*}$ denotes the vector of all entries with the first coordinates i and j, i.e., the jth row of the matrix $T_{i,*,*}$. We define $T_{*,j,*}$, $T_{*,j,k}$, etc. similarly.

Tensors can be naturally associated with semilinear functions. Given a tensor T, for each parameter p, the matrix $T_{*,p,*}$ determines a linear function. Thus the function defined by $F(p, \vec{x}) = T_{*,p,*}\vec{x}$ is a semilinear function naturally corresponding to the tensor T. Conversely, let $F(p, \vec{x})$ be a function linear in $\vec{x}$. Then there exists a tensor T^F and a vector $\vec{c}^F$ such that for all p,

$$F(p, \vec{x}) = T^F_{*,p,*}\vec{x} + \vec{c}^F_p.$$

The constants $\vec{c}^F_p$ have almost no influence on the complexity of computing F. The main information is contained in the tensor T^F, which we call a *tensor corresponding to* F.

We study the tensors corresponding to the functions introduced in section 2.1. The tensor corresponding to $Shift^n$ is defined by $D^n = T^{Shift^n}$, i.e.,

$$\begin{aligned} D^n_{i,j,k} &= 1 \quad \text{if} \quad i + j \equiv k \pmod{n}, \\ &= 0 \quad \text{otherwise.} \end{aligned}$$

D^n is called a *diagonal tensor*, or the *tensor of multiplying polynomials* modulo $x^n - 1$ since it is connected with algebraic circuits computing this bilinear operation [12].

The tensor corresponding to $Perm^n$ is defined by $P^n = T^{Perm^n}$, i.e.,

$$\begin{aligned} P^n_{i,\pi,k} &= 1 \quad \text{if} \quad k = \pi(i), \\ &= 0 \quad \text{otherwise.} \end{aligned}$$

It turns out that the size of a circuit computing a semilinear function is related to some variants of the algebraic concept of rank.

Let $\mathbf{F}$ be some fixed field. For a positive integer k, let $e^k_0, \ldots, e^k_{k-1}$ denote the standard basis of the vector space $\mathbf{F}^k$, i.e., $e^k_i(j) = 0$ for $i \ne j$ and $e^k_i(i) = 1$. For $u \in \mathbf{F}^l$, $v \in \mathbf{F}^m$, and $w \in \mathbf{F}^n$, the *tensor product* of u, v, and w denoted by $u \otimes v \otimes w$ is the $l \times m \times n$ tensor T defined by $T_{i,j,k} = u_i v_j w_k$ for $0 \le i < l$, $0 \le j < m$, and $0 \le k < n$.

We define three different ranks of a tensor. In all three cases, the rank of T is defined to be the minimal number of rank-1 tensors T^i such that $T = \sum_i T^i$; it is 0 if

all entries of T are zeros. Note that the matrix rank can be defined in a similar way since matrices of rank 1 are the matrices of the form $u \otimes v$, where u and v are nonzero vectors.

The usual *tensor rank*, $\mathrm{rank}(T)$, was introduced by Strassen [31] and is determined by

$$\mathrm{rank}(T) = 1 \quad \text{iff } T = u \otimes v \otimes w, \text{ for some nonzero vectors } u, v, w.$$

The *contact rank*, $\mathrm{rank}^{2,2}(T)$, was introduced by Razborov [27] and is determined by

$$\mathrm{rank}^{2,2}(T) = 1 \quad \text{iff } T = e_i^l \otimes v \otimes w \quad \text{or} \quad T = u \otimes e_j^m \otimes q, \text{ for some nonzero vectors } u, v, w, q \text{ and some } i, j.$$

In other words, $\mathrm{rank}^{2,2}(T) = 1$ if all nonzero entries of T are either in the slice $T_{i,*,*}$ or in the slice $T_{*,j,*}$, for some i or j, and the matrix rank of that slice is 1.

We introduce the *rigidity rank* and denote it by $\mathrm{rank}^{2,1}(T)$; it is determined by

$$\mathrm{rank}^{2,1}(T) = 1 \quad \text{iff } T = u \otimes e_j^m \otimes w \quad \text{or} \quad T = e_i^l \otimes v \otimes e_k^n, \text{ for some nonzero vectors } u, v, w \text{ and some } i, j, k.$$

Thus $\mathrm{rank}^{2,1}(T) = 1$ if either

(i) all nonzero entries of T are in the slice $T_{*,j,*}$, for some j, and the matrix rank of $T_{*,j,*}$ is 1 or

(ii) all nonzero entries of T are in the column $T_{i,*,k}$, for some i and k, and at least one entry is not zero.

From these definitions, it is easy to see that for every tensor T,

$$\mathrm{rank}(T) \le \mathrm{rank}^{2,2}(T) \le \mathrm{rank}^{2,1}(T).$$

To give yet another equivalent definition of $\mathrm{rank}^{2,1}$, we define $\mathrm{diff}(T, U)$ to be the *set of different columns* for T and U, i.e.,

$$\mathrm{diff}(T, U) = \{(i, k); \exists j \; T_{i,j,k} \neq U_{i,j,k}\}.$$

Using the fact that the rank of a matrix M is the minimal number of matrices of rank 1 that add to M, it is easy to see that

$$\mathrm{rank}^{2,1}(T) = \min_U \left(\sum_j \mathrm{rank}(U_{*,j,*}) + |\mathrm{diff}(T, U)| \right), \tag{1}$$

where the minimum is taken over all tensors U of the appropriate dimensions.

Now we extend the concept of rigidity from matrices to tensors. The *rigidity of a matrix* M is the function $R_M(r)$ equal to the minimal number of changes needed to reduce the rank of M to r or less [32]. If we want to work with a set of matrices, we can represent them as slices $T_{*,j,*}$ of a tensor T. The *rigidity of a tensor* T is the function $R_T(r)$ which for each r gives the minimal number of columns in which we have to change the tensor in order to reduce the rank of each slice to r or less. More precisely,

$$R_T(r) = \min\{|\mathrm{diff}(T, U)|; U \text{ is a tensor such that } \forall j \; \mathrm{rank}(U_{*,j,*}) \le r\}. \tag{2}$$

Note that in the special case of a single matrix, we just get Valiant's rigidity.

It is known that the rank of the diagonal tensor D^n is $\Theta(n)$ for any field; for small fields, it was proved only in 1987 by Chudnovsky and Chudnovsky [17] using methods from algebraic geometry; see also [29] for a more detailed presentation. The growth rate of $\operatorname{rank}^{2,2}(D^n)$ and $\operatorname{rank}^{2,1}(D^n)$ is not known.

Razborov proved that $\operatorname{rank}^{2,2}(D^n) = \Omega(n^{3/2})$, which shows that the gap between rank and $\operatorname{rank}^{2,2}$ can be big [27]. He conjectured that $\operatorname{rank}^{2,2}(D^n) = \Omega(n^2)$.

We disprove this conjecture by proving an upper bound on $\operatorname{rank}^{2,1}(D^n)$. In fact, we prove a stronger result, an upper bound on the rigidity of P^n, the tensor of computing all permutations,

$$R_{P^n}(r) = n^{2-\Omega(r/n)}.$$

This implies an upper bound on the rigidity rank of the same tensor,

$$\operatorname{rank}^{2,1}(P^n) = O\left(n^2 \frac{\log\log n}{\log n}\right).$$

The diagonal tensor D^n is a subtensor of P^n; hence the same upper bounds hold for D^n. The rigidity rank is always greater than or equal to the contact rank, so we get the upper bound

$$\operatorname{rank}^{2,2}(D^n) = O\left(n^2 \frac{\log\log n}{\log n}\right).$$

Rigidity and rigidity rank are closely related. From equations (1) and (2), it follows easily that

$$\operatorname{rank}^{2,1}(T) \le R_T(r) + nr. \tag{3}$$

Using this fact, it is theoretically possible to get a lower bound on the rigidity of a tensor from a lower bound on the rigidity rank of it. However, the lower bounds needed for this approach to work are far beyond what we are able to prove nowadays.

In this paper, we prove a lower bound on the rigidity of D^n directly. We get

$$R_{D^n}(r) = \Omega\left(\frac{n^2}{r}\log\frac{n}{r}\right).$$

The same bound was proved even for a single matrix by Friedman [14]. However, because we prove our bound for a special set of matrices, it enables us to prove the following lower bound on the rigidity rank:

$$\operatorname{rank}^{2,1}(D^n) = \Omega\left(n^{3/2}(\log n)^{1/2}\right).$$

This bound in turn implies some improvement of the lower bounds on the size of certain circuits of depth 2.

3. Mutual relations. The concepts of computing a function by a graph with common bits, rigidity of the corresponding tensor, and communication complexity of the corresponding one-output function are closely related and can all be potentially useful for proving lower bounds on the size of circuits. We survey these relations in this section. Some parts of this material has been known and has appeared either explicitly or implicitly in the literature; see [19, 20, 32, 33]. We quantify the connections as

precisely as possible because our upper bounds show that this might be necessary for proving lower bounds.

We first demonstrate the connections on an example. We show a restricted semilinear protocol for the function $shift^n(i, j, \vec{x})$ in which Player 2 sends only 1 bit and Player 1 sends $\lfloor n/2 \rfloor$ bits. This protocol also demonstrates some of the ideas used in our more complicated upper bounds. According to the intuition discussed before Proposition 2.3, Player 2 almost never sends the bit that is to be computed. Before reading on, the reader can try to imagine how this single bit could possibly be used so that it saves half of the communication of the other player.

All indices in the protocol are computed modulo n. The protocol works as follows.

- Player 1 sends the bits $x_0 \oplus x_j$, $x_1 \oplus x_{j-1}$, ..., i.e., parities of all pairs $x_k \oplus x_{j-k}$.
- Player 2 sends the bit x_{n-i}.
- Player 3 computes x_{i+j} as the parity of x_{n-i} (from Player 2) and $x_{n-i} \oplus x_{i+j}$ (from Player 1).

The corresponding bound on computation by a graph with common bits is for the function $Shift^n(j, \vec{x})$. We construct a graph of degree 1 with $\lfloor n/2 \rfloor$ common bits that computes $Shift^n(j, \vec{x})$. Every y_i is adjacent to x_{n-i}, which corresponds to the bit sent by Player 2. For every j, the common bits are the same as the bits sent by Player 1 in the protocol. The output level computes x_{i+j} in the same way as Player 3 in the protocol.

The corresponding bound on the rigidity says that $R_{D^n}(\lfloor n/2 \rfloor) \leq n$ for the diagonal tensor D^n. To prove this, we flip all the entries $D_{i,j,n-i}$. It is easy to verify that the rank of each matrix $D_{*,j,*}$ is at most $\lfloor n/2 \rfloor$ because the matrix is essentially a matrix with ones only along the main diagonal and along the diagonal running from the lower left corner to the upper right corner (and zero in the intersection of the diagonals).

Now we state these relationships formally.

PROPOSITION 3.1. *A function $F(p, \vec{x})$ can be computed by a graph of maximal degree d with r common bits if and only if there exists a restricted protocol for computing $bit_F(i, p, \vec{x})$ in which Player 1 sends r bits and Player 2 sends d bits.*

The computation by the graph uses only semilinear functions if and only if the protocol is a restricted semilinear protocol.

Proof. Suppose that we have a graph that computes the function F. The protocol for f is as follows. Player 1 sends the value of the common bits for the given input $(p, \vec{x})$. If the first input is i, Player 2 sends the value of all inputs x_k that are adjacent to the output y_i in the graph. Player 3 can compute the output because he knows all of the inputs of the output gate. Clearly, this protocol is a restricted protocol with the required number of bits sent by each player.

Given a protocol, we construct a graph that computes F as follows. The output y_i is adjacent to all inputs x_k such that Player 2 sends the value of input x_k if the first input is i. This graph computes F if the common bits compute the values communicated by Player 1 on the given input. Clearly, the degree of the graph and the number of the common bits are as required.

For both directions, semilinearity is preserved since the functions computed in the protocol and used in the computation by the graph are identical. □

Computing F by a graph with a bounded number of edges and a bounded number of common bits using only semilinear functions is equivalent to a bound on the rigidity of the tensor T.

PROPOSITION 3.2. *Let $F(p,\vec{x})$ be a semilinear function. Then the rigidity of the corresponding tensor T^F satisfies $R_{T^F}(r) \le R$ if and only if there exists a graph G with at most R edges such that for every p, $F(p,\vec{x})$ can be computed by G with r common bits using only semilinear functions.*

Proof. First, assume that the condition on the rigidity is satisfied, which means that we have a tensor U such that $|\mathrm{diff}(T^F,U)| \le R$ and $\forall p\ \mathrm{rank}(U_{*,p,*}) \le r$. We choose the graph G so that the input x_k and the output y_i are adjacent if $(i,k) \in \mathrm{diff}(T^F,U)$. Obviously, there are at most R edges.

Now we show how this graph can compute F for fixed p with only r common bits. Let us denote the ith row of the matrix $U_{*,p,*}$ by $\vec{u}^{(i)}$. The ith output function is then given by

$$(F(p,\vec{x}))_i = T^F_{i,p,*} \cdot \vec{x} + c_i = \vec{u}^{(i)} \cdot \vec{x} + \sum_{(i,k)\in \mathrm{diff}(T^F,U)} a_{i,k}x_k + c_i,$$

where $a_{i,k}$ and c_i are some scalar constants from the given field. It follows that it suffices to set the common bits so that $\vec{u}^{(i)} \cdot \vec{x}$ can be computed from them for all i since the rest of the expression can be computed based on the inputs adjacent to the given output in G.

From the condition on the rank, it follows that there exist r vectors $\vec{v}^{(1)},\dots,\vec{v}^{(r)}$ such that any $\vec{u}^{(i)}$ is their linear combination. We set the common bits to $\vec{v}^{(1)} \cdot \vec{x},\dots,\vec{v}^{(r)} \cdot \vec{x}$; hence any $\vec{u}^{(i)} \cdot \vec{x}$ can be computed as their linear combination.

For the other direction, we assume that F can be computed by a graph G. We want to find a tensor U which proves that the rigidity of T^F is small. For a fixed p, we know that

$$(F(p,\vec{x}))_i = g_i(h_1(\vec{x}),\dots,h_r(\vec{x}),\vec{x}^{(i)}),$$

where $\vec{x}^{(i)}$ is the substring of input variables adjacent to y_i in G and both g_i and the common bits $h_1,\dots,h_r$ are linear functions. Therefore, the function g_i can be written as a linear combination of the common bits, the extra inputs, and a constant. Similarly to the other direction, we set $U_{i,p,*}$ to be a vector which corresponds to the linear combination of the common bits used for the given output. The rank of the tensor is bounded by the number of common bits, and it differs from T^F only in columns corresponding to the edges of G. □

The relation between the rigidity of tensors and the communication complexity of the corresponding semilinear function now follows easily. We just have to examine the proof of Proposition 3.1 and verify that if we replace the maximal degree of the graph by the number of edges, the bound on the number of bits communicated by Player 2 is replaced by the average number of bits.

PROPOSITION 3.3. *Let $F(p,\vec{x})$ be a semilinear function. Then the rigidity of the corresponding tensor T^F satisfies $R_{T^F}(r) \le dn$ if and only if there exists a restricted semilinear protocol for the associated one-output function $bit_F(i,p,\vec{x})$ in which Player 1 always sends r bits and Player 2 sends d bits on the average, where the average is taken over all values of i.*

Using Valiant's reduction, Theorem 2.2, we get the following two theorems.

THEOREM 3.4. *If $F(p,\vec{x})$ can be computed by a circuit of size $O(n)$ and depth $O(\log n)$, then for any $\varepsilon > 0$, there exists a constant K such that*

(i) *F can be computed by a graph G of degree at most n^ε and with $Kn/\log\log n$ common bits*;

(ii) *there exists a restricted protocol for* $bit_F(i,p,\vec{x})$ *such that Player* 1 *sends* $O(n/\log\log n)$ *bits and Player* 2 *sends* n^ε *bits.*

THEOREM 3.5. *If a semilinear function* $F(p,\vec{x})$ *can be computed by a semilinear circuit of size* $O(n)$ *and depth* $O(\log n)$*, then for any* $\varepsilon > 0$*, there exists a constant* K *such that*

(i) F *can be computed by a graph* G *of degree at most* n^ε *and with* $Kn/\log\log n$ *common bits using only semilinear functions*;

(ii) *there exists a restricted semilinear protocol for* $bit_F(i,p,\vec{x})$ *in which Player* 1 *sends* $O(n/\log\log n)$ *bits and Player* 2 *sends* n^ε *bits*;

(iii) *the rigidity of the corresponding tensor* T^F *satisfies* $R_{T^F}(Kn/\log\log n) \le n^{1+\varepsilon}$.

Originally, several researchers believed that a superlinear lower bound for circuits of depth $O(\log n)$ can be proved by showing that

(i) $Shift^n$ cannot be computed by a graph of degree $o(n)$ with $o(n)$ common bits,

(ii) $\mathrm{SCC}(shift^n) = \Omega(n)$, or

(iii) $\mathrm{rank}^{2,1}(D^n) = \Omega(n^2)$ (which would give bounds for semilinear circuits).

We prove that all these statements are false by exhibiting an almost simultaneous protocol for $shift^n$ in which both players send $O(n\log\log n/\log n)$ bits. This does not mean that these approaches cannot be used at all—to demonstrate that, we would need a protocol in which Player 2 sends only n^ε, which is less than our bound. However, it shows that it is not sufficient to consider the total communication—to prove a lower bound that way, it would be necessary to prove that more than $\omega(n/\log\log n)$ bits are needed, and our bound shows that this is not the case. Hence it is necessary to consider more precise information, namely to estimate the number of bits sent by each of the two players instead of the total amount of communication, to use rigidity instead of the rigidity rank, or to estimate the degree of the graph in a different way than the number of common bits.

4. Upper bounds. In section 4.1, we present the constructive proof of the upper bound on the communication complexity of the shift function $shift^n$. Then we generalize it in two different ways. In section 4.2, we extend the upper bound to the permutation function $perm^n$ using the probabilistic method. These bounds also apply to the rigidity of the corresponding functions. In section 4.3, we extend the upper bound to the function $shift_k^n$.

4.1. The shift function. The constructive proof of the upper bound for the function $shift^n$ is based on a suggestion of Wigderson to use arithmetic progressions.

The idea of the protocol is to divide the input $\vec{x}$ into groups. Player 1 sends the parity of each group and Player 2 sends some substring of $\vec{x}$ such that all but one elements of the group of x_{i+j} are sent, similarly as in the simple protocol at the beginning of section 3. For the function $shift^n$, we can do this constructively based on the next lemma, which says that there exists a sparse subset B of $[0,n-1]$ such that every $a \in [0,n-1]$ can be surrounded by an arithmetic progression with all elements in B except possibly a itself.

LEMMA 4.1. *For every* n *and every* $l < \log n/(2\log\log n)$*, there exists a set* $B \subseteq [0,n-1]$ *of size* $O(ln^{1-1/(2l-1)})$ *such that for every* $a \in [0,n-1]$*, there exist* $a_1,\dots,a_{2l-2} \in B$ *such that* $a_1,\dots,a_{l-1}$, a, $a_l,\dots,a_{2l-2}$ *is an arithmetic progression (computing modulo* n*) with modulus* $m = m(a)$ *bounded by* $m \le O(n^{1-1/(2l-1)})$.

Proof. From the well-known bounds on the distribution of primes, it is easy to prove that there exist $2l-2$ primes $p_1,\dots,p_{2l-2}$ larger than $\Omega(n^{1/(2l-1)})$ whose

product is at most $n^{1-1/(2l-1)} < n/l$. Define B to be the set of all integer multiples of these primes between $-n$ and $2n$, taken modulo n, i.e.,

$$B = \{b \bmod n; b \in [-n, 2n] \wedge (p_1|b \vee p_2|b \vee \cdots \vee p_{2l-2}|b)\}.$$

For a given $a \in [0, n-1]$, consider the following system of linear congruences:

$$\begin{aligned}
a - (l-1)m &\equiv 0 \pmod{p_1}\\
a - (l-2)m &\equiv 0 \pmod{p_2}\\
&\vdots\\
a - m &\equiv 0 \pmod{p_{l-1}}\\
a + m &\equiv 0 \pmod{p_l}\\
a + 2m &\equiv 0 \pmod{p_{l+1}}\\
&\vdots\\
a + (l-1)m &\equiv 0 \pmod{p_{2l-2}}.
\end{aligned}$$

Since $0 \le l < p_1, \ldots, p_{2l-2}$, we can divide the congruences by $l-1$, $l-2, \ldots, 1$, 1, 2, $\ldots$, $l-1$ and apply the Chinese remainder theorem to solve for m. We obtain a solution $0 < m \le p_1 p_2 \cdots p_{2l-2}$, which is a modulus satisfying the requirements, because all the numbers $a_1 = a - (l-1)m$, $a_2 = a - (l-2)m, \ldots, a_{l-1} = a - m$, $a_l = a + m$, $\ldots$, $a_{2l-2} = a + (l-1)m$ taken modulo n are in B. □

THEOREM 4.2. *There exists a restricted semilinear protocol for the shift function $shift^n$ which requires only $O(n \log\log n/\log n)$ bits of communication.*

Proof. Let B be a set with the properties as in Lemma 4.1 for an l which we choose later. For a set $C \subseteq [0, n-1]$, let $x[C]$ denote the subsequence of the bits of x indexed by C. All indices of x are taken modulo n. The protocol is as follows.

- Player 1 partitions the interval $[0, n-1]$ into $n' = O(n/l)$ arithmetic progressions $C_0, \ldots, C_{n'-1}$ of length at most l with modulus $m(j)$. Player 1 communicates $\bigoplus x[C_\alpha]$ for all $\alpha = 0, \ldots, l$, i.e., parity of bits of x indexed by each of the progressions, a total of $O(n/l)$ bits.
- Player 2 sends $x[B + i]$, i.e., bits x_{b+i} for all $b \in B$.
- Player 3 computes x_{i+j} as described below.

Let C be the C_α containing $i + j$, and let $c \in C$, $c \ne i + j$. By the definition of C, $c - i - j$ is a small nonzero multiple of $m(j)$. By the definition of B and $m(j)$, it follows that $j + (c - i - j) = c - i \in B$, and hence $c \in B + i$. This means that Player 2 communicated all bits of $x[C]$ except possibly x_{i+j}. Hence Player 3 can compute x_{i+j} by subtracting these bits sent by Player 2 from $\bigoplus x[C]$ sent by Player 1, using the fact that he knows i and j and therefore knows the meaning of the bits sent by Players 1 and 2.

For $l = \log n/(4 \log\log n)$, the size of B is $O(n/\log n) = O(n \log\log n/\log n)$, and each player communicates $O(n \log\log n/\log n)$ bits. □

COROLLARY 4.3. $\operatorname{rank}^{2,1}(D^n) = O(n^2 \log\log n/\log n)$.

Proof. By Proposition 3.3, we get $R_{D^n}(n \log\log n/\log n) = O(n^2 \log\log n/\log n)$; the result follows using the relation of the rigidity and the rigidity rank (3). □

4.2. The permutation function. We prove an upper bound on the rigidity of the tensor $Perm^n$ using the following bound on the chromatic number χ of a random graph $\mathbf{G}(n, q)$, which is an undirected random graph on n vertices with each edge chosen independently at random with probability q.

THEOREM 4.4. *For every $\varepsilon > 0$, there exists $\delta > 0$ and n_0 such that*

$$\Pr\left[\chi(\mathbf{G}(n,q)) \le \left(\frac{1}{2}+\varepsilon\right)\frac{-n\log(1-q)}{\log n}\right] > 1-\exp\left(-n^{1+\delta}\right),$$

as long as $7/8 < q < 1 - 1/n^{\frac{1}{2}-\varepsilon}$ and $n \ge n_0$.

This is essentially the theorem which was proved by Bollobás for every *constant* q [9]. However, for our application, we are interested in cases when q approaches 1 as n increases, which have not been considered in the literature. Therefore, we give the proof of Theorem 4.4 in section 7.

Let $\mathbf{S}(n,p)$ be the sum of independent Bernoulli random variables with mean p.

LEMMA 4.5. *Let $n, r, d, 0 < p < 1$ be numbers such that*

$$n! \cdot \Pr\left[\chi(\mathbf{G}(n,1-p^2)) > r\right] + n \cdot \Pr\left[\mathbf{S}(n,p) > d\right] < 1. \tag{4}$$

Then $perm^n$ can be computed by a restricted semilinear protocol where Player 1 sends r bits and Player 2 sends d bits.

Proof. Let $\{x_0,\dots,x_{n-1}\}$ and $\{y_0,\dots,y_{n-1}\}$ be disjoint sets of distinct vertices. Take a random bipartite graph $\mathbf{H}$ with each edge (x_k, y_i) chosen independently with probability p. Let π be a permutation on $\{0,\dots,n-1\}$. Let $\mathbf{G}_\pi$ be an undirected graph on $\{0,\dots,n-1\}$ such that for $i \ne j$, (i,j) is and edge of $\mathbf{G}_\pi$ if both edges $(x_i, y_{\pi^{-1}(j)})$ and $(x_j, y_{\pi^{-1}(i)})$ are in $\mathbf{H}$. Then $\mathbf{G}_\pi$ is a random graph with the same distribution as $\mathbf{G}(n,p^2)$, and its complement $\overline{\mathbf{G}}_\pi$ is a random graph from $\mathbf{G}(n,1-p^2)$. By assumption (4), there exists a graph H such that the degree of H is at most d and for each π, $\chi(\overline{\mathbf{G}}_\pi) \le r$. This means that for each π, the graph $\mathbf{G}_\pi$ can be covered by at most r cliques.

Using such a graph H, we construct a protocol for $perm^n(i,\pi,\vec{x})$. (In fact, this protocol demonstrates that $Perm(\pi,\vec{x})$ can be computed by H with r common bits.) For a fixed π, let $C_1,\dots,C_r$ be the cliques covering $\mathbf{G}_\pi$.

- Player 1 sends the vector $(\bigoplus x[C_1],\dots,\bigoplus x[C_r])$ (for notation, see Theorem 4.2).
- Player 2 sends all inputs x_k adjacent to y_i in H.
- Player 3 takes the clique C_s containing $\pi(i)$. Since for every $k \in C_s - \{\pi(i)\}$ the edge $(k,\pi(i))$ is in $\mathbf{G}_\pi$, it follows that x_k is adjacent to $y_{\pi^{-1}\pi(i)} = y_i$ in H. Thus the message of Player 2 contains the bit x_k for every $t \in C_s - \{\pi(i)\}$. Hence Player 3 can compute $x_{\pi(i)}$ by subtracting these bits from $\bigoplus x[C_s]$ sent by Player 1. ∎

THEOREM 4.6. *For every $\varepsilon > 0$ and $1/n^{\frac{1}{4}-\varepsilon} < p < 1/5$, there exists n_0 such that for every $n \ge n_0$, $perm^n$ can be computed by a restricted semilinear protocol where Player 1 sends $(1+\varepsilon)(-n\log p)/\log n$ bits and Player 2 sends $(1+\varepsilon)pn$ bits.*

Proof. Let ε and p satisfying the condition above be given, and let n be sufficiently large. In order to apply Lemma 4.5, we only need to estimate the probabilities for the chromatic number and independent Bernoulli variables. The first one is proved by Theorem 4.4,

$$n! \cdot \Pr\left[\chi(\mathbf{G}(n,1-p^2)) > (1+\varepsilon)\frac{-n\log p}{\log n}\right] < \exp(n\log n - n^{1+\delta}) = o(1).$$

The second one is a direct consequence of Chernoff–Hoeffding bounds,

$$n \cdot \Pr\left[\mathbf{S}(n,p) > (1+\varepsilon)pn\right] \le n \cdot 2\exp(-\Omega(pn)) = o(1). \qquad ∎$$

COROLLARY 4.7. *There exist constants $c, \delta > 0$ such that for $cn/\log n < r < \delta n$,*

$$R_{P^n}(r) \leq n^{2-(1-o(1))\frac{r}{n}}.$$

Proof. The result follows from Theorem 4.6 and Proposition 3.3 by a computation. □

COROLLARY 4.8.

(i) *There exists a restricted semilinear protocol for $perm^n$ which uses at most $O(n \log\log n/\log n)$ bits of communication. Thus for every k, $\mathrm{ASCC}(perm_k^n) = O(n \log\log n/\log n)$.*

(ii) $\mathrm{rank}^{2,1}(R^n) = O(n^2 \log\log n/\log n)$.

Proof. (i) Apply Theorem 4.6 with $p = \log\log n/\log n$. For larger k, the communication complexity can only be smaller.

(ii) This follows from (i) using Proposition 3.3. □

4.3. The iterated shift function. In this section, we generalize the upper bound from section 4.1 to the iterated shift function. This generalization is possible because the protocol for the function *shift* is given explicitly. We are not able to generalize the nonconstructive protocol of the previous section; thus we are not able to prove a bound on $\mathrm{ASCC}(perm_k^n)$ which decreases with increasing k.

THEOREM 4.9. *There exists a semilinear protocol for the k-times iterated shift function $shift_k^n(s_1, \ldots, s_{k+1}, \vec{x})$ such that if k is an arbitrary constant, each player sends at most $O(n(\log\log n/\log n)^k)$ bits, and if $k \geq c \log n$, for some constant c, each player sends at most $O(n^{6/7})$ bits.*

Proof. Remember that in the construction for $k = 1$, Player 1 divides the input x into several groups and communicates the parity of all of them. However, only one of these bits is really used by the last player. In our generalized construction, we compute this one bit recursively, using the first k players. It turns out that if we are careful, this is very similar to computing the function $shift_{k-1}^n$.

The length l of the arithmetic sequences we use will be chosen later to balance the number of bits communicated by individual players. Let $B \subseteq [0, n-1]$ be the set constructed in Lemma 4.1.

Player $k+1$ acts as Player 2 in the construction for $k = 1$, he sends $x[B + s_1 + \cdots + s_k]$, i.e., bits $x_{b+s_1+\cdots+s_k}$ for all $b \in B$. (All indices of x are taken modulo n.)

Players 1 to k all know the input s_{k+1}; hence they can compute the modulus $m = m(s_{k+1})$ as in Lemma 4.1. First they pad the input x by zeros so that its length is the smallest $n' \geq n$ divisible by ml. Then they divide the interval $[0, n'-1]$ into sequences $C_0, \ldots, C_{n''-1}$ of length l and modulus m as follows. Sequence C_0 starts with 0, C_1 starts with 1, ..., C_{m-1} starts with $m-1$ (this covers the subinterval $[0, ml-1]$), C_m starts with x_{ml}, and so on.

Let $y_j = \bigoplus x[C_j]$ be the parity of the bits of x indexed by the jth sequence. Let $f(j)$ be a number such that $j \in C_{f(j)}$. The goal of Players 1 to k is to compute $y_{f((s_1+\cdots+s_k) \bmod n')}$. If it were true that $f((s_1 + \cdots + s_{k+1}) \bmod n') = (f(s_1) + \cdots + f(s_k) + t) \bmod n''$ for some constant t, we would just use the protocol for $shift_{k-1}^n(f(s_1), \ldots, f(s_k), y')$, where y' is y shifted by t. This is not exactly the case, but we show that from the point of view of any player, there are only constantly many possible values of t.

Every player can compute t under the assumption that the input he does not see is 0. This value differs from the correct value by $f(r+s) - f(s) - f(r)$, where s is the input he does not see and r is the sum of all other inputs s_i (taken modulo n'). We now show that the only possible values of this difference are $-m$, 0, or m. Let the

numbers $a, a', a'' < m$, and $b, b', b'' < l$, and c, c', c'' be such that $s = cml + bm + a$, $r = cml + bm + a$, and $r + s = cml + bm + a$. Then $f(s) = cm + a$, $f(r) = c'm + a'$, and $f(r + s) = c''m + a''$. The claim is proved by observing that the value of a'' is either $a + a'$ or $a + a' - m$ and the value of c'' is either $c + c'$ or $c + c' + 1$. This works even for $r + s \geq n'$ since we have chosen n' divisible by ml.[1]

If each player communicates information according to the protocol for all three possible values of t, he certainly communicates the information for the correct t. Player $k + 2$ knows all $s_1, \ldots, s_{k+1}$; hence he can determine which t is correct and recover the result. Here we use the fact that the protocol is almost simultaneous; hence a player can follow a protocol even if other players do not, as he is not dependent on them.

It is obvious that the iterated protocol is semilinear. It remains to compute the amount of communication. Let $F(n, k)$ denote the maximal number of bits sent by an individual player in our protocol for $shift_k^n$. As a base case, we know by Theorem 4.2 that $F(n, 1) = O(n \log \log n / \log n)$. From the previous analysis, we know that Players 1 to k send at most $3F(n'', k-1)$ bits. As long as Player $k+1$ does not send more bits, we have $F(n, k) \leq 3F(n'', k - 1)$.

If k is constant, we choose $l = \log n/((k + 2) \log \log n)$ for each level of recursion. Since $n'' \leq 2n/l$, the recurrence gives $F(n, k) = O(n(\log \log n / \log n)^k)$. By Lemma 4.1, the size of B is $O(n(\log n)^{-k})$; hence the number of bits communicated by the last player is small, and our use of the recurrence is correct.

If k is not constant, we choose $l = 4$. In this case, the bound on the modulus is $o(n)$ and hence $n'' = (1 + o(1))n/l = n(3/4 + o(1))$. By induction, we get $F(n, k) \leq n(3/4 + o(1))^k$. By Lemma 4.1, the size of B is $O(n^{6/7})$. Therefore, we can iterate only as long as $cn^{6/7} \leq n(3/4 + o(1))^k$, i.e., up to some $k = \Theta(\log n)$. At that point, $F(n, k) = O(n^{6/7})$. □

5. Lower bounds. In this section, we prove a trade-off between the size of graphs and common bits which are needed to compute the shift function. This implies lower bounds on the rigidity function and the rigidity rank of the tensor D^n. As corollaries, we obtain the best known lower bounds for depth-2 circuits computing the shift function and multiplication. The same technique also shows that the protocols based on disjoint parities used in our upper bound cannot be extended so that Player 2 sends less than $n^{1/3}$ bits.

Let H^n be defined as $H^n = \{(x_i, y_j); 0 \leq j < i < n\}$. We first show that this graph cannot compute identity very well.

LEMMA 5.1. *Suppose that the graph H^n computes the identity function with r common bits, either in the Boolean case or using only linear functions over an arbitrary field. Then $r \geq n$.*

Proof. In the linear case, the computed function is represented by a matrix which is a sum of a matrix of rank r (the part with the common bits) and an upper triangular matrix with zeros on the diagonal (the edges). Such a matrix can be a diagonal matrix only if $r \geq n$.

In the Boolean case, we proceed by induction. The basis $n = 1$ is trivial. For the step from n to $n+1$, let $z_1, \ldots, z_r$ be the common bits for H^{n+1}. Since y_n is isolated in H^{n+1}, $y_n = x_n = g(z_1, \ldots, z_r)$ for some function g. Let $a \in \{0, 1\}$ be such that there

[1] It might be clearer to look at special examples. If $s = ml$, then $f(s) = m$ and it is easy to check that $f(r+s) = f(r)+m = f(r)+f(s)$ for all r. If $s = m$, then $f(s) = 0$ and either $f(r+s) = f(r)$ or $f(r+s) = f(r)+m$ (if $r+s$ is a multiple of ml). If $s = 1$, then $f(s) = 1$ and either $f(r+s) = f(r)+1$ or $f(r + s) = f(r) + 1 - m$ (if $r + s$ is a multiple of m but not a multiple of ml).

are at most 2^{r-1} elements $\vec{z} \in \{0,1\}^r$ with $g(\vec{z}) = a$. The set $\{\vec{z} \in \{0,1\}^r;\ g(\vec{z}) = a\}$ can be represented as a subset of $\{0,1\}^{r-1}$, and thus we can compute Id^n with these $r-1$ common bits with the graph H^n. By the inductive assumption, $r-1 \geq n$; hence $r \geq n+1$. □

THEOREM 5.2. *For every $\alpha > 0$, there exists $\delta > 0$ such that for every n and $r \geq n^\alpha$, if $Shift^n(s, \vec{x})$ can be computed by a graph G with r common bits for any $n/2$ values of $s \in \{0, \ldots, n-1\}$, either in the Boolean case or using only semilinear functions over an arbitrary field, then the size of G satisfies*

$$|G| \geq \delta \frac{n^2}{r} \log \frac{n}{r}. \tag{5}$$

Proof. Suppose that a graph $G \subseteq \{x_0, \ldots, x_{n-1}\} \times \{y_0, \ldots, y_{n-1}\}$ computes $n/2$ shifts with r common bits. Let $d = 4|G|/n$, and let $A = \{i; \text{degree}(x_i) \leq d\}$ and $B = \{i; \text{degree}(y_i) \leq d\}$ be the sets of vertices with small degrees in G. Clearly $|A|, |B| \geq 3n/4$ since d is four times the average degree of G.

Now consider a particular shift s computed by G. Let $A' = A \cap (B + s)$, where $B + s = \{(i + s) \bmod n; i \in B\}$. Clearly $|A'| \geq n/2$. Let G_s be a graph with vertices A' such that (a, b) is an edge if $(x_{a+s}, y_b) \in G$ and $a \leq b$. Let S be the set of all shifts s such that G_s has at most $n/4$ loops (edges of type (a, a)). If $|S| < n/4$, there are at least $n^2/16$ loops in all graphs G_s, and (5) is satisfied since each edge of G corresponds to at most one loop. Hence we assume that $|S| \geq n/4$ and restrict ourselves to shifts $s \in S$.

Suppose that for some $s \in S$ the graph G_s has less than $n/24$ triangles. Then there is a set $A'' \subseteq A'$, $|A''| \geq n/8$, which induces a triangle-free graph with no loops. Ajtai, Komlós, and Szemerédi [1] proved that a triangle-free graph of degree at most d contains an independent set of size at least

$$t \geq \frac{|A''| \log d}{100d} = \frac{n \log d}{800d}.$$

If there is an independent set K in G_s, it follows that $a > b$ whenever $(x_{a+s}, y_b) \in G$ for $a, b \in K$. Hence the graph G restricted to the nodes $\{x_{a+s}; a \in K\}$ and $\{y_b; b \in K\}$ is essentially a subgraph of $H^{|K|}$, and it computes the identity function, assuming G computes the shift s correctly. By Lemma 5.1, the number of common bits r is at least $|K| \geq t$. It follows that

$$d = \Omega\left(\frac{n}{r} \log \frac{n}{r}\right)$$

and (5) is satisfied.

The remaining case is that each of the $n/4$ graphs G_s, $s \in S$, contains at least $n/24$ triangles. Let $c < a < b$ be vertices of a triangle in G_s. From the definition of G_s, it follows that $(x_{a+s}, y_b, x_{c+s}, y_a)$ is a path in G. Such a path determines s (as the difference of the indices of the first and the last nodes); hence there are at least $|S|n/24 \geq n^2/96$ such paths. However, since the degree of nodes in G indexed by A and B is bounded by d, the number of paths is at most nd^3. This gives $nd^3 \geq n^2/96$; hence $d = \Omega(n^{1/3})$, $|G| = \Omega(n^{4/3})$, and (5) is satisfied if $r \geq n^{2/3} \log n$.

Now suppose $n^\alpha < r < n^{2/3} \log n$. We reduce this case to the previous part of the proof. Take an ε such that

$$n^{\frac{2}{3}\varepsilon} \log n^\varepsilon < r < n^{\frac{5}{6}\varepsilon}.$$

Thus $6x/5 < \varepsilon < 1$. Divide each of the sets $\{x_0, \ldots, x_{n-1}\}$ and $\{y_0, \ldots, y_{n-1}\}$ into $n^{1-\varepsilon}$ disjoint intervals of size n^ε. Let $U \subseteq \{x_0, \ldots, x_{n-1}\}$ and $V \subseteq \{y_0, \ldots, y_{n-1}\}$ be two such intervals. Then $G \cap (U \times V)$ realizes $n^\varepsilon/2$ noncyclic shifts with r common bits. A simple modification of this graph ("wrapping around") gives a graph with the same number of edges which computes $n^\varepsilon/2$ cyclic shifts with r common bits. Thus for each of the $n^{2-2\varepsilon}$ disjoint sections of G, we have a lower bound $\Omega((n^{2\varepsilon}/r)\log(n^\varepsilon/r))$ and thus

$$|G| = n^{2-2\varepsilon}\,\Omega\left(\frac{n^{2\varepsilon}}{r}\log\frac{n^\varepsilon}{r}\right) = \Omega\left(\varepsilon\frac{n^2}{r}\log\frac{n^\varepsilon}{r}\right) = \Omega\left(\frac{n^2}{r}\log\frac{n}{r}\right)$$

because ε is bounded by the constant α. □

COROLLARY 5.3.

(i) *Let $\alpha > 0$ be fixed. Then $R_{D^n}(r) = \Omega((n^2/r)\log(n/r))$, for $r \geq n^\alpha$.*

(ii) *Let $\alpha > 0$ be fixed. In any restricted protocol that computes $shift^n$ in which Player 1 sends at most r bits, $r \geq n^\alpha$, Player 2 sends at least $\Omega((n/r)\log(n/r))$ bits. Thus the total communication is at least $\Omega(\sqrt{n\log n})$ bits.*

(iii) $\operatorname{rank}^{2,1}(D^n) = \Omega\left(n^{3/2}(\log n)^{1/2}\right)$.

Proof. (i) and (ii) follow from Theorem 5.2 using Propositions 3.2 and 3.1.

(iii) Let $r = n^{1/2}(\log n)^{1/2}$. Let a decomposition of D^n into tensors of the form $u \otimes e_j^n \otimes w$ and $e_i^n \otimes v \otimes e_k^n$ be given. Let S be the sum of the tensors of the first type. If $\operatorname{rank}(S_{*,j,*}) \geq r$ for at least $n/2$ of the slices, we are done, so suppose the converse. Using Theorem 5.2 for the set of shifts j such that $\operatorname{rank}(S_{*,j,*}) < r$ and the transformation from Proposition 3.2, it follows that the number of tensors of type $e_i^n \otimes v \otimes e_k^n$ is at least

$$\Omega\left(\frac{n^2}{r}\log\frac{n}{r}\right) = \Omega\left(n^{3/2}(\log n)^{1/2}\right). \qquad \square$$

COROLLARY 5.4.

(i) *For any field $\mathbf{F}$, any depth-2 semilinear circuit for $Shift^n$ has size at least $\Omega\left(n(\log n)^{3/2}\right)$.*

(ii) *Any depth-2 circuit with arbitrary Boolean functions as gates which computes $Shift^n$ has size at least $\Omega\left(n(\log n)^{3/2}\right)$.*

Proof. In both cases, let a circuit for $Shift^n$ be given. Fix $\alpha > 0$. Let $d_1 \geq d_2 \geq \cdots$ be the degrees of the vertices on the middle level. For an arbitrary $r \geq n^\alpha$, we construct a graph with r common bits that computes $Shift^n$ as follows. We take the common bits to be the functions computed at r vertices with maximal degree. The graph is constructed by connecting an input to an output if they are connected by a path going through one of remaining vertices. This graph has at most $\sum_{j>r} d_j^2$ edges. By Theorem 5.2, it follows that

$$\sum_{j>r} d_j^2 \geq \Omega\left(\frac{n^2}{r}\log\frac{n}{r}\right).$$

This bound (for $n^\alpha \leq r \leq n$) implies

$$\sum_{j\geq 1} d_j = \Omega\left(n(\log n)^{3/2}\right).$$

See [24, Lemma 4] for a proof of this implication. □

COROLLARY 5.5.

(i) *Every depth-2 algebraic circuit for multiplying two polynomials has size* $\Omega\left(n(\log n)^{3/2}\right)$.

(ii) *Every depth-2 Boolean circuit for multiplying two n-bit numbers has size* $\Omega\left(n(\log n)^{3/2}\right)$.

Proof. Reduce the function $Shift^n$ to these functions. □

These are the best lower bounds for these functions for depth 2. Also, the bound $\Omega\left(n(\log n)^{3/2}\right)$ is the asymptotically largest lower bound for any explicitly given Boolean function. Previously, such a bound for general depth-2 circuits was known only for functions which contain a superconcentrator [2]. For linear circuits, such a bound can also be proved using the bound on rigidity of the parity-check matrix of a good code; see [14, 24].

The technique of counting triangles from Theorem 5.2 can be used to prove a stronger result for a more restricted model of computation by graphs.

THEOREM 5.6. *Suppose that* $Shift^n(s,\vec{x})$ *can be computed by a graph G with* $n/3$ *common bits using only semilinear functions, with an additional condition that for every s, the common bits compute parities of* pairwise-disjoint *subsets of bits of* $\vec{x}$. *Then the degree of G is at least* $\Omega(n^{1/3})$.

Proof. Suppose that the degree of G is $d = o(n^{1/3})$. Similarly as in Theorem 5.2, we consider the graph G_s for each shift. By counting the edges, for most values of s, the graph G_s has at most $o(n^{1/3})$ loops. This means that for such a value of s, the only way how to compute the value of other outputs is to use one of the common bits and the other input bits used by that common bit. Since each input is used only by a single common bit, the corresponding vertices in G_s have to induce a complete graph (more precisely, a tournament). Because there are only $n/3$ common bits which have to use all of $n - o(n^{1/3})$ input bits not in loops, these complete graphs contain $\Omega(n)$ triangles for each s, a total of $\Omega(n^2)$ triangles. By the same argument as in Theorem 5.2, the total number of triangles is at most $nd^3 = o(n^2)$, which is a contradiction. □

Similarly as in Theorem 5.2, we can extend this result to all graphs computing some constant fraction of shifts and prove that the number of edges has to be at least $n^{4/3}$, which is slightly stronger.

If we could prove a similar result without the additional restriction on the computation, it would follow that there are no linear circuits of logarithmic depth for the function $Shift^n$. Or, conversely, this result says that the protocols from section 4, which are all based on disjoint parities, are not powerful enough to break the bound of Theorem 3.5.

6. Conservative and nonconservative computation. Now we show that a conservative model of computations of shifts, which is based on sending information along vertex disjoint paths, is less efficient than a Boolean circuit (with parity gates) or a linear circuit over any field. We show this for circuits of depth 2 when the complexity is measured as the number of edges, with a modification that we do not count edges incident to a small set of vertices (of size $o(n)$). In fact, we show a stronger result, namely that in this way we can compute even all permutations more efficiently than one can compute only shifts using shifters. Ideally, we would like to prove the result when counting all edges; however, we are not able to do so at present time.

THEOREM 6.1. *There exists a semilinear circuit of depth* 2 *for* $Perm^n$ *and a set X of vertices on the middle level such that* $|X| = o(n)$ *and there are only* $o(n^{3/2})$ *edges disjoint with X.*

Before we prove Theorem 6.1, we prove the complementary lower bound for shifters, which is quite simple.

THEOREM 6.2. *If G is an n-shifter of depth 2, then G has $\Omega(n^{3/2})$ edges, even if we remove any $o(n)$ vertices and the edges incident with them.*

Proof. Suppose G is an n-shifter of depth 2. Let A be an arbitrary set of vertices of size $o(n)$. Suppose that there are only $o(n^{3/2})$ edges of G which are not incident with A. Let B be the set of vertices incident with $n^{1/2}$ such edges. By the assumption, $|B| = o(n)$. Since G is an n-shifter, there are $n - |A \cup B|$ paths from inputs to outputs disjoint with $A \cup B$ for each shift, a total of $n^2(1 - o(1))$ paths disjoint with $A \cup B$. However, each edge which is not incident with $A \cup B$ can belong only to $n^{1/2}$ paths; hence there are $n^{3/2}(1 - o(1))$ edges not incident with $A \cup B$, a contradiction. □

Bounds on the size of bounded-depth shifters have been proved in [23], including a lower bound $\Omega(n^{3/2})$ for depth 2. Let us also mention a related unpublished result of Maass which gives a bound $\Omega(n^{3/2})$ on the size of the circuits of depth 2 that compute $Shift^n$, with the restriction that there is a constant-size set of Boolean functions assigned to each vertex and for every shift we can only use as a gate assigned to this vertex either one of these functions or an arbitrary projection.

The proof of Theorem 6.1 is again probabilistic. A constructive proof for $Shift^n$ also seems possible. The basic idea is the same as in the upper bounds of section 4. Instead of sending the bits directly, we partition the inputs into $o(n)$ blocks, send the parities of the blocks through the extra $o(n)$ vertices, and then compute the individual bits from the parities of the blocks using direct connections realized by vertex-disjoint paths. The difficult part is to realize these direct connections with only $o(n^{3/2})$ edges. We prove that a random graph with suitable parameters satisfies this condition. We show that for each permutation, it is possible to choose one block of inputs with the necessary vertex-disjoint paths with a very large probability. This enables us to choose the blocks one by one until only $o(n)$ inputs remain. The values of these remaining inputs are also sent using the extra $o(n)$ vertices.

We need to make some preliminary considerations before we begin the proof. For the rest of this section, a graph means a graph of a circuit of depth 2, formally a quadruple (V_1, V_2, V_3, E) such that V_1, V_2, and V_3 are disjoint sets of vertices and the edges are $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_3)$. An *embedding* of (V_1, V_2, V_3, E) into (V_1', V_2', V_3', E') is a one-to-one mapping g that maps vertices to vertices in the corresponding set and edges to edges. For technical reasons, we assume that the sets V_1 and V_1' are ordered and that any embedding has to preserve this ordering as well, i.e., if $V_1 = \{x_1, \ldots, x_n\}$, $V_1' = \{x_1', \ldots, x_{n'}'\}$, and $g(x_1) = x_{i_1}', \ldots, g(x_n) = x_{i_n}'$, then $i_1 < i_2 < \cdots < i_n$.

Let $G_k = (U_1, U_2, U_3, E)$ be the graph with k inputs and k outputs which realizes by vertex-disjoint paths all connections between all pairs of inputs and outputs, except for the corresponding pairs, i.e.,

$$\begin{aligned} U_1 &= \{u_1, \ldots, u_k\}, \\ U_2 &= \{w_{i,j};\ 1 \le i, j \le k, i \ne j\}, \\ U_3 &= \{v_1, \ldots, v_k\} \\ E &= \{(u_i, w_{i,j}), (w_{i,j}, v_j);\ \text{for } w_{i,j} \in U_2\}. \end{aligned}$$

The proof of the theorem is based on the following lemma, which we prove later.

LEMMA 6.3. *Let $k > 2$ be an arbitrary constant. Then for sufficiently large n, there exists a graph $G = (V_1, V_2, V_3, E)$, $|V_1| = |V_3| = n$, $|V_2| = nk^2$, with $n^{3/2}/k$ edges such that for every m, $n/k \le m \le n$, every $W_1 \subseteq V_1$, $W_2 \subseteq V_2$, $W_3 \subseteq V_3$, $|W_1| = |W_3| = m$, $|W_2| = mk^2$, and every bijection $f : W_3 \to W_1$, there exists an*

embedding g of G_k into subgraph of G induced by $W_1 \cup W_2 \cup W_3$ such that if $g(v_j) = x$, then $g(u_j) = f(x)$.

Proof of Theorem 6.1. For given n, pick the largest k such that the condition of Lemma 6.3 holds. Because k was an arbitrary constant, we have $k = \omega(1)$. Take the graph G from Lemma 6.3. Add a set X of $2n/k = o(n)$ extra vertices connected to all inputs $V_1 = \{x_1, \dots, x_n\}$ and all outputs $V_3 = \{y_1, \dots, y_n\}$. By Lemma 6.3, the graph has at most $n^{3/2}/k = o(n^{3/2})$ edges. Hence it remains only to prove that this circuit computes $Perm^n(\pi, \vec{x})$.

Fix a permutation π. By repeated applications of Lemma 6.3, we can choose disjoint embeddings of G_k into the graph G such that at most n/k inputs of G are left uncovered, and whenever an embedding maps an output to y_i, the corresponding input is mapped to $x_{\pi(i)}$.

For each of the chosen embeddings of G_k, take one of the extra vertices in X and connect inputs of G_k with outputs of G_k through it. Thus for an input x_j of G_k and an output y_i of G_k, we have just one path $x \to y$ through X if $j = \pi(i)$ and one path through X and one path in G_k if $j \neq \pi(i)$. Connect the inputs and the outputs not covered by the embeddings of G_k paths through the remaining vertices in X. This is possible since we need at most n/k vertices in X for G_k's and at most n/k vertices in X for inputs and outputs not covered by G_k's.

Assign values to the edges as follows. All edges of the selected paths through X have value 1. For each embedding of G_k, the edges going from V_1 to V_2 have value 1 and the edges going from V_2 to V_3 have value -1. The remaining edges have value 0. We set the linear function at each vertex to be the sum of the values on its predecessors, each multiplied by the value of the connecting edge. Thus for $j = \pi(i)$, there is only one path from x_j to y_i with nonzero value and its value is 1. For $j \neq \pi(j)$, there are either no nonzero paths or one with value 1 and one with value -1. Hence the circuit computes $Perm^n(\pi, \vec{x})$. □

For the rest of this section, we set

$$p = n^{\frac{5/4-k^2}{2(k^2-k)}} = n^{-\frac{1}{2}-\frac{4k-5}{8(k^2-k)}}.$$

Let $\mathbf{R}_{p,m}$ denote the random graph with vertices

$$\begin{aligned} V_1 &= \{x_1, \dots, x_m\}, \\ V_2 &= \{z_1, \dots, z_{mk^2}\}, \\ V_3 &= \{y_1, \dots, y_m\} \end{aligned}$$

and each edge from $V_1 \times V_2$ and $V_2 \times V_3$ chosen independently at random with probability p.

Proof of Lemma 6.3. We prove that for sufficiently large n, the graph $\mathbf{R}_{p,n}$ satisfies the conditions of Lemma 6.3 with high probability. The expected number of edges of $\mathbf{R}_{p,n}$ is

$$2n^2k^2p = 2k^2n^{\frac{3}{2}-\frac{4k-5}{8(k^2-k)}} = o(n^{3/2}/k);$$

hence the condition on the number of edges is satisfied with high probability.

The number of possible quadruples W_1, W_2, W_3, f is bounded by $e^{O(n \log n)}$. For a fixed quadruple W_1, W_2, W_3, f, the probability that an appropriate embedding of G_k exists is the same as the probability that there exists an embedding g of G_k into $\mathbf{R}_{p,m}$ satisfying

$$g(u_i) = x_j \quad \text{iff} \quad g(v_i) = y_j, \tag{6}$$

where $m = |W_1|$. Hence the proof is completed by the following lemma. □

LEMMA 6.4. *Let $k > 2$ be an arbitrary constant. Then for n sufficiently large and any m, $n/k \le m \le n$,*

$$\Pr[\textit{there exists no embedding of } G_k \textit{ into } \mathbf{R}_{p,m} \textit{ satisfying (6)}] \le e^{-n^{5/4+o(1)}}.$$

Proof. Let $X(H, G)$ denote the number of embeddings of H into G satisfying (6). We want to prove an upper bound on $Pr[X(G_k, \mathbf{R}_{p,m}) = 0]$. We use Janson's inequality, see [3, Theorem 1.1, Chapter 8, p. 96].

Let $G_k(1), \ldots, G_k(i), \ldots$ denote all possible occurrences of G_k in $\mathbf{R}_{p,m}$ satisfying the condition (6). Let

$$\begin{aligned}
\varepsilon &= \Pr[G_k(i) \subseteq \mathbf{R}_{p,m}],\\
\mu &= E\big(X(G_k, \mathbf{R}_{p,m})\big),\\
\Delta &= \sum_{i \sim j} \Pr[G_k(i) \subseteq \mathbf{R}_{p,m} \wedge G_k(j) \subseteq \mathbf{R}_{p,m}],
\end{aligned}$$

where we sum over the pairs of distinct occurrences of G_k which have at least one common edge. By Janson's inequality, we have

$$\Pr[X(G_k, \mathbf{R}_{p,m}) = 0] \le e^{-\mu + \frac{1}{1-\varepsilon}\frac{\Delta}{2}}.$$

Clearly, $\varepsilon = p^{2(k^2-k)} = o(1)$. We prove that $\mu = \Theta(n^{5/4})$ and $\Delta = o(\mu)$, which gives the desired bound.

First, we compute μ as the number of all possible occurrences times the probability of one fixed occurrence. (Remember that k is a constant.)

$$\mu = \binom{m}{k}\binom{mk^2}{k^2-k} p^{2(k^2-k)} = \Theta(n^k n^{k^2-k} n^{\frac{5/4-k^2}{2(k^2-k)} \cdot 2(k^2-k)}) = \Theta(n^{5/4}).$$

We can bound Δ as in [18] by

$$\Delta \le \sum_{\emptyset \ne H \subset G_k} \frac{\mu^2}{E(X(H, \mathbf{R}_{p,m}))} X(H, G_k)^2,$$

where the sum is over all graphs H with at least one edge which may occur as intersections $G_k(i) \cap G_k(j)$ for $i \ne j$. Since k is a constant, the number of such graphs is bounded and the values of $X(H, G_k)$ are bounded as well, and therefore

$$\Delta \le O\left(\frac{\mu^2}{\min E(X(H, \mathbf{R}_{p,m}))}\right).$$

For a given graph H, let r, $1 \le r \le k$, be the number of its vertices from V_1. The vertices on the middle level of H have degree at most 2. Let s_0, s_1, and s_2 denote the number of vertices of H on the middle level whose degree in H is 0, 1, and 2, respectively. Now we estimate the growth rate of $E(X(H, \mathbf{R}_{p,m}))$ and prove that it is at least $\omega(n^{5/4}) = \omega(\mu)$ for every H.

$$\begin{aligned}
E(X(H, \mathbf{R}_{p,m})) &= \binom{m}{r}\binom{mk^2}{s_2}\binom{mk^2-s_2}{s_1}\binom{mk^2-s_2-s_1}{s_0} p^{2s_2+s_1}\\
&\approx n^{r+s_2+s_1+s_0} p^{2s_2+s_1} \ge n^r (np^2)^{s_2} (np)^{s_1}\\
&= n^{r - s_2\left(\frac{4k-5}{4(k^2-k)}\right) + s_1\left(\frac{1}{2} - \frac{4k-5}{8(k^2-k)}\right)}.
\end{aligned}$$

We estimate the exponent separately for the following three cases.

(i) Let $r = 1$. Then $s_2 = 0$ since in G_k there is no path from u_i to v_i for any i. Since H is has at least one edge, $s_1 \geq 1$ and the exponent is at least $1 + 1/2 - (4k - 5)/(8(k^2 - k)) > 5/4$.

(ii) Suppose $2 \leq r \leq k - 1$. Since there are at most $r^2 - r$ vertices of degree 2 on the middle level of H, the exponent is minimized for $s_1 = 0$ and $s_2 = r^2 - r$, and then it is

$$\alpha = r - (r^2 - r)\frac{4k - 5}{4(k^2 - k)}.$$

This is a concave-down function in r; thus it is minimized at one of the endpoints of the interval, i.e., for $r = 2$ or $r = k-1$. For $r = 2$, $\alpha = 2+(4k-5)/(2(k^2-k)) > 5/4$. For $r = k - 1$, a short calculation gives $\alpha = 9/4 - 5/(2k) > 5/4$.

(iii) Finally, let $r = k$. Then $s_2 < k^2 - k$ since H is required to be a proper subgraph of G_k. Thus the exponent is minimized at $s_2 = k^2 - k - 1$ and $s_1 = 0$, when the value is $5/4 + (4k - 5)/(4(k^2 - k)) > 5/4$.

Hence $\Delta \leq \mu^2/\omega(\mu) = o(\mu)$ and we can apply Janson's inequality. □

7. The chromatic number of a random graph. In this section, we prove the estimate on the chromatic number of a random graph for a large probability of an edge, Theorem 4.4. In our presentation of the proof, we follow the lines of [3, Chapter 7]. The proof is based on the following lemma, where clique(G) denotes the size of a maximal clique in a graph G.

LEMMA 7.1. *For every $\varepsilon > 0$, there exist $\delta > 0$ and n_0 such that*

$$\Pr\left[\text{clique}(\mathbf{G}(n,p)) > (2-\varepsilon)\frac{\log n}{-\log p}\right] > 1 - \exp(-n^{1+\delta})$$

as long as $1/n^{\frac{1}{2}-\varepsilon} < p < 1/8$ and $n \geq n_0$.

First, we deduce Theorem 4.4 from Lemma 7.1. Set $m = n/(\log n)^2$, $\varepsilon' = \varepsilon/2$ and $p = 1 - q$. Let A be the event that every induced subgraph H of $\mathbf{G}(n, q)$ with m vertices has an independent set of size $\alpha(H) > (2-\varepsilon)(\log m)/(-\log(1-q))$. Note that $(2-\varepsilon')(\log m)/(-\log(1-q)) \geq (2-\varepsilon)(\log n)/(-\log(1-q))$ for n sufficiently large. Then by Lemma 7.1 applied with $\varepsilon' = \varepsilon/2$,

$$\Pr(A) > 1 - \binom{n}{m}\exp(-n^{1+\delta}) > 1 - \exp\left(-n^{\delta}\right).$$

This means that with probability at least $1 - \exp\left(-n^{\delta}\right)$, the random graph $\mathbf{G}(n, q)$ can be colored by

$$\frac{n-m}{(2-\varepsilon)\frac{\log m}{-\log(1-q)}} + m \leq \left(\frac{1}{2} + \varepsilon\right)\frac{-n\log(1-q)}{\log n}$$

colors. This finishes the proof of Theorem 4.4.

Before we give a proof of Lemma 7.1 we prove another lemma. Let $p = p(n)$ be given, $1/n^{\frac{1}{2}-\varepsilon} < p < 1/8$, $\varepsilon > 0$ constant. Let $k = k(n)$ be the largest integer such that

$$\binom{n}{k} p^{\binom{k}{2}} \geq n^3; \tag{7}$$

note that $(1-\varepsilon)(\log n)/(-\log p) \le k \le 2(\log n)/(-\log p)+1$ for every sufficiently large n. Let $Y = Y(\mathbf{G}(n,p))$ be the maximal size of a family of edge-disjoint k-cliques in $\mathbf{G}(n,p)$.

LEMMA 7.2. $E(Y) \ge n^2 p/(2k^5)$.

Proof. Let K denote the family of all k-cliques so that $E(|K|) = \binom{n}{k} p^{\binom{k}{2}} \ge n^3$. Let W be the set of ordered pairs $\{S,T\}$ of k-cliques of $\mathbf{G}(n,p)$ with $2 \le |S \cap T| < k$. Then

$$E(|W|) = \binom{n}{k} \sum_{i=2}^{k-1} \binom{k}{i} \binom{n-i}{k-i} p^{2\binom{k}{2}-\binom{i}{2}}.$$

Set

$$A_i = \binom{k}{i} \binom{n-i}{k-i} p^{2\binom{k}{2}-\binom{i}{2}}.$$

Since for $p < 1/8$, the sequence

$$B_i = \frac{A_i}{A_{i+1}} = \frac{(n-i)(i+1)}{(k-i)^2} p^i$$

decreases, we infer that for some i_0, $A_i \ge A_{i+1}$ for $i < i_0$ and $A_i \le A_{i+1}$ for $i \ge i_0$. A straightforward calculation using (7) shows that $A_{k-1} \le A_2$. Hence $A_i \le A_2$ for all $i = 2, \dots, k-1$ and thus

$$E(|W|) \le \binom{n}{k} k \binom{k}{2} \binom{n-2}{k-2} p^{2\binom{k}{2}-1}.$$

Let K' be a random subfamily of K, where every $S \in K$ is chosen independently with

$$\Pr[S \in K'] = \gamma = \frac{1}{4p^{\binom{k}{2}-1} k \binom{k}{2} \binom{n-2}{k-2}}.$$

Now construct a family $L \subseteq K'$ by removing each pair $S,T \in K'$ such that $\{S,T\} \in W$. No two cliques in L intersect; hence $|L| \le Y$. Thus

$$\begin{aligned} E(Y) &\ge E(|L|) \ge \gamma |K| - 2\gamma^2 E(|W|) \\ &= \gamma \binom{n}{k} p^{\binom{k}{2}} - 2\gamma^2 \binom{n}{k} k \binom{k}{2} \binom{n-2}{k-2} p^{2\binom{k}{2}-1} = \frac{1}{2} \gamma \binom{n}{k} p^{\binom{k}{2}} \ge \frac{n^2 p}{2k^5}. \end{aligned}$$

□

Furthermore, we need the following definition. A *martingale* is a sequence $Y_0, Y_1, \dots, Y_m$ of random variables such that for $0 \le i < m$,

$$E(Y_{i+1}|Y_i) = Y_i$$

holds. We use the following estimate.

THEOREM 7.3 (Azuma's inequality [3, 5]). *Let $0 = X_0, X_1, \dots, X_m$ be a martingale with $|X_{i+1} - X_i| \le 1$ for all $0 \le i < m$. Let $\lambda > 0$ be arbitrary. Then*

$$\Pr[X_m > \lambda] < e^{-\frac{\lambda^2}{2m}}.$$

Proof of Lemma 7.1. Let $Y_0, Y_1, \ldots, Y_m$, $m = \binom{n}{2}$, be the edge-exposure martingale on $\mathbf{G}(n,p)$ with function Y defined above. (See [3] for the definitions.) Y_i is the conditional expectation of Y when we know the first i edges (for a fixed arbitrary ordering of the edges); thus $y_0 = E(Y)$ and $Y_m = Y$. Since Y is the cardinality of a family of edge-disjoint cliques, $|Y_{i+1} - Y_i| \leq 1$ holds, and hence by Azuma's inequality, taking $X_i = Y_i - E(Y)$,

$$\begin{aligned}\Pr[\text{clique}(\mathbf{G}) < k] = \Pr[Y = 0] &\leq \Pr[Y - E(Y) \leq -E(Y)] \\ &\leq \exp\left(\frac{-E(Y)^2}{2\binom{n}{2}}\right) = \exp\left(-\frac{n^2p^2}{4k^{10}}(1+o(1))\right) \\ &\leq \exp(-n^{1+\delta}). \qquad \square\end{aligned}$$

8. Conclusions and open problems. We have shown relations between circuit complexity, multiparty complexity, and the algebraic characteristics of rigidity and rigidity rank. The particular versions of these concepts that we have considered are related to the problem of proving superlinear lower bounds on circuit complexity and on the size of bounded-depth circuits with arbitrary Boolean gates.

We have proved some upper bounds. Though it is only a small improvement, it disproves some earlier conjectures. The conclusion is that the problems are apparently harder than was expected.

We have also improved some lower bounds. This is an example of how nontrivial results in combinatorics may help in complexity theory. Still, the gaps between upper and lower bounds remain very large.

There many open problems in this area; several of them are implicit in the above text. Here we state only two which we consider to be the most challenging.

Problem 1. Does the function $Shift^n$ have circuits of size $O(n)$ and depth $O(\log n)$? This is open for semilinear circuits as well.

Problem 2. Improve the easy lower bound $\Omega(\sqrt{n})$ for the simultaneous communication complexity of $shift^n$.

We believe that $\mathrm{SCC}(shift^n)$ is large; however, even our slightly larger lower bound $\Omega(\sqrt{n \log n})$ from Corollary 5.3 works only under the restriction that Player 2 always sends a substring of the input string $\vec{x}$.

Acknowledgments. The first author would like to thank to Wolfgang Maass, from whom he learned problems on circuits with arbitrary Boolean functions as gates, and to Avi Wigderson for an idea which eventually led to the constructive upper bound in Theorem 4.2. We also thank Steven Rudich for valuable comments.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi, *A note on Ramsey numbers*, J. Combin. Theory Ser. A, 29 (1980), pp. 354–360.

[2] N. Alon and P. Pudlák, *Superconcentrators of depth* 2 *and* 3: *Odd levels help (rarely)*, J. Comput. System Sci., 48 (1994), pp. 194–202.

[3] N. Alon and J. H. Spencer, *The Probabilistic Method*, John Wiley, New York, 1992.

[4] A. Ambainis, *Upper bounds on multiparty communication complexity of shifts*, in Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 1046, Springer-Verlag, Berlin, 1996, pp. 631–642.

[5] K. Azuma, *Weighted sums of certain dependent random variables*, Tôhoku Math. J., 3 (1967), pp. 357–367.

[6] L. Babai, P. Kimmel, and S. V. Lokam, *Simultaneous messages vs. communication*, in Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 900, Springer-Verlag, Berlin, 1995, pp. 361–372.

[7] L. Babai, N. Nisan, and M. Szegedy, *Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs*, J. Comput. System Sci., 45 (1992), pp. 204–232.
[8] R. Beigel and J. Tarui, *On ACC*, Comput. Complexity, 4 (1994), pp. 350–366.
[9] B. Bollobás, *The chromatic number of random graphs*, Combinatorica, 8 (1988), pp. 49–55.
[10] B. Bollobás, *Random Graphs*, Academic Press, New York, 1985.
[11] C. Damm, S. Jukna, and J. Sgall, *Some bounds for multiparty communication complexity of pointer jumping*, Comput. Complexity, to appear.
[12] H. F. de Groote, *Lectures on the Complexity of Bilinear Problems*, Lecture Notes in Comput. Sci. 245, Springer-Verlag, Berlin, 1987.
[13] D. Dolev, C. Dwork, N. Pippenger, and A. Wigderson, *Superconcentrators, generalizers and generalized connectors with limited depth*, in Proc. 15th Annual ACM Symposium on Theory of Computing, ACM, New York, 1983.
[14] J. Friedman, *A note on matrix rigidity*, Combinatorica, 13 (1993), pp. 235–239.
[15] J. Håstad and M. Goldmann, *On the power of small-depth threshold circuits*, Comput. Complexity, 1 (1991), pp. 113–129.
[16] A. K. Chandra, M. L. Furst, and R. J. Lipton, *Multi-party protocols*, in Proc. 15th Annual ACM Symposium on Theory of Computing, ACM, New York, 1983, pp. 94–99.
[17] D. V. Chudnovsky and G. V. Chudnovsky, *Algebraic complexities and algebraic curves over finite fields*, Proc. Nat. Acad. Sci. U.S.A., 84 (1987), pp. 1739–1743.
[18] S. Janson, T. Luczak, and A. Ruciński, *An exponential bound for the probability of nonexistence of a specified subgraph in a random graph*, in Random Graphs '87, M. Karoński et al., eds., John Wiley, New York, 1990, pp. 73–87.
[19] E. Kushilevitz and N. Nisan, *Communication Complexity*, Cambridge University Press, Cambridge, UK, to appear.
[20] N. Nisan and A. Wigderson, *Rounds in communication complexity revisited*, SIAM J. Comput., 22 (1993), pp. 211–219.
[21] N. Pippenger, *The complexity of computations by networks*, IBM J. Res. Develop., 31 (1987), pp. 235–243.
[22] N. Pippenger and L. G. Valiant, *Shifting graphs and their applications*, J. Assoc. Comput. Mach., 23 (1976), pp. 423–432.
[23] N. Pippenger and A. C.-C. Yao, *Rearrangeable networks with limited depth*, SIAM J. Algebraic Discrete Meth., 3 (1982), pp. 411–417.
[24] P. Pudlák, *Communication in bounded depth circuits*, Combinatorica, 14 (1994), pp. 203–216.
[25] P. Pudlák and V. Rödl, *Modified ranks of tensors and the size of circuits*, in Proc. 25th Annual ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 523–531.
[26] P. Pudlák and P. Savický, *On shifting networks*, Theoret. Comput. Sci., 116 (1993), pp. 415–419.
[27] A. A. Razborov, *On rigid matrices*, preprint, Mathematical Institute, Academy of Sciences of USSR, Moscow, 1989 (in Russian).
[28] J. Sgall, *A note on multiparty communication complexity of shifts*, manuscript, 1992.
[29] M. A. Shokrollahi, *Beiträge zur codierungs- und komplexitätstheorie mittels algebraischer funktionkörper*, Bayreuth. Math. Schr., 39 (1991).
[30] V. Shoup and R. Smolensky, *Lower bounds for polynomial evaluation and interpolation*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 378–383.
[31] V. Strassen, *Vermeidung von divisionen*, Crelles J. Reine Angew. Math., 264 (1973), pp. 184–202.
[32] L. G. Valiant, *Graph-theoretic arguments in low level complexity*, in Proc. 6th Mathematical Foundations of Computer Sci., Lecture Notes in Comput. Sci. 53, Springer-Verlag, Berlin, 1977, pp. 162–176.
[33] L. G. Valiant, *Why is Boolean complexity theory difficult?*, in Boolean Function Complexity, M. S. Paterson, ed., Cambridge University Press, Cambridge, UK, 1992, pp. 84–94.
[34] A. C.-C. Yao, *On ACC and threshold circuits*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1990, pp. 619–627.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 634–653, June 1997

002

UNAMBIGUOUS COMPUTATION: BOOLEAN HIERARCHIES AND SPARSE TURING-COMPLETE SETS*

LANE A. HEMASPAANDRA† AND JÖRG ROTHE‡

Abstract. It is known that for any class $\mathcal{C}$ closed under *union and intersection*, the Boolean closure of $\mathcal{C}$, the Boolean hierarchy over $\mathcal{C}$, and the symmetric difference hierarchy over $\mathcal{C}$ all are equal. We prove that these equalities hold for any complexity class closed under *intersection*; in particular, they thus hold for unambiguous polynomial time (UP). In contrast to the NP case, we prove that the Hausdorff hierarchy and the nested difference hierarchy over UP both fail to capture the Boolean closure of UP in some relativized worlds.

Karp and Lipton proved that if *nondeterministic* polynomial time has sparse Turing-complete sets, then the polynomial hierarchy collapses. We establish the first consequences from the assumption that *unambiguous* polynomial time has sparse Turing-complete sets: (a) UP $\subseteq \mathrm{Low}_2$, where Low_2 is the second level of the low hierarchy, and (b) each level of the unambiguous polynomial hierarchy is contained one level lower in the promise unambiguous polynomial hierarchy than is otherwise known to be the case.

Key words. unambiguous computation, Boolean hierarchy, sparse Turing-complete sets

AMS subject classifications. 68Q15, 68Q10, 03D15

PII. S0097539794261970

1. Introduction.

NP and NP-based hierarchies—such as the polynomial hierarchy [47, 57] and the Boolean hierarchy over NP [9, 10, 41]—have played such a central role in complexity theory, and have been so thoroughly investigated, that it would be natural to take them as predictors of the behavior of other classes or hierarchies. However, over and over during the past decade it has been shown that NP is a singularly poor predictor of the behavior of other classes (and, to a lesser extent, that hierarchies built on NP are poor predictors of the behavior of other hierarchies).

As examples regarding hierarchies, we have the following: though the polynomial hierarchy possesses downward separation (that is, if its low levels collapse, then all its levels collapse) [47, 57], downward separation does not hold "robustly" (i.e., in every relativized world) for the exponential time hierarchy [24, 36] or for limited-nondeterminism hierarchies [32] (see also [4]). As examples regarding UP, we have the following: NP has $\leq_m^p$-complete sets, but UP does not robustly possess $\leq_m^p$-complete sets [22] or even $\leq_T^p$-complete sets [31]; NP positively relativizes, in the sense that it collapses to P if and only if it does so with respect to every tally oracle [45] (see also [1]), but UP does not robustly positively relativize [29]; NP has "constructive programming systems," but UP does not robustly have such systems [52]; NP (actually, nondeterministic computation) admits time hierarchy theorems [25], but it is an open question whether unambiguous computation has nontrivial time hierarchy theorems; NP displays upward separation (that is, NP − P contains sparse sets if and

* Received by the editors January 24, 1994; accepted for publication June 7, 1995.
http://www.siam.org/journals/sicomp/26-3/26197.html

† Department of Computer Science, University of Rochester, Rochester, NY 14627 (lane@cs.rochester.edu). The research of this author was supported in part by NSF grants CCR-8957604, INT-9116781/JSPS-ENGR-207, CCR-9322513, and INT-9513368/DAAD-315-PRO-of-ab and an NAS/NRC COBASE grant.

‡ Institut für Informatik, Friedrich-Schiller-Universität Jena, 07743 Jena, Germany (rothe@informatik.uni-jena.de). The research of this author was supported in part by a grant from the DAAD and NSF grants CCR-8957604 and INT-9513368/DAAD-315-PRO-of-ab and was done in part while visiting the University of Rochester.

only if NE $\neq$ E) [24], but it is not known whether UP does (see [32], which shows that R and BPP do not robustly display upward separation, and [51], which shows that FewP does possess upward separation).

In light of the above list of the many ways in which NP parts company with UP, it is clear that we should not merely assume that results for NP hold for UP, but, rather, we must carefully check to see to what extent, if any, results for NP suggest results for UP. In this paper, we study, for UP, two topics that have been intensely studied for the NP case: the structure of Boolean hierarchies, and the effects of the existence of sparse Turing-complete/Turing-hard sets.

For the Boolean hierarchy over NP, which has generated quite a bit of interest and the collapse of which is known to imply the collapse of the polynomial hierarchy [37, 16, 3], a large number of definitions are known to be equivalent. For example, for NP, all the following coincide [9]: the Boolean closure of NP, the Boolean (alternating sums) hierarchy, the nested difference hierarchy, and the Hausdorff hierarchy. The symmetric difference hierarchy also characterizes the Boolean closure of NP [41]. In fact, these equalities are known to hold for all classes that contain Σ^* and $\emptyset$ and are closed under union and intersection [26, 9, 41, 5, 20, 15, 14]. In section 3, we prove that both the symmetric difference hierarchy (SDH) and the Boolean hierarchy (CH) remain equal to the Boolean closure (BC) *even in the absence of the assumption of closure under union.* That is, for any class $\mathcal{K}$ containing Σ^* and $\emptyset$ and closed under intersection (e.g., UP, US, and DP, first defined, respectively, in [59], [6], and [50] and each of which is not currently known to be closed under union): $\mathrm{SDH}(\mathcal{K}) = \mathrm{CH}(\mathcal{K}) = \mathrm{BC}(\mathcal{K})$. However, for the remaining two hierarchies, we show that not all classes containing Σ^* and $\emptyset$ and closed under intersection robustly display equality. In particular, the Hausdorff hierarchy over UP and the nested difference hierarchy over UP both fail to robustly capture the Boolean closure of UP. In fact, the failure is relatively severe; we show that even low levels of other Boolean hierarchies over UP—the third level of the symmetric difference hierarchy and the fourth level of the Boolean (alternating sums) hierarchy—fail to be robustly captured by either the Hausdorff hierarchy or the nested difference hierarchy.

It is well known, thanks to the work of Karp and Lipton [39] (see also the related references given in section 4), that if NP has sparse Turing-hard sets, then the polynomial hierarchy collapses. Unfortunately, the promise-like definition of UP—its unambiguity, the very core of its nature—seems to block any similarly strong claim for UP and the unambiguous polynomial hierarchy (which was introduced recently by Niedermeier and Rossmanith [48]). Section 4 studies this issue and shows that if UP has sparse Turing-complete sets, then the levels of the unambiguous polynomial hierarchy "slip down" slightly in terms of their location within the promise unambiguous polynomial hierarchy (a version of the unambiguous polynomial hierarchy that requires only that computations *actually executed* be unambiguous), i.e., the kth level of the unambiguous polynomial hierarchy is contained in the $(k-1)$st level of the promise unambiguous polynomial hierarchy. Various related results are also established. For example, if UP has Turing-hard sparse sets, then (a) UP $\subseteq \mathrm{Low}_2$, where Low_2 is the second level of the low hierarchy [53], and (b) the kth level of the unambiguous polynomial hierarchy can be accepted via a deterministic polynomial-time Turing transducer given access to both a Σ_2^p set and the $(k-1)$st level of the promise unambiguous polynomial hierarchy.

2. Notation. In general, we adopt the standard notations of Hopcroft and Ullman [35]. Fix the alphabet $\Sigma = \{0, 1\}$. Σ^* is the set of all strings over Σ. For each

string $u \in \Sigma^*$, $|u|$ denotes the length of u. The empty string is denoted by ϵ. For each set $L \subseteq \Sigma^*$, $\|L\|$ denotes the cardinality of L and $\overline{L} = \Sigma^* - L$ denotes the complement of L. $L^{=n}$ ($L^{\leq n}$) is the set of all strings in L having length n (less than or equal to n). Let Σ^n and $\Sigma^{\leq n}$ be shorthands for $(\Sigma^*)^{=n}$ and $(\Sigma^*)^{\leq n}$, respectively. A set S is said to be *sparse* if there is a polynomial q such that for every $m \geq 0$, $\|S^{\leq m}\| \leq q(m)$. To encode a pair of strings, we use a polynomial-time computable pairing function, $\langle \cdot, \cdot \rangle : \Sigma^* \times \Sigma^* \to \Sigma^*$, that has polynomial-time computable inverses; this notion is extended to encode every k-tuple of strings, in the standard way. Let $\leq_{\text{lex}}$ denote the standard quasi-lexicographical ordering on Σ^*, that is, for strings x and y, $x \leq_{\text{lex}} y$ if either $x = y$, or $|x| < |y|$, or ($|x| = |y|$ and there exists some $z \in \Sigma^*$ such that $x = z0u$ and $y = z1v$). $x <_{\text{lex}} y$ indicates that $x \leq_{\text{lex}} y$ but $x \neq y$.

For sets A and B, their join, $A \oplus B$, is $\{0x \,|\, x \in A\} \cup \{1x \,|\, x \in B\}$, and their symmetric difference, $A \,\Delta\, B$, is $(A - B) \cup (B - A)$. For any class $\mathcal{C}$, define $\text{co}\mathcal{C} \stackrel{\text{df}}{=} \{L \,|\, \overline{L} \in \mathcal{C}\}$, and let $\text{BC}(\mathcal{C})$ denote the Boolean algebra generated by $\mathcal{C}$, i.e., the smallest class containing $\mathcal{C}$ and closed under all Boolean operations. For any classes $\mathcal{A}$ and $\mathcal{B}$, let $\mathcal{A} \oplus \mathcal{B}$ denote the class $\{A \oplus B \,|\, A \in \mathcal{A} \wedge B \in \mathcal{B}\}$. Similarly, for classes $\mathcal{C}$ and $\mathcal{D}$ of sets, define

$$\mathcal{C} \wedge \mathcal{D} \stackrel{\text{df}}{=} \{A \cap B \,|\, A \in \mathcal{C} \wedge B \in \mathcal{D}\}, \qquad \mathcal{C} \;\boldsymbol{\Delta}\; \mathcal{D} \stackrel{\text{df}}{=} \{A \,\Delta\, B \,|\, A \in \mathcal{C} \wedge B \in \mathcal{D}\},$$
$$\mathcal{C} \vee \mathcal{D} \stackrel{\text{df}}{=} \{A \cup B \,|\, A \in \mathcal{C} \wedge B \in \mathcal{D}\}, \qquad \mathcal{C} - \mathcal{D} \stackrel{\text{df}}{=} \{A - B \,|\, A \in \mathcal{C} \wedge B \in \mathcal{D}\}.$$

We will abbreviate "polynomial-time deterministic (nondeterministic) Turing machine" by DPM (NPM). An *unambiguous* (sometimes called categorical) polynomial-time Turing machine (UPM) is an NPM that on no input has more than one accepting computation path [59]. UP is the class of all languages that are accepted by some UPM [59]. For the respective oracle machines, we use the shorthands DPOM, NPOM, and UPOM.

Note, crucially, that whether a machine is categorical or not depends on its oracle. In fact, it is well known that machines that are categorical with respect to all oracles accept only easy languages [23] and thus create a polynomial hierarchy analogue that is completely contained in a low level of the polynomial hierarchy (Allender and Hemachandra as cited in [29]). So, when we speak of a UPOM, we will simply mean an NPOM that, with the oracle the machine has in the context being discussed, happens to be categorical.

For any Turing machine M, $L(M)$ denotes the set of strings accepted by M, and the notation $M(x)$ means "M on input x." For any oracle Turing machine M and any oracle set A, $L(M^A)$ denotes the set of strings accepted by M relative to A, and the notation $M^A(x)$ means "M^A on input x." Without loss of generality, we assume each NPM and NPOM (in our standard enumeration of such machines) M has the property that for every n, there is an integer ℓ_n such that, for every x of length n, every path of $M(x)$ is of length ℓ_n, and furthermore, in the case of oracle machines, that ℓ_n is independent of the oracle. Let A and B be sets. We say A is *Turing reducible* to B (denoted by $A \leq_T^p B$ or $A \in \text{P}^B$) if there is a DPOM M such that $A = L(M^B)$. A set B is *Turing-hard* for a complexity class $\mathcal{C}$ if for all $A \in \mathcal{C}$, $A \leq_T^p B$. A set B is *Turing-complete* for $\mathcal{C}$ if B is Turing-hard for $\mathcal{C}$ and $B \in \mathcal{C}$.

3. Boolean hierarchies over classes closed under intersection. The Boolean hierarchy is a natural extension of the classes NP [17, 44] and DP $\stackrel{\text{df}}{=}$ NP $\wedge$ coNP [50]. Both NP and DP contain natural problems, as do the levels of the Boolean hierarchy. For example, graph minimal uncolorability is known to be complete for DP [13]. Note

that DP clearly is closed under intersection but is not closed under union unless the polynomial hierarchy collapses (due to [37]; see also [15, 14]).

DEFINITION 3.1. [9, 41, 26] *Let $\mathcal{K}$ be any class of sets.*

1. *The* Boolean ("alternating sums") hierarchy over $\mathcal{K}$*:*

$$\mathrm{C}_1(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathcal{K}, \quad \mathrm{C}_k(\mathcal{K}) \stackrel{\mathrm{df}}{=} \begin{cases} \mathrm{C}_{k-1}(\mathcal{K}) \vee \mathcal{K} & \text{if } k \text{ is odd,} \\ \mathrm{C}_{k-1}(\mathcal{K}) \wedge \mathrm{co}\mathcal{K} & \text{if } k \text{ is even,} \end{cases} \quad k \geq 2, \quad \mathrm{CH}(\mathcal{K}) \stackrel{\mathrm{df}}{=} \bigcup_{k \geq 1} \mathrm{C}_k(\mathcal{K}).$$

2. *The* nested difference hierarchy over $\mathcal{K}$*:*

$$\mathrm{D}_1(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathcal{K}, \qquad \mathrm{D}_k(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathcal{K} - \mathrm{D}_{k-1}(\mathcal{K}), \quad k \geq 2, \qquad \mathrm{DH}(\mathcal{K}) \stackrel{\mathrm{df}}{=} \bigcup_{k \geq 1} \mathrm{D}_k(\mathcal{K}).$$

3. *The* Hausdorff ("union of differences") hierarchy over $\mathcal{K}$:[1]

$$\mathrm{E}_1(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathcal{K}, \ \mathrm{E}_2(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathcal{K} - \mathcal{K}, \ \mathrm{E}_k(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathrm{E}_2(\mathcal{K}) \vee \mathrm{E}_{k-2}(\mathcal{K}), \ k > 2, \ \mathrm{EH}(\mathcal{K}) \stackrel{\mathrm{df}}{=} \bigcup_{k \geq 1} \mathrm{E}_k(\mathcal{K}).$$

4. *The* symmetric difference hierarchy over $\mathcal{K}$*:*

$$\mathrm{SD}_1(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathcal{K}, \qquad \mathrm{SD}_k(\mathcal{K}) \stackrel{\mathrm{df}}{=} \mathrm{SD}_{k-1}(\mathcal{K}) \ \boldsymbol{\Delta} \ \mathcal{K}, \quad k \geq 2, \qquad \mathrm{SDH}(\mathcal{K}) \stackrel{\mathrm{df}}{=} \bigcup_{k \geq 1} \mathrm{SD}_k(\mathcal{K}).$$

It is easily seen that for any X chosen from {C, D, E, SD}, if $\mathcal{K}$ contains $\emptyset$ and Σ^*, then for any $k \geq 1$,

$$\mathrm{X}_k(\mathcal{K}) \cup \mathrm{coX}_k(\mathcal{K}) \subseteq \mathrm{X}_{k+1}(\mathcal{K}) \cap \mathrm{coX}_{k+1}(\mathcal{K}).$$

The following fact is shown by an easy induction on n.

FACT 3.2. *For every class $\mathcal{K}$ of sets and every $n \geq 1$,*

1. $\mathrm{D}_{2n-1}(\mathcal{K}) = \mathrm{coC}_{2n-1}(\mathrm{co}\mathcal{K})$ *and*
2. $\mathrm{D}_{2n}(\mathcal{K}) = \mathrm{C}_{2n}(\mathrm{co}\mathcal{K})$.

Proof. The base case holds by definition. Suppose both statements of the fact to be true for $n \geq 1$. Then

$$\begin{aligned} \mathrm{D}_{2n+1}(\mathcal{K}) &= \mathcal{K} \wedge (\mathrm{co}\mathcal{K} \vee \mathrm{D}_{2n-1}(\mathcal{K})) \stackrel{\text{hyp.}}{=} \mathcal{K} \wedge (\mathrm{co}\mathcal{K} \vee \mathrm{coC}_{2n-1}(\mathrm{co}\mathcal{K})) \\ &= \mathcal{K} \wedge \mathrm{co}(\mathcal{K} \wedge \mathrm{C}_{2n-1}(\mathrm{co}\mathcal{K})) = \mathcal{K} \wedge \mathrm{coC}_{2n}(\mathrm{co}\mathcal{K}) \\ &= \mathrm{co}(\mathrm{co}\mathcal{K} \vee \mathrm{C}_{2n}(\mathrm{co}\mathcal{K})) = \mathrm{coC}_{2n+1}(\mathrm{co}\mathcal{K}) \end{aligned}$$

shows part 1 for $n+1$, and

$$\mathrm{D}_{2n+2}(\mathcal{K}) = \mathcal{K} - (\mathcal{K} - \mathrm{D}_{2n}(\mathcal{K})) \stackrel{\text{hyp.}}{=} \mathcal{K} \wedge (\mathrm{co}\mathcal{K} \vee \mathrm{C}_{2n}(\mathrm{co}\mathcal{K})) = \mathrm{C}_{2n+2}(\mathrm{co}\mathcal{K})$$

shows part 2 for $n+1$. □

COROLLARY 3.3. $\mathrm{CH(UP)} = \mathrm{coCH(UP)} = \mathrm{DH(coUP)}$ *and* $\mathrm{CH(coUP)} = \mathrm{coCH(coUP)} = \mathrm{DH(UP)}$.

[1] Hausdorff hierarchies [26] (see [9, 5, 20], respectively, for applications to NP, R, and ~~G~~P are interesting both in the case where, as in the definition here, the sets are arbitrary sets from $\mathcal{K}$, and, as is sometimes used in definitions, the sets from $\mathcal{K}$ are required to satisfy additional containment conditions. For classes closed under union and intersection, such as NP, the two definitions are identical, level by level [26] (see also [9]). In this paper, since UP, for example, is not known to be closed under union, the distinction is nontrivial.

We are interested in the Boolean hierarchies over classes closed under intersection (but perhaps not under union or complementation), such as UP, US, and DP. We state our theorems in terms of the class of primary interest to us in this paper, UP. However, many apply to any nontrivial class (i.e., any class containing Σ^* and $\emptyset$) closed under intersection (see Theorem 3.10). Although it has been proven in [9] and [41] that all the standard normal forms of Definition 3.1 coincide for NP,[2] the situation for UP seems to be different since UP is probably not closed under union. (The closure of UP under intersection is straightforward.) Thus all the relations among those normal forms have to be reconsidered for UP.

We first prove that the symmetric difference hierarchy over UP (or any class closed under intersection) equals the Boolean closure. Though Köbler, Schöning, and Wagner [41] proved this for NP, their proof gateways through a class whose proof of equivalence to the Boolean closure uses closure under union, and thus the following result is not implicit in their paper.

THEOREM 3.4. SDH(UP) = BC(UP).

Proof. The inclusion from left to right is clear. For the converse inclusion, it is sufficient to show that SDH(UP) is closed under all Boolean operations since BC(UP), by definition, is the smallest class of sets that contains UP and is closed under all Boolean operations. Let L and L' be arbitrary sets in SDH(UP). Then for some $k, \ell \geq 1$, there are sets $A_1, \ldots, A_k, B_1, \ldots, B_\ell$ in UP representing L and L':

$$L = A_1 \,\Delta\, \cdots \,\Delta\, A_k \quad \text{and} \quad L' = B_1 \,\Delta\, \cdots \,\Delta\, B_\ell.$$

So

$$L \cap L' = \left(\Delta_{i=1}^{k} A_i\right) \cap \left(\Delta_{j=1}^{\ell} B_j\right) = \Delta_{i \in \{1,\ldots,k\},\, j \in \{1,\ldots,\ell\}} (A_i \cap B_j),$$

and since UP is closed under intersection and SDH(UP) is (trivially) closed under symmetric difference, we clearly have that $L \cap L' \in$ SDH(UP). Furthermore, since $\overline{L} = \Sigma^* \,\Delta\, L$ implies that $\overline{L} \in$ SDH(UP), SDH(UP) is closed under complementation. Since all Boolean operations can be represented in terms of complementation and intersection, our proof is complete. □

Next, we show that for any class closed under intersection, instantiated below to the case of UP, the Boolean (alternating sums) hierarchy over the class equals the Boolean closure of the class. Our proof is inspired by the techniques used to prove equality in the case where closure under union may be assumed.

THEOREM 3.5. CH(UP) = BC(UP).

Proof. We will prove that SDH(UP) $\subseteq$ CH(UP). By Theorem 3.4, this will suffice.

Let L be any set in SDH(UP). Then there is a $k > 1$ (the case $k = 1$ is trivial) such that $L \in \mathrm{SD}_k(\mathrm{UP})$. Let $U_1, \ldots, U_k$ be the witnessing UP sets; that is, $L = U_1 \,\Delta\, U_2 \,\Delta\, \cdots \,\Delta\, U_k$. By the inclusion–exclusion rule, L satisfies the equalities below. For odd k,

$$L = \left(\cdots \left(\left((U_1 \cup U_2 \cup \cdots \cup U_k) \cap \left(\overline{\bigcup_{j_1 < j_2} (U_{j_1} \cap U_{j_2})} \right) \right) \right.\right.$$

[2] Due essentially to its closure under union and intersection, and this reflects a more general behavior of classes closed under union and intersection, as studied by Bertoni et al. [5] (see also [26, 9, 41, 15, 14]).

$$\cup \left(\bigcup_{j_1<j_2<j_3} (U_{j_1} \cap U_{j_2} \cap U_{j_3}) \right) \Bigg) \cap \cdots \cup \left(\bigcup_{j_1<\cdots<j_k} (U_{j_1} \cap \cdots \cap U_{j_k}) \right) \Bigg),$$

where each subscripted j term must belong to $\{1, \ldots, k\}$. For even k, we similarly have

$$L = \Bigg(\cdots \Bigg(\Bigg((U_1 \cup U_2 \cup \cdots \cup U_k) \cap \left(\overline{\bigcup_{j_1<j_2} (U_{j_1} \cap U_{j_2})} \right) \Bigg)$$
$$\cup \left(\bigcup_{j_1<j_2<j_3} (U_{j_1} \cap U_{j_2} \cap U_{j_3}) \right) \Bigg) \cap \cdots \cap \left(\overline{\bigcup_{j_1<\cdots<j_k} (U_{j_1} \cap \cdots \cap U_{j_k})} \right) \Bigg).$$

For notational convenience, let us use $A_1, \ldots, A_k$ to represent the respective terms in the above expressions (ignoring the complementations). By the closure of UP under intersection, each A_i, $1 \le i \le k$, is the union of $\binom{k}{i}$ UP sets $B_{i,1}, \ldots, B_{i,\binom{k}{i}}$. Using the fact that $\emptyset$ is clearly in UP, we can easily turn the union of n arbitrary UP sets (or the intersection of n arbitrary coUP sets) into an alternating sum of $2n-1$ UP sets. So for instance, $A_1 = U_1 \cup U_2 \cup \cdots \cup U_k$ can be written

$$\Big(\cdots \Big(\Big(\Big(U_1 \cap \overline{\emptyset} \Big) \cup U_2 \Big) \cap \overline{\emptyset} \Big) \cup \cdots \cup U_k \Big);$$

call this C_1. Clearly, $C_1 \in \mathrm{C}_{2k-1}(\mathrm{UP})$. To transform the above representation of L into an alternating sum of UP sets, we need two (trivial) transformations holding for any $m \ge 1$ and for arbitrary sets S and $T_1, \ldots, T_m$:

$$S \cap \left(\overline{T_1 \cup T_2 \cup \cdots \cup T_m} \right) = \left(\cdots \left(\left(S \cap \overline{T_1} \right) \cap \overline{T_2} \right) \cap \cdots \right) \cap \overline{T_m} \tag{3.1}$$

$$S \cup (T_1 \cup T_2 \cup \cdots \cup T_m) = (\cdots ((S \cup T_1) \cup T_2) \cup \cdots) \cup T_m. \tag{3.2}$$

Using (3.1) with $S = C_1$ and $T_1 = B_{2,1}, \ldots, T_m = B_{2,\binom{k}{2}}$ and the fact that $\emptyset$ is in UP, $A_1 \cap \overline{A_2}$ can be transformed into an alternating sum of UP sets; call this C_2. Now apply (3.2) with $S = C_2$ and $T_1 = B_{3,1}, \ldots, T_m = B_{3,\binom{k}{3}}$ to obtain, again using that $\emptyset$ is in UP, an alternating sum $C_3 = (A_1 \cap \overline{A_2}) \cup A_3$ of UP sets, and so on. Eventually, this procedure of alternately applying (3.1) and (3.2) will yield an alternating sum C_k of sets in UP that equals L. Thus $L \in \mathrm{CH}(\mathrm{UP})$. □

COROLLARY 3.6. SDH(UP) *and* CH(UP) *are both closed under all Boolean operations.*

Note that the proofs of Theorems 3.5 and 3.4 implicitly give a recurrence yielding an upper bound on the level-wise containments. We find the issue of equality to BC(UP), or lack thereof, to be the central issue, and thus we focus on that. Nonetheless, we point out in the corollary below that losing the assumption of closure under union seems to have exacted a price: though the hierarchies SDH(UP) and CH(UP) are indeed equal, the above proof embeds $\mathrm{SD}_k(\mathrm{UP})$ in an exponentially higher level of the C hierarchy over UP. Similarly, the proof of Theorem 3.4 embeds $\mathrm{C}_k(\mathrm{UP})$ in an exponentially higher level of SDH(UP).

COROLLARY 3.7.

1. *For each* $k \ge 1$, $\mathrm{SD}_k(\mathrm{UP}) \subseteq \mathrm{C}_{2^{k+1}-k-2}(\mathrm{UP})$.
2. *For each* $k \ge 1$, $\mathrm{C}_k(\mathrm{UP}) \subseteq \mathrm{SD}_{T(k)}(\mathrm{UP})$, *where* $T(k) = 2^k - 1$ *if* k *is odd, and* $T(k) = 2^k - 2$ *if* k *is even.*

Proof. For an $\mathrm{SD}_k(\mathrm{UP})$ set L to be placed into the $R(k)$th level of $\mathrm{CH}(\mathrm{UP})$, L is represented (in the proof of Theorem 3.5) as an alternating sum of k terms $A_1, \ldots, A_k$, each A_i consisting of $\binom{k}{i}$ UP sets $B_{i,j}$. In the subsequent transformation of L according to equations (3.1) and (3.2), each A_i requires as many as $\binom{k}{i} - 1$ additional terms $\emptyset$ or $\overline{\emptyset}$, respectively, to be inserted, and each such insertion brings us one level higher in the C hierarchy. Thus

$$R(k) = \sum_{i=1}^{k} \binom{k}{i} + \left(\binom{k}{i} - 1 \right) = -k + 2 \sum_{i=1}^{k} \binom{k}{i} = 2^{k+1} - k - 2.$$

A close inspection of the proof of $\mathrm{C}_k(\mathrm{UP}) \subseteq \mathrm{SD}_{T(k)}(\mathrm{UP})$ according to Theorem 3.4 leads to the recurrence

$$T(1) = 1 \quad \text{and} \quad T(k) = \begin{cases} 2T(k-1) + 3 & \text{if } k > 1 \text{ is odd,} \\ 2T(k-1) & \text{if } k > 1 \text{ is even} \end{cases}$$

since any set $L \in \mathrm{C}_k(\mathrm{UP})$ can be represented by sets $A \in \mathrm{C}_{k-1}(\mathrm{UP})$ and $B \in \mathrm{UP}$ as follows:

$$\begin{array}{ll} L = A \cup B = \overline{\overline{A} \cap \overline{B}} = \Sigma^* \,\Delta\, ((\Sigma^* \,\Delta\, A) \cap (\Sigma^* \,\Delta\, B)) & \text{if } k \text{ is odd,} \\ L = A \cap \overline{B} = A \cap (\Sigma^* \,\Delta\, B) & \text{if } k \text{ is even.} \end{array}$$

The above recurrence is in (almost) closed form:

$$T(k) = \begin{cases} 2^k - 1 & \text{if } k \geq 1 \text{ is odd,} \\ 2^k - 2 & \text{if } k \geq 1 \text{ is even,} \end{cases}$$

as can be proven by induction on k (we omit the trivial induction base): For odd k (i.e., $k = 2n - 1$ for $n \geq 1$), assume $T(2n-1) = 2^{2n-1} - 1$ to be true. Then

$$T(2n+1) = 2T(2n) + 3 = 4T(2n-1) + 3 \stackrel{\text{hyp.}}{=} 4\left(2^{2n-1} - 1\right) + 3 = 2^{2n+1} - 1.$$

For even k (i.e., $k = 2n$ for $n \geq 1$), assume $T(2n) = 2^{2n} - 2$ to be true. Then

$$T(2n+2) = 2T(2n+1) = 2(2T(2n) + 3) \stackrel{\text{hyp.}}{=} 4\left(2^{2n} - 2\right) + 6 = 2^{2n+2} - 2. \qquad \square$$

Remark 3.8. The upper bound in the second part of the above proof can be slightly improved using the fact that $\Sigma^* \,\Delta\, \Sigma^* \,\Delta\, A = \emptyset \,\Delta\, A = A$ for any set A. This gives the recurrence

$$T(1) = 1 \quad \text{and} \quad T(k) = \begin{cases} 2T(k-1) + 1 & \text{if } k > 1 \text{ is odd,} \\ 2T(k-1) & \text{if } k > 1 \text{ is even,} \end{cases}$$

or, equivalently, $T(1) = 1$, $T(2) = 2$, and $T(k) = 2^{k-1} + T(k-2)$ for $k \geq 3$. Though this shows that the upper bound given in the above proof is not optimal, the new bound is not a strong improvement, since it still embeds $\mathrm{C}_k(\mathrm{UP})$ in an exponentially higher level of $\mathrm{SDH}(\mathrm{UP})$. We propose as an interesting task the establishment of *tight* level-wise containments, at least up to the limits of relativizing techniques, between the hierarchies $\mathrm{SDH}(\mathrm{UP})$ and $\mathrm{CH}(\mathrm{UP})$, both of which capture the Boolean closure of UP.

We conjecture that there is some relativized world in which an exponential increase (though less dramatic than the particular exponential increase of Corollary 3.7) indeed is necessary.

Theorem 3.9 below shows that each level of the nested difference hierarchy is contained in the same level of both the C and the E hierarchy. Surprisingly, it turns out (see Theorem 3.13 below) that, relative to a recursive oracle, even the fourth level of CH(UP) and the third level of SDH(UP) are not subsumed by any level of the EH(UP) hierarchy. Consequently, neither the D nor the E normal forms of Definition 3.1 capture the Boolean closure of UP.

THEOREM 3.9. *For every* $k \geq 1$, $\mathrm{D}_k(\mathrm{UP}) \subseteq \mathrm{C}_k(\mathrm{UP}) \cap \mathrm{E}_k(\mathrm{UP})$.

Proof. For the first inclusion, by [11, Proposition 2.1.2], each set L in $\mathrm{D}_k(\mathrm{UP})$ can be represented as

$$L = A_1 - (A_2 - (\cdots (A_{k-1} - A_k) \cdots)),$$

where $A_i = \bigcap_{1 \leq j \leq i} L_j$, $1 \leq i \leq k$, and the L_j's are the original UP sets representing L. Note that since the proof of [11, Proposition 2.1.2] only uses intersection, the sets A_i are in UP. A special case of [11, Proposition 2.1.3] says that sets in $\mathrm{D}_k(\mathrm{UP})$ via decreasing chains such as the A_i are in $\mathrm{C}_k(\mathrm{UP})$, and so $L \in \mathrm{C}_k(\mathrm{UP})$.

The proof of the second inclusion is done by induction on the odd and even levels separately. The induction base follows by definition in either case. For odd levels, assume $\mathrm{D}_{2n-1}(\mathrm{UP}) \subseteq \mathrm{E}_{2n-1}(\mathrm{UP})$ to be valid, and let L be any set in $\mathrm{D}_{2n+1}(\mathrm{UP}) = \mathrm{UP} - (\mathrm{UP} - \mathrm{D}_{2n-1}(\mathrm{UP}))$. By our inductive hypothesis, L can be represented as

$$L = A - \left(B - \left(\bigcup_{i=1}^{n-1} \left(C_i \cap \overline{D_i} \right) \cup E \right) \right),$$

where A, B, C_i, D_i, and E are sets in UP. Thus

$$\begin{aligned} L &= A \cap \left(\overline{B \cap \left(\overline{\bigcup_{i=1}^{n-1} \left(C_i \cap \overline{D_i} \right) \cup E} \right)} \right) \\ &= A \cap \left(\overline{B} \cup \left(\bigcup_{i=1}^{n-1} \left(C_i \cap \overline{D_i} \right) \cup E \right) \right) \\ &= (A \cap \overline{B}) \cup \left(\bigcup_{i=1}^{n-1} A \cap C_i \cap \overline{D_i} \right) \cup (A \cap E) \\ &= \left(\bigcup_{i=1}^{n} F_i \cap \overline{D_i} \right) \cup G, \end{aligned}$$

where $F_i = A \cap C_i$, for $1 \leq i \leq n-1$, $F_n = A$, $D_n = B$, and $G = A \cap E$. Since UP is closed under intersection, each of these sets is in UP. Thus $L \in \mathrm{E}_{2n+1}(\mathrm{UP})$. The proof for the even levels is analogous except that the set E is dropped. □

Note that most of the above proofs used only the facts that the class is closed under intersection and contains Σ^* and $\emptyset$.

THEOREM 3.10. *Theorems* 3.4, 3.5, *and* 3.9 *and Corollaries* 3.6 *and* 3.7 *apply to all classes that contain* Σ^* *and* $\emptyset$ *and are closed under intersection.*

Remark 3.11. Although DP is closed under intersection but seems to lack closure under union (unless the polynomial hierarchy collapses to DP [37, 15, 14]) and thus

Theorem 3.10 in particular applies to DP, we note that the known results about Boolean hierarchies over NP [9, 41] in fact even for the DP case imply stronger results than those given by our Theorem 3.10, due to the very special structure of DP. Indeed, since, e.g., $\mathrm{E}_k(\mathrm{DP}) = \mathrm{E}_{2k}(\mathrm{NP})$ for any $k \geq 1$ (and the same holds for the other hierarchies), it follows immediately that all the level-wise equivalences among the Boolean hierarchies (and also their ability to capture the Boolean closure) that are known to hold for NP also hold for DP even in the absence of the assumption of closure under union. This appears to contrast with the UP case (see Remark 3.8).

The following combinatorial lemma will be useful in proving Theorem 3.13.

LEMMA 3.12. [12] *Let $G = (S, T, E)$ be any directed bipartite graph with out-degree bounded by d for all vertices. Let $S' \subseteq S$ and $T' \subseteq T$ be subsets such that $S' \supseteq \{s \in S \mid (\exists t \in T)\,[\langle s, t\rangle \in E]\}$, and $T' \supseteq \{t \in T \mid (\exists s \in S)\,[\langle t, s\rangle \in E]\}$. Then either*

1. $\|S'\| \leq 2d$, *or*
2. $\|T'\| \leq 2d$, *or*
3. $(\exists s \in S')\,(\exists t \in T')\,[\langle s, t\rangle \notin E \wedge \langle t, s\rangle \notin E]$.

For papers concerned with oracles separating internal levels of Boolean hierarchies over classes other than those of this paper, we refer the reader to [9, 8, 20, 7, 18] (see also [21]). Theorem 3.13 is optimal since clearly $\mathrm{C}_3(\mathrm{UP}) \subseteq \mathrm{EH}(\mathrm{UP})$ and $\mathrm{SD}_2(\mathrm{UP}) \subseteq \mathrm{EH}(\mathrm{UP})$, and both these containments relativize.

THEOREM 3.13. *There are recursive oracles A and D (though we may take $A = D$) such that*

1. $\mathrm{C}_4(\mathrm{UP}^A) \not\subseteq \mathrm{EH}(\mathrm{UP}^A)$ *and*
2. $\mathrm{SD}_3(\mathrm{UP}^D) \not\subseteq \mathrm{EH}(\mathrm{UP}^D)$.

COROLLARY 3.14. *There is a recursive oracle A such that*

1. $\mathrm{EH}(\mathrm{UP}^A) \neq \mathrm{BC}(\mathrm{UP}^A)$ *and* $\mathrm{DH}(\mathrm{UP}^A) \neq \mathrm{BC}(\mathrm{UP}^A)$, [3] *and*
2. $\mathrm{EH}(\mathrm{UP}^A)$ *and* $\mathrm{DH}(\mathrm{UP}^A)$ *are not closed under all Boolean operations.*

Proof of Theorem 3.13. Although the theorem claims that there is an oracle keeping $\mathrm{C}_4(\mathrm{UP})$ from being contained in any level of $\mathrm{EH}(\mathrm{UP})$, we will only prove that for any fixed k we can ensure that $\mathrm{C}_4(\mathrm{UP})$ is not contained in $\mathrm{E}_k(\mathrm{UP})$, relative to some oracle $A^{(k)}$. In the standard way, by interleaving diagonalizations, the sequence of oracles, $A^{(k)}$, can be combined into a single oracle, A, that fulfills the claim of the theorem. An analogous comment holds for the second claim of the theorem, with a sequence of oracles $D^{(k)}$ yielding a single oracle D. Similarly, both statements of the theorem can be satisfied simultaneously via just one oracle, via interleaving with each other the constructions of A and D. Though below we construct just $A^{(k)}$ and $D^{(k)}$, as a notational shorthand we will use A and D below to represent $A^{(k)}$ and $D^{(k)}$.

Before the actual construction of the oracles, we state some preliminaries that apply to the proofs of both statements in the theorem.

For any $n \geq 0$ and any string $v \in \Sigma^{\leq n}$, define $S^n_v \stackrel{\mathrm{df}}{=} \{vw \mid vw \in \Sigma^n\}$. The sets S^n_v are used to distinguish between different segments of Σ^n in the definition of the test languages, L_A and L_D.

Fix any standard enumeration of all NPOMs. Fix any $k > 0$. We need only consider even levels of $\mathrm{EH}(\mathrm{UP})$ since each odd level is contained in some even level. Call any collection of $2k$ NPOMs, $H = \langle N_{1,1}, \ldots, N_{k,1}, N_{1,2}, \ldots, N_{k,2}\rangle$, a potential

[3] Since both Corollary 3.3 (establishing $\mathrm{DH}(\mathrm{UP}) = \mathrm{CH}(\mathrm{coUP})$) and Theorem 3.5 ($\mathrm{BC}(\mathrm{UP}) = \mathrm{CH}(\mathrm{UP})$) relativize, this oracle A also separates the Boolean (alternating sums) hierarchy over coUP from the fourth level of the same hierarchy over UP and thus from BC(UP).

(relativized) $\mathrm{E}_{2k}(\mathrm{UP})$ machine, and for any oracle X, define its language to be:

$$L(H^X) \stackrel{\mathrm{df}}{=} \bigcup_{i=1}^{k} \left(L(N_{i,1}^X) - L(N_{i,2}^X)\right).$$

If for some fixed oracle Y, a potential (relativized) $\mathrm{E}_{2k}(\mathrm{UP})$ machine H^Y has the property that each of its underlying NPOMs with oracle Y is unambiguous, then $L(H^Y)$ indeed is in $\mathrm{E}_{2k}(\mathrm{UP}^Y)$. Clearly, our enumeration of all NPOMs induces an enumeration of all potential $\mathrm{E}_{2k}(\mathrm{UP})$ oracle machines. For $j \geq 1$, let H_j be the jth machine in this enumeration. Let p_j be a polynomial bounding the length of the computation paths of each of H_j's underlying machines (and thus bounding the number of and length of the strings they each query). As a notational convenience, we henceforth will use H and p as shorthands for H_j and p_j, and we will denote the underlying NPOMs by $N_{1,1}, \ldots, N_{k,1}, N_{1,2}, \ldots, N_{k,2}$.

The oracle X, where X stands for A or D, is constructed in stages, $X = \bigcup_{j \geq 1} X_j$. In stage j, we diagonalize against H by satisfying the following requirement R_j for every $j \geq 1$:

R_j: Either there is an $n > 2$ and an i, $1 \leq i \leq k$, such that one of $N_{i,1}^{X_j}$ or $N_{i,2}^{X_j}$ on input 0^n is ambiguous (thus H is in fact not an $\mathrm{E}_{2k}(\mathrm{UP})$ machine relative to X), or $L(H^X) \neq L_X$, where L_X is as defined below.

Let X_j be the set of strings contained in X by the end of stage j, and let X_j' be the set of strings forbidden membership in X during stage j. The restraint function $r(j)$ will satisfy the condition that at no later stage will strings of length smaller than $r(j)$ be added to X. Also, our construction will ensure that $r(j)$ is so large that X_{j-1} contains no strings of length greater than $r(j)$. Initially, both X_0 and X_0' are empty, and $r(1)$ is set to be 2.

We now start the proof of Part 1 of the theorem. Define the test language

$$L_A \stackrel{\mathrm{df}}{=} \{0^n \mid (\exists x)\,[x \in S_0^n \cap A] \wedge (\forall y)\,[y \notin S_{10}^n \cap A] \wedge (\forall z)\,[z \notin S_{11}^n \cap A]\}.$$

Clearly, L_A is in $\mathrm{NP}^A \wedge \mathrm{coNP}^A \wedge \mathrm{coNP}^A$. However, if we ensure in the construction that the invariant $\|S_v^n \cap A\| \leq 1$ is maintained for $v \in \{0, 10, 11\}$ and every $n \geq 2$, then L_A is even in $\mathrm{UP}^A \wedge \mathrm{coUP}^A \wedge \mathrm{coUP}^A$ and thus in $\mathrm{C}_4(\mathrm{UP}^A)$. We now describe stage $j > 0$ of the oracle construction.

Stage j: Choose $n > r(j)$ so large that $2^{n-2} > 3p(n)$.

Case 1: $0^n \in L(H^{A_{j-1}})$. Since $0^n \notin L_A$, we have $L(H^A) \neq L_A$.

Case 2: $0^n \notin L(H^{A_{j-1}})$. Choose some $x \in S_0^n$ and set $B_j := A_{j-1} \cup \{x\}$.

Case 2.1: $0^n \notin L(H^{B_j})$. Letting $A_j := B_j$ implies $0^n \in L_A$, so $L(H^A) \neq L_A$.

Case 2.2: $0^n \in L(H^{B_j})$. Then there is an i, $1 \leq i \leq k$, such that $0^n \in L(N_{i,1}^{B_j})$ and $0^n \notin L(N_{i,2}^{B_j})$. "Freeze" an accepting path of $N_{i,1}^{B_j}(0^n)$ into A_j'; that is, add those strings queried negatively on that path to A_j', thus forbidding them from A for all later stages. Clearly, at most $p(n)$ strings are "frozen."

Case 2.2.1: $(\exists z \in (S_{10}^n \cup S_{11}^n) - A_j')\,[0^n \notin L(N_{i,2}^{B_j \cup \{z\}})]$.
Choose any such z. Set $A_j := B_j \cup \{z\}$. We have $0^n \in L(H^A)$ but $0^n \notin L_A$.

Case 2.2.2: $(\forall z \in (S_{10}^n \cup S_{11}^n) - A_j')\,[0^n \in L(N_{i,2}^{B_j \cup \{z\}})]$.
To apply Lemma 3.12, define a directed bipartite graph $G =$

(S, T, E) by $S \stackrel{\text{df}}{=} S_{10}^n - A_j'$, $T \stackrel{\text{df}}{=} S_{11}^n - A_j'$, and for each $s \in S$ and $t \in T$, $\langle s, t\rangle \in E$ if and only if $N_{i,2}^{B_j \cup \{s\}}$ queries t along its lexicographically first accepting path, and $\langle t, s\rangle \in E$ is defined analogously. The out-degree of all vertices of G is bounded by $p(n)$. By our choice of n, $\min\{\|S\|, \|T\|\} \geq 2^{n-2} - p(n) > 2p(n)$, and thus alternative 3 of Lemma 3.12 applies. Hence there exist strings $s \in S$ and $t \in T$ such that $N_{i,2}^{B_j \cup \{s\}}(0^n)$ accepts on some path p_s on which t is not queried, and $N_{i,2}^{B_j \cup \{t\}}(0^n)$ accepts on some path p_t on which s is not queried. Since p_s (p_t) changes from reject to accept exactly by adding string s (t) to the oracle, s (t) must have been queried on p_s (p_t). We conclude that $p_s \neq p_t$, and thus $N_{i,2}^{B_j \cup \{s,t\}}(0^n)$ has at least two accepting paths. Set $A_j := B_j \cup \{s, t\}$.

In each case, requirement R_j is fulfilled. Let $r(j+1)$ be $\max\{n, w_j\}$, where w_j is the length of the largest string queried through stage j.

End of stage j.

We now turn to the proof of part 2 of the theorem. The test language here, L_D, is defined by:

$$L_D \stackrel{\text{df}}{=} \left\{ 0^n \left| \begin{array}{l} ((\exists x)\,[x \in S_0^n \cap D] \wedge (\exists y)\,[y \in S_{10}^n \cap D] \wedge (\exists z)\,[z \in S_{11}^n \cap D]) \vee \\ ((\forall x)\,[x \notin S_0^n \cap D] \wedge (\forall y)\,[y \notin S_{10}^n \cap D] \wedge (\exists z)\,[z \in S_{11}^n \cap D]) \vee \\ ((\exists x)\,[x \in S_0^n \cap D] \wedge (\forall y)\,[y \notin S_{10}^n \cap D] \wedge (\forall z)\,[z \notin S_{11}^n \cap D]) \vee \\ ((\forall x)\,[x \notin S_0^n \cap D] \wedge (\exists y)\,[y \in S_{10}^n \cap D] \wedge (\forall z)\,[z \notin S_{11}^n \cap D]) \end{array} \right. \right\}.$$

Again, provided that the invariant $\|S_v^n \cap D\| \leq 1$ is maintained for $v \in \{0, 10, 11\}$ and every $n \geq 2$ throughout the construction, L_D is clearly in $\mathrm{SD}_3(\mathrm{UP}^D)$, as for all sets A, B, and C,

$$A \,\Delta\, B \,\Delta\, C = (A \cap B \cap C) \cup (\overline{A} \cap \overline{B} \cap C) \cup (A \cap \overline{B} \cap \overline{C}) \cup (\overline{A} \cap B \cap \overline{C}).$$

Stage $j > 0$ of the construction of D is as follows.

Stage j: Choose $n > r(j)$ so large that $2^{n-2} > 3p(n)$.

Case 1: $0^n \in L(H^{D_{j-1}})$. Since $0^n \notin L_D$, we have $L(H^D) \neq L_D$.

Case 2: $0^n \notin L(H^{D_{j-1}})$. Choose some $x \in S_0^n$ and set $E_j := D_{j-1} \cup \{x\}$.

Case 2.1: $0^n \notin L(H^{E_j})$. Letting $D_j := E_j$ implies $0^n \in L_D$, so $L(H^D) \neq L_D$.

Case 2.2: $0^n \in L(H^{E_j})$. Then there is an i, $1 \leq i \leq k$, such that $0^n \in L(N_{i,1}^{E_j})$ and $0^n \notin L(N_{i,2}^{E_j})$. "Freeze" an accepting path of $N_{i,1}^{E_j}(0^n)$ into D_j'. Again, at most $p(n)$ strings are "frozen."

Case 2.2.1: $(\exists w \in (S_{10}^n \cup S_{11}^n) - D_j')\,[0^n \notin L(N_{i,2}^{E_j \cup \{w\}})]$.

Choose any such w and set $D_j := E_j \cup \{w\}$. We have $0^n \in L(H^D)$ but $0^n \notin L_D$.

Case 2.2.2: $(\forall w \in (S_{10}^n \cup S_{11}^n) - D_j')\,[0^n \in L(N_{i,2}^{E_j \cup \{w\}})]$.

As before, Lemma 3.12 yields two strings $s \in S_{10}^n - D_j'$ and $t \in S_{11}^n - D_j'$ such that $N_{i,2}^{E_j \cup \{s,t\}}(0^n)$ is ambiguous. Set $D_j := E_j \cup \{s, t\}$.

Again, R_j is always fulfilled. Define $r(j+1)$ as before.

End of stage j. □

Finally, we note that a slight modification of the above proof establishes the analogous result (of Theorem 3.13) for the case of US [6] (which is denoted 1NP in [21, 18]).

4. Sparse Turing-complete and Turing-hard sets for UP. In this section, we show some consequences of the existence of sparse Turing-complete and Turing-hard sets for UP. This question has been carefully investigated for the class NP [39, 34, 40, 1, 45, 54, 38].[4] Kadin showed that if there is a sparse $\leq_T^p$-complete set in NP, then the polynomial hierarchy collapses to $\mathrm{P}^{\mathrm{NP[log]}}$ [38]. Due to the promise nature of UP (in particular, UP probably lacks complete sets [22]), Kadin's proof does not seem to apply here. But does the existence of a sparse Turing-complete set in UP cause at least some collapse of the unambiguous polynomial hierarchy (which was introduced recently in [48])?[5]

Cai, Hemachandra, and Vyskoč [12] observe that ordinary Turing access to UP, as formalized by P^{UP}, may be too restrictive a notion to capture adequately one's intuition of Turing access to unambiguous computation since in that model the oracle machine has to be unambiguous on *every* input—even those the base DPOM never asks (on any of *its* inputs). To relax that unnaturally strong uniformity requirement, they introduce the class denoted $\mathrm{P}^{\mathcal{UP}}$, in which NP oracles are accessed in a *guardedly* unambiguous manner, a natural notion of access to unambiguous computation—suggested in the rather analogous case of NP $\cap$ coNP by Grollmann and Selman [19]—in which *only computations actually executed need be unambiguous.* Lange, Niedermeier, and Rossmanith [43], [48, p. 482] generalize this approach to build up an entire hierarchy of unambiguous computations in which the oracle levels are guardedly accessed (Definition 4.1, part 3)—the *promise unambiguous polynomial hierarchy.*

Definition 4.1.

1. *The* polynomial hierarchy [47, 57] *is defined as follows:*
$\Sigma_0^p \stackrel{\mathrm{df}}{=} \mathrm{P}$, $\Delta_0^p \stackrel{\mathrm{df}}{=} \mathrm{P}$, $\Sigma_k^p \stackrel{\mathrm{df}}{=} \mathrm{NP}^{\Sigma_{k-1}^p}$, $\Pi_k^p \stackrel{\mathrm{df}}{=} \mathrm{co}\Sigma_k^p$, $\Delta_k^p \stackrel{\mathrm{df}}{=} \mathrm{P}^{\Sigma_{k-1}^p}$, $k \geq 1$, *and* $\mathrm{PH} \stackrel{\mathrm{df}}{=} \bigcup_{k\geq 0} \Sigma_k^p$.

2. *The* unambiguous polynomial hierarchy [48] *is defined as follows:*
$\mathrm{U}\Sigma_0^p \stackrel{\mathrm{df}}{=} \mathrm{P}$, $\mathrm{U}\Delta_0^p \stackrel{\mathrm{df}}{=} \mathrm{P}$, $\mathrm{U}\Sigma_k^p \stackrel{\mathrm{df}}{=} \mathrm{UP}^{\mathrm{U}\Sigma_{k-1}^p}$, $\mathrm{U}\Pi_k^p \stackrel{\mathrm{df}}{=} \mathrm{coU}\Sigma_k^p$, $\mathrm{U}\Delta_k^p \stackrel{\mathrm{df}}{=} \mathrm{P}^{\mathrm{U}\Sigma_{k-1}^p}$, $k \geq 1$, *and* $\mathrm{UPH} \stackrel{\mathrm{df}}{=} \bigcup_{k\geq 0} \mathrm{U}\Sigma_k^p$.

3. *The* promise unambiguous polynomial hierarchy [43], [48, p. 482] *is defined as follows:* $\mathcal{U}\Sigma_0^p \stackrel{\mathrm{df}}{=} \mathrm{P}$, $\mathcal{U}\Sigma_1^p \stackrel{\mathrm{df}}{=} \mathrm{UP}$, *and for* $k \geq 2$, $L \in \mathcal{U}\Sigma_k^p$ *if and only if* $L \in \Sigma_k^p$ *via NPOMs* $N_1, \ldots, N_k$ *satisfying for all inputs* x *and every* i, $1 \leq i \leq k-1$, *that if* N_i *asks some query* q *during the computation of* $N_1(x)$, *then* $N_{i+1}(q)$ *with oracle* $L(N_{i+2}^{L(N_{i+3}^{\cdot^{\cdot^{\cdot^{L(N_k)}}}})})$ *has at most one accepting path.* $\mathcal{UPH} \stackrel{\mathrm{df}}{=} \bigcup_{k\geq 0} \mathcal{U}\Sigma_k^p$. *The classes* $\mathcal{U}\Delta_k^p$ *and* $\mathcal{U}\Pi_k^p$, $k \geq 0$, *are defined analogously. As a notational shorthand, we often use* $\mathrm{P}^{\mathcal{UP}}$ *to represent* $\mathcal{U}\Delta_2^p$; *we stress that both notations are used here to represent the class of sets accepted via* guardedly unambiguous *access to an NP oracle* (*that is, the class of sets accepted by some P machine with an NP machine's language as its*

[4] For reductions less flexible than Turing reductions (e.g., $\leq_m^p$, $\leq_{btt}^p$, etc.), this issue has been studied even more intensely (see, e.g., the surveys [61, 28]).

[5] Note that it is not known whether such a collapse implies a collapse of PH. Note also that Toda's [58] result on whether P-selective sets can be truth-table hard for UP does not imply such a collapse since truth-table reductions are less flexible than Turing reductions.

oracle such that on no input does the P machine ask its oracle machine any question on which the oracle machine has more than one accepting path).

4. *For each of the above hierarchies, we use* $\Sigma_k^{p,A}$ (*respectively,* $\mathrm{U}\Sigma_k^{p,A}$ *and* $\mathcal{U}\Sigma_k^{p,A}$) *to denote that the* Σ_k^p (*respectively,* $\mathrm{U}\Sigma_k^p$ *and* $\mathcal{U}\Sigma_k^p$) *computation is performed relative to oracle* A; *similar notation is used for the* Π *and* Δ *classes of the hierarchies.*

The following facts follow from the definition (see also [48]) or can easily be shown.

FACT 4.2. *For every* $k \geq 1$, *the following hold:*

1. $\mathrm{U}\Sigma_k^p \subseteq \mathcal{U}\Sigma_k^p \subseteq \Sigma_k^p$ *and* $\mathrm{U}\Delta_k^p \subseteq \mathcal{U}\Delta_k^p \subseteq \Delta_k^p$.
2. *If* $\mathrm{U}\Sigma_k^p = \mathrm{U}\Pi_k^p$, *then* $\mathrm{UPH} = \mathrm{U}\Sigma_k^p$.
3. *If* $\mathrm{U}\Sigma_k^p = \mathrm{U}\Sigma_{k-1}^p$, *then* $\mathrm{UPH} = \mathrm{U}\Sigma_{k-1}^p$.
4. $\mathrm{U}\Sigma_k^{p,\mathrm{UP}\cap\mathrm{coUP}} = \mathrm{U}\Sigma_k^p$ *and* $\mathrm{P}^{\mathrm{U}\Sigma_k^p\cap\mathrm{U}\Pi_k^p} = \mathrm{U}\Sigma_k^p \cap \mathrm{U}\Pi_k^p$.

The classes "$\mathrm{UP}_{\leq k}$," the analogues of UP in which up to k accepting paths are allowed, have been studied in various contexts [60, 27, 2, 12, 30, 33]. One motivation for $\mathrm{U}\Sigma_k^p$ is that, for each k, $\mathrm{UP}_{\leq k} \subseteq \mathrm{U}\Sigma_k^p$ [48].

Although we are not able to settle affirmatively the question posed at the end of the first paragraph of this section, we do prove in the theorem below that if there is a sparse Turing-complete set for UP, then the levels of the unambiguous polynomial hierarchy are simpler than one would otherwise expect: they "slip down" slightly in terms of their location within the promise unambiguous polynomial hierarchy, i.e., for each $k \geq 3$, the kth level of UPH is contained in the $(k-1)$st level of $\mathcal{UPH}$.

THEOREM 4.3. *If there exists a sparse Turing-complete set for* UP, *then*

1. $\mathrm{UP}^{\mathrm{UP}} \subseteq \mathrm{P}^{\mathcal{UP}}$ *and*
2. $\mathrm{U}\Sigma_k^p \subseteq \mathcal{U}\Sigma_{k-1}^p$ *for every* $k \geq 3$.

Proof. For the first statement, let L be any set in $\mathrm{UP}^{\mathrm{UP}}$. By assumption, L is in $\mathrm{UP}^{\mathrm{P}^S} = \mathrm{UP}^S$ for some sparse set $S \in \mathrm{UP}$. Let q be a polynomial bounding the density of S, that is, $\|S^{\leq m}\| \leq q(m)$ for every $m \geq 0$, and let N_S be a UPM for S. Let N_L be a UPOM witnessing that $L \in \mathrm{UP}^S$, that is, $L = L(N_L^S)$. Let $p(n)$ be a polynomial bounding the length of all query strings that can be asked during the computation of N_L on inputs of length n. Define the polynomial $r(n) \stackrel{\mathrm{df}}{=} q(p(n))$ that bounds the number of strings in S that can be queried in the run of N_L on inputs of length n.

To show that $L \in \mathrm{P}^{\mathcal{UP}}$, we shall construct a DPOM M that may access its $\mathcal{UP}$ oracle D in a guarded manner (more formally, "may access its NP oracle D in a guardedly unambiguous manner," but we will henceforth use $\mathcal{UP}$ and other $\mathcal{U}\cdots$ notations in this informal manner). Before formally describing machine M (Figure 4.1), we give some informal explanations. M will proceed in three basic steps: First, M determines the exact census of that part of S that is relevant for the given input length, $\|S^{\leq p(n)}\|$. Knowing the exact census, M can construct (by prefix search) a table T of all strings in $S^{\leq p(n)}$ without asking queries that make its oracle's machine ambiguous, so the $\mathrm{P}^{\mathcal{UP}}$-like behavior is guaranteed. Finally, M asks its oracle D to simulate the computation of N_L on input x (answering N_L's oracle queries by table-lookup using table T), and accepts accordingly.

In the formal description of machine M (given in Figure 4.1), three oracle sets A, B, and C are used. Since M has only one $\mathcal{UP}$ oracle, the actual set to be used is $D = A \oplus B \oplus C$ (with suitably modified queries to D). A, B, and C are defined as follows (we assume the set T below is coded in some standard reasonable way):

$$A \stackrel{\mathrm{df}}{=} \left\{ \langle 1^n, k\rangle \;\middle|\; \begin{array}{l} n \geq 0 \wedge 0 \leq k \leq r(n) \wedge (\exists c_1 <_{\mathrm{lex}} c_2 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} c_k) \\ (\forall \ell : 1 \leq \ell \leq k)\,[|c_\ell| \leq p(n) \wedge N_S(c_\ell) \text{ accepts}\,] \end{array} \right\},$$

Description of DPOM M.
input x;
begin
$n := |x|$;
$k := r(n)$;
loop
if $\langle 1^n, k\rangle \in A$ **then exit loop**
else $k := k - 1$
end loop (* k is now the exact census of $S^{\leq p(n)}$ *)
$T := \emptyset$; (* T collects the strings of $S^{\leq p(n)}$ *)
for $j = 1$ **to** k **do**
$c_j := \epsilon$;
$i := 1$;
repeat
if $\langle 1^n, i, j, k, 0\rangle \in B$ **then** $c_j := c_j 0$; $i := i + 1$
else
if $\langle 1^n, i, j, k, 1\rangle \in B$ **then** $c_j := c_j 1$; $i := i + 1$
else $i := 0$ (* the lex. jth string of $S^{\leq p(n)}$ has no ith bit *)
until $i = 0$;
$T := T \cup \{c_j\}$
end for
if $\langle x, T\rangle \in C$ **then accept**
else reject
end
End of description of DPOM M.

FIG. 4.1. *DPOM M guardedly unambiguously accessing an* NP *oracle to accept a set in* $\mathrm{UP}^{\mathrm{UP}}$.

$$B \stackrel{\mathrm{df}}{=} \left\{ \langle 1^n, i, j, k, b\rangle \;\middle|\; \begin{array}{l} n \geq 0 \wedge 1 \leq j \leq k \wedge 0 \leq k \leq r(n) \wedge \\ (\exists c_1 <_{\mathrm{lex}} c_2 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} c_k)\,(\forall \ell : 1 \leq \ell \leq k) \\ [|c_\ell| \leq p(n) \wedge N_S(c_\ell) \text{ accepts} \wedge \text{the } i\text{th bit of } c_j \text{ is } b] \end{array} \right\},$$

$$C \stackrel{\mathrm{df}}{=} \{\langle x, T\rangle \mid \|T\| \leq r(|x|) \wedge N_L^T(x) \text{ accepts}\}.$$

It is easy to see that M runs deterministically in polynomial time. This proves that $L \in \mathrm{P}^{\mathcal{UP}}$.

In order to prove the second statement, let L be a set in $\mathrm{U}\Sigma_k^p$ for any fixed $k \geq 3$. By assumption, there exists a sparse set S in UP such that L is in $\mathrm{U}\Sigma_{k-1}^{p,\mathrm{P}^S} = \mathrm{U}\Sigma_{k-1}^{p,S}$; let $N_1, N_2, \ldots, N_{k-1}$ be the UPOMs that witness this fact, that is, $L = L(N_1^{L(N_2^{\cdot^{\cdot^{\cdot^{L(N_{k-1}^S)}}}})})$.

Now we describe the computation of a $\mathcal{U}\Sigma_{k-1}^p$ machine N recognizing L. As before, N on input x computes in $\mathrm{P}^{\mathcal{UP}}$ its table of advice strings, $T = S^{\leq p(|x|)}$, and then simulates the $\mathrm{U}\Sigma_{k-1}^{p,S}$ computation of $N_1^{L(N_2^{\cdot^{\cdot^{\cdot^{L(N_{k-1}^S)}}}})}(x)$ except with N_1, $N_2, \ldots, N_{k-1}$ modified as follows. If in the simulation some machine N_i, $1 \leq i \leq k-2$, consults its original oracle $L(N_{i+1}^{(\cdot)})$ about some string, say z, then the modified machine N_i' queries the modified machine at the next level, N_{i+1}', about the string $\langle z, T\rangle$ instead. Finally, the advice table T, which has been "passed up" in this manner,

is used to correctly answer all queries of N_{k-1}.

Note that N's oracle in this simulation, $L(N_2'^{L(N_3'^{\cdot^{\cdot^{\cdot^{L(N_{k-1}')}}}})})$, is not in general a $\mathrm{U}\Sigma_{k-2}^p$ set (and L is thus not in $\mathrm{U}\Sigma_{k-1}^p$ in general) since the above-described computation depends on the advice table T, and so, for some bad advice T, the unambiguity of the modified machines $N_1', N_2', \ldots, N_{k-1}'$ is no longer guaranteed. But since our base machine N is able to provide *correct* advice T, we have indeed shown that $L \in \mathcal{U}\Sigma_{k-1}^p$. □

In the above proof, the assumption that the sparse set S is in UP is needed to determine the exact census of S using the UPM for S. Let us now consider the weaker assumption that UP has only a Turing-*hard* sparse set. Karp and Lipton have shown that if there is a sparse Turing-hard set for NP, then the polynomial hierarchy collapses to its second level [39].[6] Hopcroft [34] dramatically simplified their proof, and Balcázar, Book, and Schöning [1, 54] generalized, as Theorem 4.6, the Karp–Lipton result; the general approach of Hopcroft and Balcázar, Book, and Schöning will be central to our upcoming proof of Theorem 4.7. Schöning's low hierarchy [53] gives a way of classifying the complexity of NP sets that seem to be neither in P nor NP-complete. Of particular interest to us is the class $\mathrm{Low}_2 \stackrel{\mathrm{df}}{=} \{A \mid A \in \mathrm{NP} \text{ and } \mathrm{NP}^{\mathrm{NP}^A} \subseteq \mathrm{NP}^{\mathrm{NP}}\}$. Note that for the special case $k = 0$, Theorem 4.6 below says that $\mathrm{Low}_2 \supseteq \mathrm{NP} \cap \mathrm{P/poly} \cap \{L \mid L \text{ is self-reducible}\}$.

DEFINITION 4.4. [46]

1. *A partial order* $<_{\mathrm{pwl}}$ *on* Σ^* *is* polynomially well-founded and length-related *if and only if* (a) *every strictly decreasing chain is finite and there is a polynomial* p *such that every finite* $<_{\mathrm{pwl}}$*-decreasing chain is shorter than* p *of the length of its maximum element, and* (b) $(\exists q : q \text{ polynomial})(\forall x, y \in \Sigma^*)\,[x <_{\mathrm{pwl}} y \implies |x| \leq q(|y|)]$.

2. *A set* A *is* self-reducible *if and only if there exist a polynomially well-founded and length-related order* $<_{\mathrm{pwl}}$ *on* Σ^* *and a DPOM* M *such that* $A = L(M^A)$ *and on any input* $x \in \Sigma^*$, M *queries only strings* y *with* $y <_{\mathrm{pwl}} x$.

LEMMA 4.5. [1] *Let* A *be a self-reducible set and let* M *witness* A*'s self-reducibility. For any set* B *and any* n, *if* $(L(M^B))^{\leq n} = B^{\leq n}$, *then* $A^{\leq n} = B^{\leq n}$.[7]

THEOREM 4.6. [1] *If* A *is a self-reducible set and there is a* $k \geq 0$ *and a sparse set* S *such that* $A \in \Sigma_k^{p,S}$, *then* $\Sigma_2^{p,A} \subseteq \Sigma_{k+2}^p$.

We now state and prove our results regarding sparse Turing-hard sets for UP.

THEOREM 4.7. *If there exists a sparse Turing-hard set for* UP, *then*

1. $\mathrm{UP} \subseteq \mathrm{Low}_2$ *and*
2. $\mathrm{U}\Sigma_k^p \subseteq \mathrm{U}\Sigma_j^{p,\Sigma_2^{p,\,\mathcal{U}\Sigma_{k-j-3}^p}} \cap \mathrm{P}^{\mathcal{U}\Sigma_{k-1}^p \oplus \Sigma_2^p}$ *for every* $k \geq 3$ *and* j, $0 \leq j \leq k-3$.

Proof. 1. Let $L \in \Sigma_2^{p,A}$, where $A \in \mathrm{UP}$ via UPM N_A and polynomial-time bound t. (We assume that each step is nondeterministic—one can require this, without loss of generality, while maintaining categoricity.) Our proof uses the well-known fact that the "left set" [55, 49] of any UP set is self-reducible and is in UP. More precisely, to apply Theorem 4.6 we would need A to be self-reducible. Although that cannot be assumed in general of an arbitrary UP set, the left set of A, i.e., the set of prefixes of

[6] Very recently, Köbler and Watanabe [42] have improved this collapse to $\mathrm{ZPP}^{\mathrm{NP}}$, and they have also obtained new consequences from the assumption that $\mathrm{UP} \subseteq (\mathrm{NP} \cap \mathrm{coNP})/\mathrm{poly}$, whereas we obtain different consequences from the assumption that $\mathrm{UP} \subseteq \mathrm{P/poly}$ (see [42] for the notations not defined in this footnote).

[7] A can be viewed as a "fixed point" of M.

Description of self-reducer M_{self} for B.
  input $\langle x, y\rangle$;
  begin
    if $|y| > t(|x|)$ **then reject**;
    if $N_A(x)$ accepts on path y **then accept**
    else
      if $\langle x, y0\rangle \in B$ or $\langle x, y1\rangle \in B$ **then accept**
      else reject
  end
End of description of self-reducer M_{self} for B.

FIG. 4.2. *A self-reducing machine for the left set of a* UP *set.*

Description of DPOM M_A.
  input x;
  begin
    $y := \epsilon$;
    while $|y| < t(|x|)$ **do**
      if $\langle x, y0\rangle \in B$ **then accept**
      else $y := y1$
    end while
    if $\langle x, y\rangle \in B$ **then accept**
    else reject
  end
End of description of DPOM M_A.

FIG. 4.3. *A Turing reduction from a* UP *set A to its left set B via prefix search.*

witnesses for elements in A defined by

$$B \stackrel{\text{df}}{=} \{\langle x, y\rangle \mid (\exists z)\,[|yz| = t(|x|) \wedge N_A(x) \text{ accepts on path } yz]\},$$

does have this property and is also in UP. A self-reducing machine M_{self} for B is given in Figure 4.2. Note that the queries asked in the self-reduction are strictly less than the input with respect to a polynomially well-founded and length-related partial order $<_{\text{pwl}}$ defined by the following: For fixed x and all strings $y_1, y_2 \in \Sigma^{\leq p(|x|)}$, $\langle x, y_1\rangle <_{\text{pwl}} \langle x, y_2\rangle$ if and only if y_2 is a prefix of y_1.

By assumption, since B is a UP set, $B \in \mathrm{P}^S$ for some sparse set S, so Theorem 4.6 with $k = 0$ applies to B. Furthermore, A is in P^B, via prefix search by DPOM M_A (Figure 4.3). Thus $L \in \Sigma_2^{p,\mathrm{P}^B} \subseteq \Sigma_2^{p,B} \subseteq \Sigma_2^p$, which shows that $A \in \text{Low}_2$.

2. For $k = 3$ (thus $j = 0$), both inclusions have already been shown in part 1, as $\Sigma_2^p \subseteq \Delta_3^p$. Now fix any $k > 3$, and let $L \in \mathrm{U}\Sigma_k^p = \mathrm{U}\Sigma_{k-1}^{p,A}$ be witnessed by UPOMs $N_1, N_2, \ldots, N_{k-1}$ and $A \in \mathrm{UP}$. Define B to be the left set of A as in part 1, so $A \in \mathrm{P}^B$ via DPOM M_A (see Figure 4.3), B is self-reducible via M_{self} (see Figure 4.2), and B is in UP. By hypothesis, $B \in \mathrm{P}^S$ for some sparse set S; let M_B be the reducing machine, that is $B = L(M_B^S)$, and let m be a polynomial bound on the runtime of M_B. Let q

be a polynomial such that $\|S^{\leq m}\| \leq q(m)$ for every $m \geq 0$. Let $p(n)$ be a polynomial bounding the length of all query strings whose membership in the oracle set B can be asked in the run of N_1 (with oracle machines N_2, $N_3, \ldots, N_{k-1}$, M_A^B) on inputs of length n. Define the polynomials $r(n) \stackrel{\text{df}}{=} m(p(n))$ and $s(n) \stackrel{\text{df}}{=} q(r(n))$.

To show that $L \in \mathrm{P}^{\mathcal{U}\Sigma_{k-1}^p \oplus \Sigma_2^p}$, we will describe a DPOM M that on input x, $|x| = n$, using the Σ_2^p part D (defined below) of its oracle, performs a prefix search to extract the lexicographically smallest of all "good" advice sets (this informal term will be formally defined in the next paragraph), say T, and then calls the $\mathcal{U}\Sigma_{k-1}^p$ part of its oracle to simulate the $\mathrm{U}\Sigma_{k-1}^{p,A}$ computation of $N_1^{L(N_2^{\cdot^{L(N_{k-1}^A)}})}(x)$ except with N_1, N_2, $\ldots, N_{k-1}$ modified in the same way as was described in the proof of Theorem 4.3. In more detail, if in the simulation some machine N_i, $1 \leq i \leq k-2$, consults its original oracle $L(N_{i+1}^{(\cdot)})$ about some string, say z, then the modified machine N_i' queries the modified machine at the next level, N_{i+1}', about the string $\langle z, T\rangle$ instead. Finally, if N_{k-1} consults its original oracle A about some query y, then the modified machine N_{k-1}' runs the P computation $M_A^{L(M_B^T)}$ on input $\langle y, T\rangle$ instead to correctly answer this query without consulting an oracle.

An advice set T is said to be *good* if the set $L(M_B^T)$ is a fixed point of B's self-reducer M_{self} up to length $p(n)$, that is, $(L(M_{\text{self}}^{L(M_B^T)}))^{\leq p(n)} = (L(M_B^T))^{\leq p(n)}$, and thus $B^{\leq p(n)} = (L(M_B^T))^{\leq p(n)}$ by Lemma 4.5. This property is checked for each guessed T in the Σ_2^p part of the oracle. Formally,

$$D \stackrel{\text{df}}{=} \left\{ \langle 1^n, i, j, b\rangle \;\middle|\; \begin{array}{l} n \geq 0 \wedge (\exists T \subseteq \Sigma^{\leq r(n)})\,(\forall w : |w| \leq p(n)\,)\,[T = \{c_1, \ldots, c_k\} \\ \wedge\ 0 \leq k \leq s(n) \wedge c_1 <_{\text{lex}} \cdots <_{\text{lex}} c_k \wedge \text{the } i\text{th bit of } c_j \text{ is } b \\ \wedge\ (w \in L(M_B^T) \iff w \in L(M_{\text{self}}^{L(M_B^T)}))] \end{array} \right\}.$$

The prefix search of M is similar to the one performed in the proof of Theorem 4.3 (see Figure 4.1); M queries D to construct each string of T bit by bit.

To prove the other inclusion, fix any j, $0 \leq j \leq k-3$. We describe a UPOM N witnessing that $L \in \mathrm{U}\Sigma_j^{p,\Sigma_2^{p,\,\mathcal{U}\Sigma_{k-j-3}^p}}$. On input x, N simulates the $\mathrm{U}\Sigma_j^p$ computation of the first j UPOMs $N_1, \ldots, N_j$. In the subsequent Σ_2^p computation, two tasks have to be solved in parallel: the computation of N_{j+1} and N_{j+2} is to be simulated, and good advice sets T have to be determined. For the latter task, the base machine of the Σ_2^p computation guesses all possible advice sets and the top machine checks if the guessed advice is good (that is, if $L(M_B^T)$ is a fixed point of M_{self}). Again, each good advice set T is "passed up" to the machines at higher levels $N_{j+3}, \ldots, N_{k-1}$ (in the same fashion as was employed earlier in this proof and also in the proof of Theorem 4.3) and is used to correctly answer all queries of N_{k-1} without consulting an oracle. This proves the theorem. □

Since Theorem 4.7 relativizes and there are relativized worlds in which UP^A is not Low_2^A [56], we have the following corollary.

COROLLARY 4.8. *There is a relativized world in which* (*relativized*) UP *has no sparse Turing-hard sets.*

Acknowledgments. We are very grateful to Gerd Wechsung for his help in bringing about this collaboration, and for his kind and insightful advice over many years. We thank Marius Zimand for proofreading and Nikolai Vereshchagin for helpful discussions during his visit to Rochester. We thank Osamu Watanabe for discussing

with us his results joint with Johannes Köbler, and we thank Osamu Watanabe and Johannes Köbler for providing us with copies of their paper [42].

REFERENCES

[1] J. Balcázar, R. Book, and U. Schöning, *The polynomial-time hierarchy and sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 603–617.

[2] R. Beigel, *On the relativized power of additional accepting paths*, in Proc. 4th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 216–224.

[3] R. Beigel, R. Chang, and M. Ogiwara, *A relationship between difference hierarchies and relativized polynomial hierarchies*, Math. Systems Theory, 26 (1993), pp. 293–310.

[4] R. Beigel and J. Goldsmith, *Downward separation fails catastrophically for limited nondeterminism classes*, in Proc. 9th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 134–138.

[5] A. Bertoni, D. Bruschi, D. Joseph, M. Sitharam, and P. Young, *Generalized Boolean hierarchies and Boolean hierarchies over RP*, in Proc. 7th Conference on Fundamentals of Computation Theory, Lecture Notes in Comput. Sci. 380, Springer-Verlag, Berlin, 1989, pp. 35–46.

[6] A. Blass and Y. Gurevich, *On the unique satisfiability problem*, Inform. and Control, 55 (1982), pp. 80–88.

[7] D. Bruschi, D. Joseph, and P. Young, *Strong separations for the Boolean hierarchy over RP*, Internat. J. Found. Comput. Sci., 1 (1990), pp. 201–218.

[8] J. Cai, *Probability one separation of the Boolean hierarchy*, in Proc. 4th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 247, Springer-Verlag, Berlin, 1987, pp. 148–158.

[9] J. Cai, T. Gundermann, J. Hartmanis, L. Hemachandra, V. Sewelson, K. Wagner, and G. Wechsung, *The Boolean hierarchy* I: *Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[10] J. Cai, T. Gundermann, J. Hartmanis, L. Hemachandra, V. Sewelson, K. Wagner, and G. Wechsung, *The Boolean hierarchy* II: *Applications*, SIAM J. Comput., 18 (1989), pp. 95–111.

[11] J. Cai and L. Hemachandra, *The Boolean hierarchy: Hardware over NP*, Technical Report 85-724, Department of Computer Science, Cornell University, Ithaca, NY, 1985.

[12] J. Cai, L. Hemachandra, and J. Vyskoč, *Promises and fault-tolerant database access*, in Complexity Theory, K. Ambos-Spies, S. Homer, and U. Schöning, eds., Cambridge University Press, Cambridge, UK, 1993, pp. 101–146.

[13] J. Cai and G. Meyer, *Graph minimal uncolorability is* D^{P}*-complete*, SIAM J. Comput., 16 (1987), pp. 259–277.

[14] R. Chang, *On the structure of NP computations under Boolean operators*, Ph.D. thesis, Cornell University, Ithaca, NY, 1991.

[15] R. Chang and J. Kadin, *On computing Boolean connectives of characteristic functions*, Technical Report TR 90-1118, Department of Computer Science, Cornell University, Ithaca, NY, 1990.

[16] R. Chang and J. Kadin, *The Boolean hierarchy and the polynomial hierarchy: A closer connection*, SIAM J. Comput., 25 (1996), pp. 340–354.

[17] S. Cook, *The complexity of theorem-proving procedures*, in Proc. 3rd ACM Symposium on Theory of Computing, ACM, New York, 1971, pp. 151–158.

[18] K. Cronauer, *A criterion to separate complexity classes by oracles*, Technical Report 76, Institut für Informatik, Universität Würzburg, Würzburg, Germany, 1994.

[19] J. Grollmann and A. Selman, *Complexity measures for public-key cryptosystems*, SIAM J. Comput., 17 (1988), pp. 309–335.

[20] T. Gundermann, N. Nasser, and G. Wechsung, *A survey on counting classes*, in Proc. 5th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 140–153.

[21] T. Gundermann and G. Wechsung, *Counting classes with finite acceptance types*, Comput. Artificial Intelligence, 6 (1987), pp. 395–409.

[22] J. Hartmanis and L. Hemachandra, *Complexity classes without machines: On complete languages for UP*, Theoret. Comput. Sci., 58 (1988), pp. 129–142.

[23] J. Hartmanis and L. Hemachandra, *Robust machines accept easy sets*, Theoret. Comput. Sci., 74 (1990), pp. 217–226.

[24] J. Hartmanis, N. Immerman, and V. Sewelson, *Sparse sets in NP−P: EXPTIME versus NEXPTIME*, Inform. and Control, 65 (1985), pp. 159–181.
[25] J. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.
[26] F. Hausdorff, *Grundzüge der Mengenlehre*, Walter De Gruyten & Co., Berlin, Leipzig, 1927.
[27] L. Hemachandra, *Counting in Structural Complexity Theory*, Ph.D. thesis, Technical Report TR87-840, Department of Computer Science, Cornell University, Ithaca, NY, 1987.
[28] L. Hemachandra, M. Ogiwara, and O. Watanabe, *How hard are sparse sets*?, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 222–238.
[29] L. Hemachandra and R. Rubinstein, *Separating complexity classes with tally oracles*, Theoret. Comput. Sci., 92 (1992), pp. 309–318.
[30] E. Hemaspaandra and L. Hemaspaandra, *Quasi-injective reductions*, Theoret. Comput. Sci., 123 (1994), pp. 407–413.
[31] L. Hemaspaandra, S. Jain, and N. Vereshchagin, *Banishing robust Turing completeness*, Internat. J. Found. Comput. Sci., 4 (1993), pp. 245–265.
[32] L. Hemaspaandra and S. Jha, *Defying upward and downward separation*, Inform. and Computation, 121 (1995), pp. 1–13.
[33] L. Hemaspaandra and M. Zimand, *Strong self-reducibility precludes strong immunity*, Math. Systems Theory, 29 (1996), pp. 535–548.
[34] J. Hopcroft, *Recent directions in algorithmic research*, in Proc. 5th GI Conference on Theoretical Computer Science, Lecture Notes in Comput. Sci. 104, Springer-Verlag, Berlin, 1981, pp. 123–134.
[35] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, MA, 1979.
[36] R. Impagliazzo and G. Tardos, *Decision versus search problems in super-polynomial time*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 222–227.
[37] J. Kadin, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263–1282; erratum, SIAM J. Comput., 20 (1991), p. 404.
[38] J. Kadin, $P^{NP[\log n]}$ *and sparse Turing-complete sets for NP*, J. Comput. System Sci., 39 (1989), pp. 282–298.
[39] R. Karp and R. Lipton, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th ACM Symposium on Theory of Computing, ACM, New York, 1980, pp. 302–309; an extended version has also appeared as *Turing machines that take advice*, Enseign. Math. (2), 28 (1982), pp. 191–209.
[40] K. Ko and U. Schöning, *On circuit-size complexity and the low hierarchy in NP*, SIAM J. Comput., 14 (1985), pp. 41–51.
[41] J. Köbler, U. Schöning, and K. Wagner, *The difference and truth-table hierarchies for NP*, RAIRO Inform. Théor. Appl., 21 (1987), pp. 419–435.
[42] J. Köbler and O. Watanabe, *New collapse consequences of NP having small circuits*, in Proc. 22nd International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 944, Springer-Verlag, Berlin, 1995, pp. 196–207.
[43] K.-J. Lange and P. Rossmanith, *Unambiguous polynomial hierarchies and exponential size*, in Proc. 9th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 106–115.
[44] L. Levin, *Universal sorting problems*, Problems Inform. Transmission, 9 (1973), pp. 265–266.
[45] T. Long and A. Selman, *Relativizing complexity classes with sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 618–627.
[46] A. Meyer and M. Paterson, *With what frequency are apparently intractable problems difficult*?, Technical Report MIT/LCS/TM-126, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1979.
[47] A. Meyer and L. Stockmeyer, *The equivalence problem for regular expressions with squaring requires exponential space*, in Proc. 13th IEEE Symposium on Switching and Automata Theory, IEEE Computer Society Press, Los Alamitos, CA, 1972, pp. 125–129.
[48] R. Niedermeier and P. Rossmanith, *Extended locally definable acceptance types*, in Proc. 10th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 665, Springer-Verlag, Berlin, 1993, pp. 473–483.
[49] M. Ogiwara and O. Watanabe, *On polynomial-time bounded truth-table reducibility of NP sets to sparse sets*, SIAM J. Comput., 20 (1991), pp. 471–483.
[50] C. Papadimitriou and M. Yannakakis, *The complexity of facets (and some facets of complexity)*, J. Comput. System Sci., 28 (1984), pp. 244–259.

[51] R. Rao, J. Rothe, and O. Watanabe, *Upward separation for FewP and related classes*, Inform. Process. Lett., 52 (1994), pp. 175–180.
[52] K. Regan, *Provable complexity properties and constructive reasoning*, manuscript, 1989.
[53] U. Schöning, *A low and a high hierarchy within NP*, J. Comput. System Sci., 27 (1983), pp. 14–28.
[54] U. Schöning, *Complexity and Structure*, Lecture Notes in Comput. Sci. 211, Springer-Verlag, 1986.
[55] L. Selman, *Natural self-reducible sets*, SIAM J. Comput., 17 (1988), pp. 989–996.
[56] M. Sheu and T. Long, *UP and the low and high hierarchies: A relativized separation*, Math. Systems Theory, 29 (1996), pp. 423–450.
[57] L. Stockmeyer, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.
[58] S. Toda, *On polynomial-time truth-table reducibilities of intractable sets to P-selective sets*, Math. Systems Theory, 24 (1991), pp. 69–82.
[59] L. Valiant, *The relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20–23.
[60] O. Watanabe, *On hardness of one-way functions*, Inform. Process. Lett., 27 (1988), pp. 151–157.
[61] P. Young, *How reductions to sparse sets collapse the polynomial-time hierarchy: A primer*, SIGACT News, 1992, #3, pp. 107–117 (part I), #4, pp. 83–94 (part II), and #4, p. 94 (corrigendum to part I).

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 654–674, June 1997

003

RAY SHOOTING AMIDST SPHERES IN THREE DIMENSIONS AND RELATED PROBLEMS*

SHAI MOHABAN† AND MICHA SHARIR‡

Abstract. We consider the problem of ray shooting amidst spheres in 3-space: given n arbitrary (possibly intersecting) spheres in 3-space and any $\varepsilon > 0$, we show how to preprocess the spheres in time $O(n^{3+\varepsilon})$ into a data structure of size $O(n^{3+\varepsilon})$ so that any ray-shooting query can be answered in time $O(n^{\varepsilon})$. Our result improves previous techniques (see [P. K. Agarwal, L. Guibas, M. Pellegrini, and M. Sharir, "Ray shooting amidst spheres," unpublished note] and [P. K. Agarwal and J. Matoušek, *Discrete Comput. Geom.*, 11 (1994), pp. 393–418]), where roughly $O(n^4)$ storage was required to support fast queries. Our result shows that ray shooting amidst spheres has complexity comparable with that of ray shooting amidst planes in 3-space. Our technique applies to more general (convex) objects in 3-space, and we also discuss these extensions.

Key words. computational geometry, ray shooting

AMS subject classifications. 52B11, 68P05, 68Q20, 68Q25

PII. S0097539793252080

1. Introduction.

The *ray shooting* problem can be defined as follows:

Given a collection $\mathcal{S}$ of n objects in $\mathbb{R}^d$ preprocess $\mathcal{S}$ into a data structure so that one can quickly determine the first object of $\mathcal{S}$ intersected by a query ray.

The ray-shooting problem has received considerable attention in the past few years because of its applications in computer graphics and other geometric problems [1, 4, 5, 6, 10, 11, 13, 16, 21]. Most of the work to date has studied the planar case, where $\mathcal{S}$ is a collection of line segments in $\mathbb{R}^2$. Chazelle and Guibas proposed an optimal algorithm for the special case where $\mathcal{S}$ is the boundary of a simple polygon [16]. Their algorithm answers a ray-shooting query in $O(\log n)$ time using $O(n)$ space; simpler algorithms with the same asymptotic performance bounds were recently developed in [13, 24]. If $\mathcal{S}$ is a collection of arbitrary segments in the plane, the best known algorithm answers a ray-shooting query in time $O((n/\sqrt{s})\log^{O(1)} n)$ using $O(s^{1+\varepsilon})$ space and preprocessing[1] [1, 6, 10], where s is a parameter that can vary between n and n^2. Although no lower bound is known for this case, it is conjectured that this bound is close to optimal.

In spite of some recent developments, the three-dimensional ray-shooting problem seems much harder, and it is still far from being fully solved. The general three-dimensional ray-shooting problem is to preprocess a collection $\mathcal{S}$ of n convex objects so that the first object hit by a query ray can be computed efficiently. Most studies of

* Received by the editors July 15, 1993; accepted for publication (in revised form) June 14, 1995. This research was supported by NSF grant CCR-91-22103 and grants from the U.S.–Israeli Binational Science Foundation, the German–Israeli Foundation for Scientific Research and Development (GIF), and the Fund for Basic Research administered by the Israeli Academy of Sciences. This paper is part of the first author's M.Sc. thesis, prepared under the supervision of the second author.

http://www.siam.org/journals/sicomp/26-3/25208.html

† School of Mathematical Sciences, Tel Aviv University, 69978 Tel Aviv, Israel (shai@math.tau.ac.il).

‡ School of Mathematical Sciences, Tel Aviv University, 69978 Tel Aviv, Israel and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012 (sharir@math.tau.ac.il).

[1] Throughout this paper, bounds of this kind mean that, given any arbitrarily small positive constant ε, the algorithm can be fine-tuned so that its performance satisfies the bound; the multiplicative constants in such bounds usually depend on ε and tend to ∞ as $\varepsilon \downarrow 0$.

this problem consider the case where $\mathcal{S}$ is a collection of triangles. If these triangles are the faces of a convex polyhedron, then an optimal algorithm with $O(\log n)$ query time and linear space can be obtained using the hierarchical-decomposition scheme of Dobkin and Kirkpatrick [19]. If the triangles form a *polyhedral terrain* (a piecewise-linear surface intersecting every vertical line in exactly one point), then the technique of Chazelle et al. [15] yields an algorithm that requires $O(n^{2+\varepsilon})$ space and answers ray-shooting queries in $O(\log n)$ time. Nontrivial solutions to the general problem (involving triangles) were obtained only recently; see [4, 6, 11] for some of these results. The best known algorithm for ray-shooting among triangles in three dimensions is due to Agarwal and Matoušek [5]; it answers a ray shooting query in time $O(n^{1+\varepsilon}/s^{1/4})$ with $O(s^{1+\varepsilon})$ space and preprocessing. The parameter s can range between n and n^4. If s assumes its maximum value, queries can be answered in $O(\log^2 n)$ time; see [5, 6] for more details. A variant of this technique was given recently in [7] for the case of ray shooting amidst a collection of convex polyhedra. We remark that no nontrivial lower bounds are known for the three-dimensional problem (for triangles) as well, although such bounds are known for the related *simplex range-searching* problem [12], which is used as a subprocedure in the solutions just mentioned. These bounds are close to $\Omega(n/s^{1/4})$ and thus might suggest that the known ray-shooting algorithms for triangles in $\mathbb{R}^3$ are close to optimal.

On the other hand, there are certain special cases of the three-dimensional ray-shooting problem which can be solved more efficiently. For example, if the objects of $\mathcal{S}$ are planes or half-planes, ray shooting amidst them can be performed in time $O(n^{1+\varepsilon}/s^{1/3})$ with $O(s^{1+\varepsilon})$ space and preprocessing; see [4] for details. It is therefore of interest to identify additional classes of objects for which ray shooting can also be performed more efficiently.

In this paper, we consider the case of spheres. This problem has recently been studied in [5] and in a yet unpublished work [3]. The algorithm presented in [5] has query time $O(n^{1+\varepsilon}/s^{1/4})$ with $O(s^{1+\varepsilon})$ space and preprocessing, that is, the same performance as in the case of triangles. The algorithm of [3] achieves similar performance for $s = n^4$.

We present an improved solution for the case where large storage is allowed. Specifically, we achieve query time $O(n^{\varepsilon})$ with only $O(n^{3+\varepsilon})$ space and preprocessing. Thus we show that in some sense, the case of spheres is essentially no harder than the case of planes, which might sound somewhat surprising given that spheres are more complex objects (and require four real parameters to specify, as opposed to only three for planes). Our solution is fairly general—the spheres may have arbitrary radii and may also intersect one another.

Our technique adapts some ideas from [3]. Roughly speaking, in both approaches, we reduce the ray-shooting problem to the following problem (although certain parts of our reduction are simpler than those of [3]). We are given a collection $\mathcal{S}$ of n arbitrary spheres in 3-space, $S_1, \ldots, S_n$, and we want to preprocess them so that we can answer the following queries efficiently: we are given a line ℓ such that for each sphere S_i, either ℓ intersects S_i or else S_i contains no point that lies vertically above some point on ℓ, and we want to determine whether ℓ intersects any of these spheres. The reduction of the ray-shooting problem to this problem is fairly routine using mostly standard machinery.

We solve this main subproblem in a more careful manner than in [3]. This subproblem can be mapped onto the four-dimensional parametric space that represents lines in 3-space, and it reduces to the problem of point location in the region of this 4-

space lying above the upper envelope of certain low-degree algebraic surface patches, each representing an "upper tangency" between a line and one of the spheres. Almost a year after the original submission of this paper, it was shown by Aronov et al. [2] that point location above such an envelope can be performed fast (in polylogarithmic time) using only $O(n^{3+\varepsilon})$ storage and $O(n^{3+\varepsilon})$ randomized expected time. This gives an alternative solution with roughly the same asymptotic performance as our solution. However, (a) the algorithm of [2] is randomized, whereas our solution is deterministic, and (b) our solution analyzes and exploits the geometric structure of the problem much more explicitly, and we hope that its analysis will find additional applications. Although our approach is more geometric in nature, it does not exploit any special properties of spheres and is thus much more general. As a matter of fact, our technique can be viewed as an explicit solution of the problem of point location above an envelope in 4-space for the special type of surfaces arising in the context of ray shooting. In section 3, we indeed present a generalization of our technique to ray shooting among more general convex objects in 3-space and exemplify it for the case of axis-parallel ellipsoids.

2. The algorithm.

2.1. Overview. We use the general approach to ray shooting due to Agarwal and Matoušek [4]. This approach, which is based on the parametric searching technique of Megiddo [27], reduces the ray-shooting problem to the *segment-emptiness* problem. Namely, we need to preprocess the spheres so that given any segment e in 3-space, we can quickly determine whether e intersects any of the spheres. As shown in [4], the time for the actual ray-shooting query is only within a logarithmic factor of the query time for the segment-emptiness problem.

We first describe the preprocessing scheme. We construct a multilevel data structure, where each level of the structure filters out spheres that satisfy a different geometric relationship with respect to the query segment, so a sphere satisfying the conjunction of all these relationships must intersect the query segment; see [26] for a more detailed discussion of multilevel data structures of this kind. Each level of the data structure consists of a tree-like structure $\mathcal{T}$, where each node of $\mathcal{T}$ corresponds to some "canonical" subset of spheres. The root of the top-level structure represents the entire collection $\mathcal{S}$ of spheres.

First, here is a brief overview of our approach. It is fairly easy to show that a segment e intersects a sphere S if and only if one of the following two conditions hold:

(i) One endpoint of e lies inside S and the other lies outside S.

(ii) Both endpoints of e lie outside S, the center of S lies inside the slab Σ_e bounded by the two planes passing through the endpoints of e and perpendicular to e, and the line containing e intersects S. See Figure 1 for an illustration.

Consequently, we construct two data structures, one for testing whether condition (i) holds for any sphere, and the other for testing condition (ii).

2.2. First data structure. We start by describing the first (and simpler) data structure. Its first level $\mathcal{T}_1$ is used to find all spheres of $\mathcal{S}$ containing a query point. The root of $\mathcal{T}_1$ is associated with the entire set $\mathcal{S}$. We fix some sufficiently large constant parameter r and construct a $(1/r)$-net $\mathcal{E}$ of $O(r \log r)$ spheres from $\mathcal{S}$ for certain ranges (subsets of $\mathcal{S}$) that will be defined in a moment. ($\mathcal{E}$ is a $(1/r)$-net if each range that contains more than $|\mathcal{S}|/r$ elements must contain an element of $\mathcal{E}$; see [23] for more details.) $\mathcal{E}$ is constructed by the deterministic technique of Matoušek [25] (or by simply choosing a random sample of $O(r \log r)$ spheres). We next construct

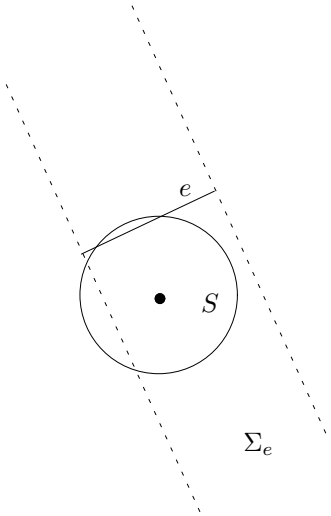


FIG. 1. *Illustration of condition* (ii) *for segment-intersection detection.*

the arrangement $\mathcal{A}(\mathcal{E})$ of these spheres and apply the *vertical decomposition* technique of [18] (see also [14]) to that arrangement. It decomposes 3-space into $O(r^{3+\varepsilon})$ cells of "constant description complexity" so that the interior of each cell is not crossed by any of the spheres of $\mathcal{E}$; see [18] for more details. The ranges with respect to which $\mathcal{E}$ has to be a $(1/r)$-net are sets of spheres of $\mathcal{S}$, each consisting of those spheres intersecting a region having the shape of a cell of the vertical decomposition in an arrangement of spheres, as above. As is easy to see, the resulting range space has so-called finite VC-dimension, which thus implies the existence of a $(1/r)$-net $\mathcal{E}$ with only $O(r \log r)$ spheres. (We again refer the reader to [23] for more details.) For each cell C of the decomposed arrangement, we create a child of the root of $\mathcal{T}_1$ and associate with it the set $\mathcal{S}_C$ of spheres intersecting C. Since $\mathcal{E}$ is a $(1/r)$-net, none of the sets $\mathcal{S}_C$ contains more than n/r spheres. We also store with each cell C the set $\mathcal{S}_C^0$ of all spheres that completely contain C in their interior. We then continue the preprocessing recursively at each child of the root with its associated set $\mathcal{S}_C$.

The second-level structure is built for each of the nodes of the first level. The set of spheres associated with the root of the second-level structure at a node corresponding to some cell C in the first-level decomposition is the set $\mathcal{S}_C^0$ defined above. The purpose of the second-level structure is to test whether any of the given spheres does not contain a query point. This is achieved using almost the same preprocessing as for the first level, except that here we define the set $\mathcal{S}_C^0$ to be the set of all spheres whose enclosed ball is disjoint from the cell C.

Given a query segment $e = pq$, we first search with p through the first-level structure, obtaining the set of spheres containing p as a disjoint union of sets $\mathcal{S}_C^0$. For each of these sets, we search with q through the corresponding second-level structure; if we find a node there whose set $\mathcal{S}_C^0$ is nonempty, we stop and conclude that e intersects a sphere of $\mathcal{S}$. Otherwise, we repeat this procedure, searching with q in the first-level structure and with p in the second-level structures. Again, we either detect an intersection between e and some sphere or else conclude that no intersection of type (i) occurs, and then we move on to search in the second data structure.

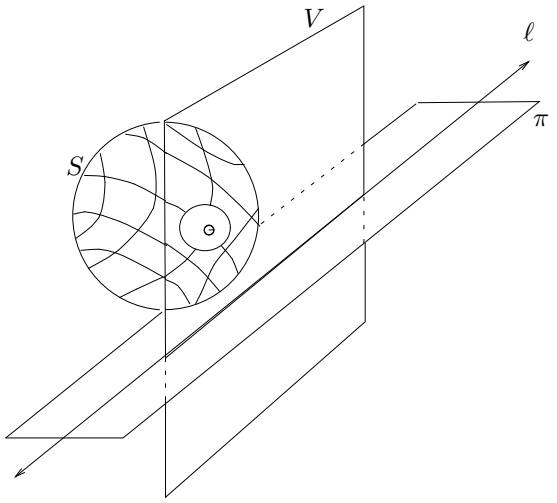


FIG. 2. *The planes V and π and their relationships to a sphere.*

2.3. Second data structure. The first two levels of the second data structure are very similar to those of the first data structure. Their purpose is to filter out all spheres that do not contain any of the endpoints of the query segment e. These levels are constructed in much the same way as above, except that the sets $\mathcal{S}_C^0$ in both levels are the sets of spheres whose enclosed balls are disjoint from the corresponding cell C. We search with e in these two levels as described above, and the search results in a collection of canonical sets $\mathcal{S}_C^0$ of spheres whose (disjoint) union is the subset of all spheres whose enclosed balls do not contain any endpoint of e.

The next two levels of the structure aim at finding all spheres whose centers lie in the slab Σ_e as defined in condition (ii) above. In both levels, we take the set of the centers of the relevant spheres and preprocess them for half-space range searching in three dimensions (using, e.g., the method of [17]). Since Σ_e is the intersection of two half-spaces H_1 and H_2, we search with H_1 in the third-level structures and with H_2 in the fourth-level structures, and we wind up with a collection of canonical sets of spheres whose (disjoint) union is the set of all spheres that satisfy the first two parts of condition (ii). Hence for each of the resulting sets of spheres, we only need to determine whether the line ℓ containing the query segment e intersects any sphere in that set.

The fifth level of our structure is also a half-space range-searching structure on the centers of the relevant spheres. We search in that level with the two half-spaces bounded by the plane π passing through ℓ and orthogonal to the vertical plane V passing through ℓ. Let π^+ denote the half-space lying above π, and let π^- denote the half-space lying below π. Let S be a sphere whose center lies in π^+ and which intersects V in a disc D. Then the center of D lies above ℓ, so either ℓ intersects S or else it *passes below* S, in the sense that ℓ and S are disjoint and there is a point on ℓ that lies vertically below a point in S. See Figure 2 for an illustration. A symmetric property holds if the center of S lies in π^-; we will continue the description of our data structure so that it handles only the latter situation since the handling of the former case is fully symmetric.

To recap, we have reduced our problem to the following subproblem. We are

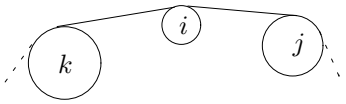


FIG. 3. *Two bridges connecting a disc D_i to two larger discs.*

given a collection of arbitrary spheres in 3-space, which is a canonical set of spheres in the output of the fifth level of our structure, and we want to preprocess it so that we can answer the following queries efficiently: we are given a line ℓ such that for each sphere S in our collection, either ℓ intersects S or else there is no point of S that lies vertically above ℓ. We want to determine whether ℓ intersects any of the given spheres. For brevity of notation, we call the given collection $\mathcal{S}$ and assume its size to be n.

2.4. Detecting intersection between a line and the upper hull of a vertical planar cross-section of $\mathcal{S}$. We begin by describing a few constructs that will be used in our technique. For each point (ξ_1, ξ_2), let $V = V_{\xi_1,\xi_2}$ denote the vertical plane passing through the line $y = \xi_1 x + \xi_2$. We denote by (ρ, z) the coordinates within V, where ρ is the horizontal coordinate (along the line $y = \xi_1 x + \xi_2$) and z is the vertical coordinate. Let $\mathcal{E} = \{S_1, \ldots, \mathcal{S}_t\}$ be a subset of t spheres of $\mathcal{S}$. Let $D_1, \ldots, D_t$ denote the (possibly empty) two-dimensional discs formed by intersecting the corresponding spheres $S_1, \ldots, S_t$ with V. We consider the upper convex hull of these discs. This hull consists of *bridges*, namely common tangents between pairs of discs which form edges of the hull, and of circular arcs which are portions of the boundaries of the discs and which lie between two adjacent bridges; see Figure 3. Suppose the disc D_i is part of the upper convex hull of these t discs (within V), and consider the set of bridges that connect D_i with other discs whose radius is greater than or equal to that of D_i. It is easily checked that there can be at most two such bridges, at most one connecting D_i to a disc D_j following it in the positive ρ-direction and at most one connecting D_i to a disc D_k preceding it; see Figure 3 again. (This claim is trivial if the discs are pairwise disjoint, but it also holds if they are allowed to intersect.) For each bridge b, we denote by b^+ the half-plane lying above the line containing it. For each circular arc γ of the hull, we denote by γ^+ the region above it (that is, we erect a vertical ray upwards from each point on γ and take the union of all these rays); see Figure 4.

Here is a quick overview of our approach: we fix some sufficiently large constant parameter r and construct a set $\mathcal{E} \subseteq \mathcal{S}$ of size $t = O(r \log r)$, which will serve as a $(1/r)$-net with properties that will be explained in a moment. Consider a query line ℓ and the vertical plane V passing through it. We will build a data structure on $\mathcal{E}$, from which we will be able to quickly determine the set of bridges of the upper convex hull of the t intersection discs of the spheres of $\mathcal{E}$ with V. We will then check if ℓ intersects this hull. If it does, we are done—ℓ intersects a sphere of $\mathcal{S}$. Otherwise, we find two bridges b and b' whose slopes are nearest to the slope of ℓ so that ℓ lies completely in the union of the regions b^+, b'^+, and γ^+, where γ is the circular arc between b and b'; see Figure 4. We then continue the query recursively in the set of spheres intersecting each of these three regions. For this approach to be efficient, we need each of the regions b^+ and γ^+ to be intersected by a small number of spheres (at most n/r spheres). We will actually define two range spaces here, where each of the ranges of each of these spaces is defined in terms of a constant number of spheres and both range spaces have finite VC-dimension. The first range space deals with ranges

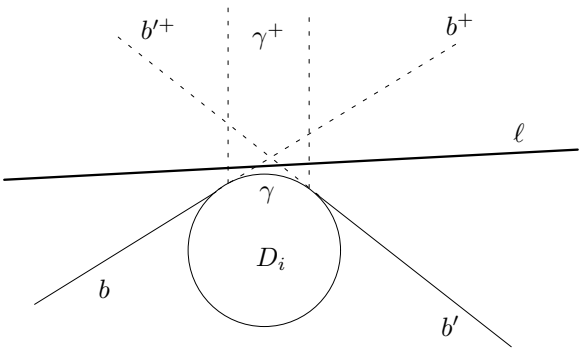


FIG. 4. *A line ℓ passing above the upper convex hull, and the three regions whose union contains ℓ.*

related to the regions b^+, and the second range space deals with ranges related to the regions γ^+. We will build a $(1/r)$-net $\mathcal{E}_1$ of size $O(r \log r)$ for the first range space and another $(1/r)$-net $\mathcal{E}_2$ of size $O(r \log r)$ for the second range space, so the union $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$ is a $(1/r)$-net for both range spaces of size $t = O(r \log r)$.

2.5. The planar maps $M^{\pm}(S_i)$. We now describe our technique in detail. We build a data structure so that given any vertical plane $V = V_{\xi_1,\xi_2}$ represented by the point (ξ_1, ξ_2) as above, the bridges and circular arcs of the upper convex hull of the discs D_i within V can be found quickly. For each sphere $S \in \mathcal{E}$, we construct two planar maps $M^+(S)$ and $M^-(S)$ in the $\xi_1\xi_2$-coordinate system, where ξ_1 is the slope and ξ_2 is the intercept of the xy-projection of a line in 3-space. If D_i contributes to the upper convex hull and the unique larger disc following it along the hull in the positive ρ-direction is D_j, we label in $M^+(S_i)$ the point (ξ_1, ξ_2) by the index j; similarly, if D_i contributes to the upper convex hull and the unique larger disc following it along the hull in the negative ρ-direction is D_k, we label in $M^-(S_i)$ the point (ξ_1, ξ_2) by the index k; otherwise, we label (ξ_1, ξ_2) by 0 in the respective map. (This alternative labeling also applies in the cases where D_i is empty or does not appear at all along the upper hull.) The faces of $M^+(S_i)$ and of $M^-(S_i)$ are maximal connected regions, all of whose points have the same label; edges and vertices of these maps are defined accordingly. The edge between two adjacent cells of the map $M^+(S_i)$ $(M^-(S_i))$, which are labeled by j and k, respectively, consists of points (ξ_1, ξ_2) where the disc following (preceding) D_i along the hull is about to change from D_j to D_k, or the radius of D_j becomes equal to the radius of D_i, or D_i is getting out of the hull boundary by another disc. Thus each edge e of any of these maps consists of points (ξ_1, ξ_2) at which the upper hull of the discs in the plane V_{ξ_1,ξ_2} has a bridge tangent to three discs, or has two adjacent discs of equal radius, or has a disc degenerating to a single point. See below for a more detailed analysis of these maps; we will show that the overall complexity of all these maps is only roughly cubic in r. We next construct a vertical decomposition of each of the planar maps $M^+(S_i)$ and $M^-(S_i)$ by erecting a vertical segment from each vertex and from the points with ξ_2-vertical tangency on each edge, and we extend it upwards and downwards until it hits another edge of the map. Note that the set of labels of a point (ξ_1, ξ_2) in all of these maps provides complete information about the structure of the upper convex hull of the intersection discs D_i in the vertical plane V_{ξ_1,ξ_2}.

Let C be a cell in, say, the vertical decomposition of the map $M^+(S_i)$, and suppose

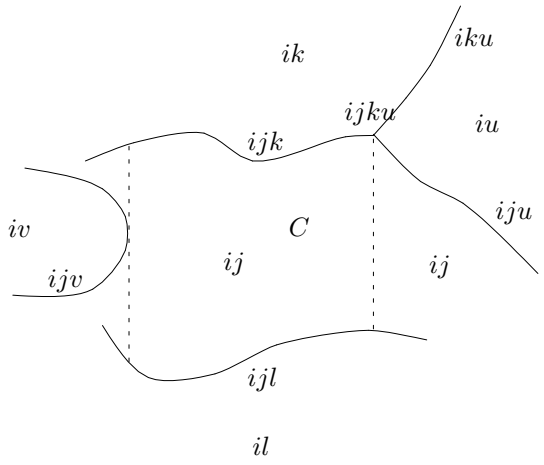


FIG. 5. *A cell C labeled by $j > 0$ in the map $M^+(S_i)$.*

that its label is $j > 0$ (cells with label 0 can be ignored in this process). For each point (ξ_1, ξ_2) in C, we consider the plane $V = V_{\xi_1,\xi_2}$ as defined above. We know that S_i contributes to the upper convex hull of the discs of intersection of the spheres of $\mathcal{E}$ with this plane, and we know that the unique next larger disc along the hull in the positive ρ-direction is D_j. Let b denote the bridge connecting the two discs in the upper convex hull; we denote by $b^+(\xi_1, \xi_2)$ the half-plane of V above the line containing b (that is, the half-plane supporting the discs D_i and D_j from above). Consider the three-dimensional region $R_C = \bigcup_{(\xi_1,\xi_2)\in C} b^+(\xi_1, \xi_2)$, and let $\mathcal{S}_C$ be the subset of all the original spheres intersecting R_C. We define similar regions R_C for all the cells C in all the other maps $M^{\pm}(S_k)$ for $k = 1, \ldots, t$.

Note that each of the regions R_C is the union of half-planes $b^+(\xi_1, \xi_2)$, where b is a bridge determined by a fixed pair of spheres of $\mathcal{E}$ and where (ξ_1, ξ_2) ranges over a cell C of one of the maps $M^{\pm}(S_i)$, which is a region of constant complexity. Actually, suppose C has label $j > 0$. Then C is defined in terms of at most six spheres: at most three spheres, S_i, S_j, and another sphere S_k, define the top edge of C, at most three spheres, S_i, S_j, and another S_l, define its bottom edge, at most one more sphere is needed to define the left vertical edge of C, and at most one more sphere is needed to define its right edge. See Figure 5 for an illustration. It follows that C is defined by at most six spheres of $\mathcal{E}$. Moreover, the two spheres forming the bridge b belong to this collection of six spheres, so R_C is also defined in terms of at most six spheres and is thus a region of constant complexity. The ranges with respect to which $\mathcal{E}_1$ (and hence $\mathcal{E}$) should be a $(1/r)$-net are subsets of the form $\mathcal{S}_C$, defined for regions R_C which are defined in terms of at most six spheres, as just outlined. Again, it is easy to verify that the resulting range space has finite VC-dimension. Since by construction none of the spheres of $\mathcal{E}$ intersects any of the regions R_C, it follows by definition that the maximum cardinality of the sets $\mathcal{S}_C$ over all cells C of all the $2t$ maps $M^{\pm}(S_i)$ is at most n/r.

We construct the sets $\mathcal{S}_C$ using any brute-force method, which takes $O(n)$ time since we assume r to be constant. For each cell C in each map, we then create a child of the root node of $\mathcal{T}$, associate the set $\mathcal{S}_C$ of spheres with that child, and continue the construction of the data structure recursively at each child with the corresponding set

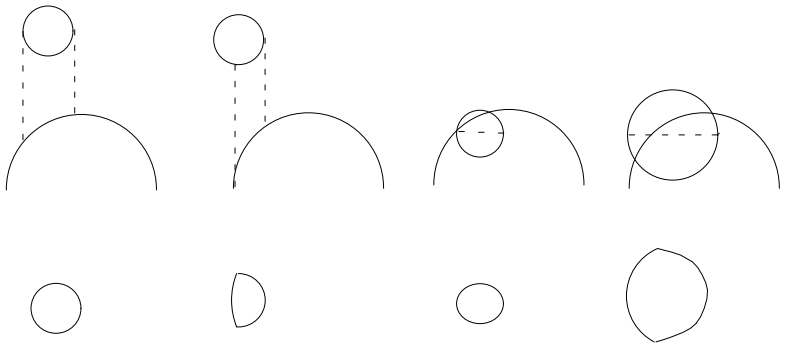


FIG. 6. *The possible xy-projections of the portions S_j^+. (The projections have smooth boundaries, but some are drawn with nonsmooth boundary to illustrate the fact that they are composed of two different arcs.)*

of spheres. There is, however, a second set of children of the root, obtained through a second decomposition scheme, which we now proceed to describe.

2.6. The spatial maps $N(S_i)$. We now define our second range space, with respect to which $\mathcal{E}_2$ (and hence $\mathcal{E}$) should be a $(1/r)$-net. Let N denote the three-dimensional parametric space with coordinates (ξ_1, ξ_2, ρ), where, as above, (ξ_1, ξ_2) give the dual representation of a line $\lambda(\xi_1, \xi_2) : \ y = \xi_1 x + \xi_2$ in the xy-plane and where ρ measures the distance along $\lambda(\xi_1, \xi_2)$. Note that each point $(\xi_1, \xi_2, \rho) \in N$ actually represents (in a many-to-one manner) a point in the xy-plane, namely the point $\sigma(\xi_1, \xi_2, \rho)$ with coordinate ρ along the line $\lambda(\xi_1, \xi_2)$. The reason for this three-dimensional representation of the xy-plane will become clearer later on.

For each of the t spheres $S_i \in \mathcal{E}$, we form in N a spatial subdivision $N(S_i)$ consisting of roughly $O(t^2)$ cells of constant complexity, as follows. For any other sphere $S_j \in \mathcal{E}$, let S_j^+ denote the portion of S_j that lies above S_i and let S_j^* denote the xy-projection of S_j^+. We obtain a collection of at most t planar regions, each of which is the intersection of Q_i, the xy-projection of S_i, with either a disc, an ellipse, or a convex region with a smooth boundary which is the union of a circular arc and an elliptic arc; see Figures 6 and 7.

Let K_i denote the complement (within Q_i) of the union $\bigcup_{j \neq i} S_j^*$. We map K_i into the following set in our parametric space N:

$$\tilde{K_i} = \{(\xi_1, \xi_2, \rho) \ : \ \rho \in K_i(\xi_1, \xi_2)\},$$

where $K_i(\xi_1, \xi_2) = K_i \cap \lambda(\xi_1, \xi_2)$. In other words, $\tilde{K_i}$ is the preimage of K_i under the mapping σ from N onto the xy-plane, as defined above. Note that a point (ξ_1, ξ_2, ρ) is in $\tilde{K_i}$ if and only if the vertical line (in actual 3-space) passing through the point on $\lambda(\xi_1, \xi_2)$ with coordinate ρ does not meet any sphere of $\mathcal{E}$ above its highest intersection with S_i.

We next decompose $\tilde{K_i}$ into constant-complexity cells by applying a standard vertical decomposition (with ρ being the "vertical" coordinate) as in [14, 18]. This yields the spatial map $N(S_i)$. As observed in the papers just cited (and as is easily seen), the complexity (number of resulting cells) of $N(S_i)$ is proportional to the number of *vertically visible* pairs (e, e') of edges of $\tilde{K_i}$ (where we also include among these edges the loci of points with vertical tangency along the boundary of $\tilde{K_i}$). Such

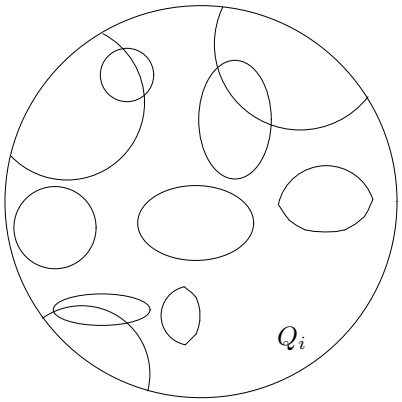


FIG. 7. *The arrangement of the xy-projections of the portions* S_j^+.

a pair of edges (e, e') is vertically visible if there exists a point (ξ_1, ξ_2) such that the vertical line in N passing through that point intersects both e and e' and the segment between these points of intersection lies fully in $\tilde{K}_i$. An edge e of $\tilde{K}_i$ consists of points (ξ_1, ξ_2, ρ) such that the line $\lambda(\xi_1, \xi_2)$ passes through a vertex of K_i or is tangent to the boundary of K_i and ρ is the coordinate of that vertex or point of tangency along $\lambda(\xi_1, \xi_2)$. Thus each vertically visible pair (e, e') of edges of $\tilde{K}_i$ corresponds to a line $\lambda(\xi_1, \xi_2)$ which passes through two points, each being either a vertex of K_i or a point of tangency between $\lambda(\xi_1, \xi_2)$ and the boundary of K_i, so that the interval between these two points is fully contained in K_i.

In other words, the number of vertically visible pairs (e, e') as above is proportional to $\sum_f c_f^2$, where the sum extends over all faces f of K_i and where c_f is the complexity of f. All of these faces are faces of a planar arrangement of $O(t)$ circular and elliptic arcs, and each pair of these arcs intersect in at most four points. Hence, as shown in [20], we have $\sum_f c_f^2 = O(\lambda_6^2(t))$, where $\lambda_6(t) = t \cdot 2^{O(\alpha^2(t))}$ is the (nearly linear) maximum length of a $(t, 6)$ *Davenport–Schinzel sequence* [8]. We have thus shown that the number of cells (each having constant complexity) in all the subdivisions $N(S_i)$ is $O(t\lambda_6^2(t))$, and it is thus nearly cubic in r (recall that $t = O(r \log r)$).

We next associate a set $\mathcal{S}_C$ of spheres with each cell C of $N(S_i)$. For each point $(\xi_1, \xi_2, \rho) \in C$, let $\ell(\xi_1, \xi_2, \rho)$ denote the vertical line (in actual 3-space) passing through the point of coordinate ρ on the line $\lambda(\xi_1, \xi_2)$, and let $\ell^+(\xi_1, \xi_2, \rho)$ denote the portion of $\ell(\xi_1, \xi_2, \rho)$ above its highest intersection point with S_i. Let R_C denote the union of all the rays $\ell^+(\xi_1, \xi_2, \rho)$ for $(\xi_1, \xi_2, \rho) \in C$. The set $\mathcal{S}_C$ consists of all spheres of $\mathcal{S}$ that intersect R_C. We note that R_C is a region of constant complexity defined in terms of only a constant number of spheres of $\mathcal{E}$. The ranges with respect to which $\mathcal{E}_2$ (and hence $\mathcal{E}$) should be a $(1/r)$-net are subsets $\mathcal{S}_C$, each consisting of those spheres that intersect a region of the form R_C defined for cells C in N that have the same structure as the cells of the vertical decompositions constructed above. Again, it is easily seen that the resulting range space has finite VC-dimension. (Intuitively, this is due to the fact that each of the regions R_C has constant description complexity; see, e.g., [29] for more details.)

We next observe that none of the spheres of $\mathcal{E}$ intersect R_C. Indeed, if one of these spheres, S_j, did intersect R_C, then there would exist a point $(\xi_1, \xi_2, \rho) \in \tilde{K}_i$ such that the vertical line $\ell(\xi_1, \xi_2, \rho)$ meets S_j at a point that lies above S_i. Then,

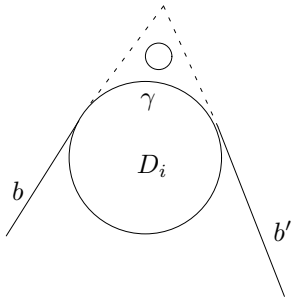


FIG. 8. *An original sphere intersecting V in a disc meeting the small region between D_i and the extension of its two bridges.*

however, by definition, the line $\ell(\xi_1, \xi_2, \rho)$ meets the xy-plane at a point belonging to S_j^*, so by definition of K_i, ρ does not belong to $K_i(\xi_1, \xi_2)$ and thus (ξ_1, ξ_2, ρ) does not belong to $\tilde{K}_i$, a contradiction. We thus conclude that the maximum cardinality of the sets $\mathcal{S}_C$ over all cells C of $N(S_i)$ and over all spheres S_i of $\mathcal{E}$ is at most n/r. We construct the sets $\mathcal{S}_C$ by any brute-force method in linear time (r is assumed to be a constant), create a new set of additional children of the root of the tree $\mathcal{T}$, one child for each cell C in each of the subdivisions $N(S_i)$, associate with that node the corresponding set $\mathcal{S}_C$ of spheres, and continue the construction of the data structure recursively at each child with the corresponding set of spheres. This concludes the description of the sixth (and last) level of our data structure.

2.7. Answering a query. We next describe how a query is processed. Let $e = pq$ be a query segment in 3-space. We have already described how to search with e in the first data structure and how to search with e in the first five levels of the second data structure. Let $\mathcal{S}$ be a canonical set of spheres in the output of the search in the first five levels of the second structure, and let ℓ be the line containing e. We assume with no loss of generality that for each sphere $S \in \mathcal{S}$, either ℓ meets S or else S has no point that lies vertically above ℓ. Our goal is to determine whether ℓ intersects any sphere in $\mathcal{S}$.

Suppose ℓ is represented by the two equations $y = \xi_1 x + \xi_2$ and $z = \xi_3 x + \xi_4$. We start at the root of the sixth-level tree of $\mathcal{S}$ and locate the point (ξ_1, ξ_2) in each of the $2t$ maps $M^+(S_i)$ and $M^-(S_i)$ of the root. Let $C_1, \ldots, C_k$, for some $k \leq 2t$, denote the cells of these maps which contain (ξ_1, ξ_2) and whose label is not 0. As noted, the labels of these cells provide a complete description of the upper convex hull of the t intersection discs D_i of our net spheres with the plane V_{ξ_1,ξ_2}. We compute the slopes $\mu_1, \ldots, \mu_k$ of the k bridges of that hull and compare each of them with the slope of the query line within V, which is, as is easily verified, $\kappa = \xi_3/\sqrt{1+\xi_1^2}$. If one of the bridges, say b, has slope $\mu_b = \kappa$, we simply check whether our query line passes below b, in which case ℓ intersects a sphere of $\mathcal{S}$, so we can return a positive answer to the query. Otherwise, ℓ passes above (the line containing) b, and we recurse in the subtree corresponding to the cell C that defines b. (Only spheres in $\mathcal{S}_C$ can meet ℓ.) In general, though, we will obtain two adjacent bridges, b and b', both tangent to the same sphere of $\mathcal{E}$, say S_i, such that $\mu_b \leq \kappa \leq \mu_{b'}$. In this case, if an original sphere S intersects ℓ, then either S meets one of the two half-planes $b^+(\xi_1, \xi_2)$ and $b'^+(\xi_1, \xi_2)$ or $S \cap V_{\xi_1,\xi_2}$ meets the small region R enclosed between S_i and the two lines containing b and b'; see Figure 8 for an illustration. Let γ denote the (upper) arc of D_i between

its two points of tangency with b and b'. It is clear that no sphere of $\mathcal{E}$ intersects the region γ^+ consisting of all points of V_{ξ_1,ξ_2} that lie above γ, which implies that the xy-projection γ^* of γ is fully contained in (a single connected component of) the region $K_i(\xi_1,\xi_2)$, as defined above. It thus follows that the set $\tilde{\gamma}$ consisting of all points (ξ_1,ξ_2,ρ), where ρ is the coordinate along $\lambda(\xi_1,\xi_2)$ of a point of γ^*, is fully contained in $\tilde{K}_i$ and, in fact, is fully contained within a single cell of the subdivision $N(S_i)$ of $\tilde{K}_i$. ($\tilde{\gamma}$ is a vertical segment contained in $\tilde{K}_i$, and such a segment must be fully contained in a single cell of its vertical decomposition.)

We thus continue the search recursively at the two nodes of $\mathcal{T}$ associated with the cells corresponding to the bridges b and b' and with the node corresponding to the cell C of $N(S_i)$ containing the arc $\tilde{\gamma}$. This concludes the description of the query processing.

2.8. Complexity analysis. We next analyze the complexity of our algorithm in terms of space, preprocessing time, and query time. The first five levels of our second structure (as well as the two levels of the first structure) involve half-space range searching and structures for point location among spheres in $\mathbb{R}^3$. At each of these levels, the size of the structure for a set of m spheres, excluding substructures at deeper levels, is $O(m^{3+\varepsilon})$ for any $\varepsilon > 0$. Indeed, for half-space range searching, this follows from [17]. For point location among spheres, this follows by noting that the maximum storage $S(m)$ and preprocessing cost $P(m)$ for this structure on m spheres satisfy the following recurrences:

$$P(m) \le c_1 r^{3+\varepsilon} P(m/r) + c_2 m r^{3+\varepsilon}$$

and

$$S(m) \le c_1 r^{3+\varepsilon} S(m/r) + c_2' r^{3+\varepsilon}$$

for appropriate constants c_1, c_2, and c_2' (some of which depend on ε). The solution of these recurrences is easily seen to be $O(m^{3+\varepsilon'})$ for another, still arbitrarily small $\varepsilon' > \varepsilon > 0$, where the constant of proportionality depends on ε.

As observed in [26] (and as is easy to verify), the overall size and preprocessing cost of a multilevel data structure of the type considered here can be deduced from the maximum size and preprocessing at any fixed level. In particular, if we show that the sixth level of our structure also requires $O(m^{3+\varepsilon})$ storage and preprocessing for a set of m spheres, it will follow that the overall storage and preprocessing cost of the full multilevel structure will also be $O(n^{3+\varepsilon})$ (with a larger constant of proportionality); see [26] for more details.

We thus turn to consider the cost of the sixth level of our structure, which involves the upper convex hull structure of the intersection discs. We first claim that the total complexity of all the $2t$ planar maps $M^+(S_i)$ and $M^-(S_i)$ is only roughly $O(r^3)$. To see this, ignore for now the added complexity caused by the vertical segments forming the vertical decompositions of the maps. (This increases the complexity by only a small constant factor.) Fix a sphere $S_i \in \mathcal{E}$ and let C be a cell of, say, $M^+(S_i)$ with a nonzero label j. Each edge e on the boundary of C consists of points (ξ_1,ξ_2) at which either the sphere S_j following S_i in the upper hull is about to change, or S_i itself is about to disappear from the upper hull, or the sizes of the two intersection discs D_i and D_j become equal. In the first two cases, either D_i shrinks to a point and then disappears from V or the (line containing the) bridge connecting D_i and D_j in the plane V_{ξ_1,ξ_2} becomes tangent to a third disc D_k (it is also possible that D_i

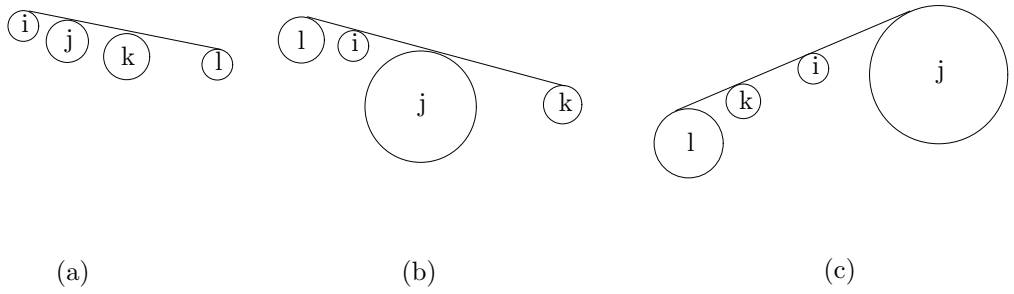


FIG. 9. *The three possible configurations that can occur at a vertex of the map* $M^+(S_i)$.

and D_j become tangent to each other); in the first case, D_k is between D_i and D_j or after D_j, and in the second case, D_k precedes D_i. Each vertex v of C is formed by the intersection of two such boundary edges e and e'. Suppose first that both e and e' represent events of triple tangency. Then v must be of one of the following three types:

- Each of the edges e and e' represents a change in the sphere following S_i. Denote the two corresponding new spheres that are about to replace S_j as the next sphere by S_k and S_l. Since v lies on e, the discs D_i, D_j, and D_k have a common tangent that appears as a bridge in the upper convex hull. Similarly, since v also lies on e', the same is true for D_i, D_j, and D_l. Hence at v the four discs D_i, D_j, D_k, and D_l have a common tangent bridge. See Figure 9(a) for an illustration.
- One of the edges, say e, represents a change in the sphere following S_i, say from S_j to some S_k, and the other edge e' represents a situation where S_i is displaced from the upper convex hull by another sphere S_l preceding S_i along the hull. As is easily verified, in this case as well, the four corresponding discs D_i, D_j, D_k, and D_l must have a common tangent bridge at v. See Figure 9(b) for an illustration.
- Each of the edges e and e' represents a situation where S_i is displaced from the upper convex hull, by two respective other spheres S_k and S_l preceding S_i along the hull. In this case as well, the four corresponding discs D_i, D_j, D_k, and D_l must have a common tangent bridge at v. See Figure 9(c) for an illustration.

We have thus shown that each vertex of $M^+(S_i)$ of these types is a point (ξ_1, ξ_2) for which the plane V_{ξ_1,ξ_2} contains a line tangent to four intersection discs and passing above all other spheres. Similar arguments imply that this also holds for all the other maps $M^{\pm}(S_k)$. Note that, assuming general position of the spheres, each such vertex can be present in at most eight of the $2t$ maps, twice for each of the at most four spheres that define it. (It has no effect on the labeling of cells in other maps.) Similarly, each edge in one of these maps is defined in terms of at most three spheres and can thus appear only in the maps of those three spheres.

Note that if two edges e and e', both representing situations where the size of D_i becomes equal to that of the following disc on the upper hull, meet at a point v, then v also lies on an edge of a triple tangency, as is easily seen.

The next case to consider is thus that of vertices formed by the intersection of two edges e and e', where e consists of points (ξ_1, ξ_2) at which the upper hull of the intersection discs in the plane V_{ξ_1,ξ_2} contains a bridge tangent to D_i and to two other discs, D_j and D_k, and e' consists of points (ξ_1, ξ_2) at which the size of D_i becomes equal to the size of one of these discs, say D_j. It is easily seen that the number of

such vertices is only $O(t^3)$ over the entire collection of maps $M^{\pm}(S_i)$ because each such vertex is defined by three spheres of $\mathcal{E}$, and each triple of spheres gives rise to only a constant number of such vertices.

Finally, consider cases where a disc D_i shrinks to a point or becomes tangent to an adjacent disc along the upper hull. Again, it is easily verified that a vertex v of $M^{\pm}(S_i)$ at which this occurs is defined in terms of at most three spheres (S_i plus two other spheres defining an edge of triple tangency on which v lies, or S_i, S_j, and another sphere becoming tangent to the common tangent line to D_i and D_j). Thus the total number of such vertices over all maps is $O(t^3)$.

We thus need to bound the number of points (ξ_1, ξ_2) at which the upper convex hull of the intersection discs between the spheres and the vertical plane V_{ξ_1,ξ_2} has a bridge tangent to four discs. To do so, we apply the following transformation to the problem. For each sphere $S \in \mathcal{E}$, we define a partial trivariate function $F_S(\xi_1, \xi_2, \xi_3)$ as follows. Suppose that the intersection disc $D = S \cap V_{\xi_1,\xi_2}$ is not empty (otherwise, $F_S(\xi_1, \xi_2, \xi_3)$ is undefined). We define $F_S(\xi_1, \xi_2, \xi_3)$ to be the value of ξ_4 for which the line $y = \xi_1 x + \xi_2$, $z = \xi_3 x + \xi_4$ is tangent to D from above. Note that F_S is (partially) well defined and that its graph is an algebraic surface patch of low constant degree.

It follows by definition that the upper envelope of the functions F_S represents the locus of all lines that are tangent to the upper convex hull of any vertical planar cross-section of the spheres of $\mathcal{E}$. Moreover, each vertex of the upper envelope represents a line tangent to such an upper convex hull and touching four of the corresponding intersection discs, and each such "critical" line is indeed a vertex of the envelope.

By the recent results of [22, 28], the complexity of the upper envelope of n algebraic surface patches of constant maximum degree in d-space is $O(n^{d-1+\varepsilon})$ for any $\varepsilon > 0$. Hence it follows that the complexity of the upper envelope of the $t = O(r \log r)$ functions F_S is $O(r^{3+\varepsilon})$ for any $\varepsilon > 0$. This implies that the number of vertices of the types considered above in our $2t$ maps $M^{\pm}(S_i)$ is $O(r^{3+\varepsilon})$, which is thus also a bound on the overall complexity of the vertical decompositions of these maps. As already argued, this bound also dominates the overall complexity of the spatial subdivisions $N(S_i)$ and implies that the number of children of any node in any sixth-level tree structure is $O(r^{3+\varepsilon})$.

We can finally complete the analysis of the performance of our algorithm. Assuming r to be a (sufficiently large) constant, the construction of the $(1/r)$-net $\mathcal{E}$ can be done in $O(n)$ time, as in [25]. The construction of the $2t$ maps $M^{\pm}(S_i)$ can be done in constant time. More precisely, we first need to construct the upper envelope of the functions F_{S_i}. As shown in [2], this can be done in time $O(r^{3+\varepsilon})$ for any $\varepsilon > 0$. The features of these envelopes are then distributed among the maps $M^{\pm}(S_i)$, the additional edges and vertices of these maps are constructed in a straightforward manner, and the maps are then vertically decomposed, within the same asymptotic running time.

The construction of the t spatial subdivisions $N(S_i)$ can also be performed in constant time. For each fixed sphere $S_i \in \mathcal{E}$, we first compute the "top portions" S_j^+ of the other spheres of $\mathcal{E}$, project them onto the xy-plane, and compute the complement K_i of their union. We then construct all vertices of $N(S_i)$ by examining every pair of vertices and/or edges of each face of K_i in the manner described above, from which the complete subdivision $N(S_i)$ is easy to construct.

For any cell C in each of these maps, we compute the set $\mathcal{S}_C$ of all spheres intersecting its associated region R_C. This can be done in $O(n)$ time since each such region is of constant complexity and the number of regions is also constant. More

precisely, all the canonical sets $\mathcal{S}_C$ can be computed in $O(nr^{3+\varepsilon})$ time. As already noted, the maximum size of any set $\mathcal{S}_C$ is $\leq n/r$.

Let $S_6(n)$, $T_6(n)$, and $Q_6(n)$ denote, respectively, the expected space complexity, preprocessing time, and query time for the sixth level of the data structure on a canonical set of n spheres. Then we have the following recurrence relations:

$$P_6(n) \leq c_1 r^{3+\varepsilon} P_6(n/r) + c_2 n r^{3+\varepsilon}$$

and

$$S_6(n) \leq c_1 r^{3+\varepsilon} S_6(n/r) + c_2' r^{3+\varepsilon}$$

for appropriate constants c_1, c_2, and c_2' (some of which depend on ε). As above, the solution of these recurrences is easily seen to be $O(n^{3+\varepsilon'})$ for another, still arbitrarily small $\varepsilon' > \varepsilon > 0$, where the constant of proportionality depends on ε.

As for the query time $Q_6(n)$, we have the following recurrence:

$$Q_6(n) \leq 3Q_6(n/r) + O(r^{3+\varepsilon}).$$

(The factor 3 appears since we search recursively in at most three children of any node that is visited during the query processing.) The solution of this recurrence is easily seen to be $Q_6(n) = O(n^{\varepsilon})$, again with a constant of proportionality that depends on ε.

If we now combine the sixth level of the structure with the preceding five levels, apply the observations of [26] concerning multilevel structures, as mentioned above, and also take into account the overhead of the parametric searching, we easily conclude the following main result.

THEOREM 2.1. *Given a collection $\mathcal{S}$ of n arbitrary spheres in 3-space and any $\varepsilon > 0$, one can preprocess $\mathcal{S}$ in time $O(n^{3+\varepsilon})$ into a data structure of size $O(n^{3+\varepsilon})$ which supports ray shooting queries among the spheres of $\mathcal{S}$ in time $O(n^{\varepsilon})$ per query.*

3. Extensions. In this section, we extend our algorithm so that it applies to more general classes of objects. As a matter of fact, we present a general approach to ray shooting in three dimensions amidst a collection $\mathcal{S}$ of n (possibly intersecting) convex objects, each of constant description complexity. The approach is modular and consists of several stages, where each stage filters out objects that satisfy a certain geometric constraint with respect to the query ray so that the query ray intersects all objects that satisfy the conjunction of all of these constraints; using this property, the actual ray shooting is then easy to perform. Thus to obtain a ray shooting algorithm for a specific class of objects, we need to provide appropriate and efficient filtering mechanisms for this class at each level separately. These mechanisms and their efficiency depend on the class in question; we will exemplify them for two classes of objects: triangles and ellipsoids.

As above, we only consider the case where we want to obtain fast queries ($O(n^{\varepsilon})$ time per query) and are willing to use large storage. We follow the general approach of [4], reducing the ray-shooting problem to the segment-emptiness problem. We then proceed through the following stages:

(1) We begin by reducing the segment-emptiness problem to the line-emptiness problem, and we accomplish this in the first two stages. Let $e = pq$ be the query segment and let ℓ be the line containing e. In the first stage, we want to partition our objects into three categories:

(i) *Objects that contain exactly one endpoint of e in their interior.* If there is any such object, we can stop the query immediately with a positive answer. We can compute this subset of objects (and also the two other subsets in (ii) and (iii) below) using the same technique that we have used for spheres since we did not exploit any special properties of spheres there. The main tool that we used there was the existence of a vertical decomposition of an arrangement of objects in 3-space into a slightly supercubic number of cells of constant complexity each. This can be done, e.g., for any collection of objects whose boundary is defined in terms of a constant number of algebraic equalities and inequalities of constant maximum degree [14]. Thus, in general, this stage can be performed using $O(n^{3+\varepsilon})$ storage and preprocessing and $O(\log n)$ query time.

(ii) *Objects that contain both endpoints of e in their interior.* Since the objects are assumed to be convex, we can ignore such objects because e cannot intersect any of them.

(iii) *Objects that do not contain any endpoint of e.* These objects require further treatment and are passed to the subsequent stages.

(2) Let S be an object of the third type with respect to our query segment e, and suppose that e intersects S. Since p lies outside S, there exists a plane through p that avoids S, and it is clear that q and S lie on the same side of this plane. (As a matter of fact, this holds for any plane that passes through p and avoids S.) A symmetric property holds at q. Conversely, if there exist a pair of such planes, then, as is easily seen, e intersects S if and only if ℓ intersects S. Thus at this stage, we want to sift out all objects for which there exist a pair of planes with these properties.

Here is a way (perhaps not always the most efficient way) of doing this. Suppose that each object $S \in \mathcal{S}$ is defined by a polynomial inequality $Q(x, y, z) \leq 1$. Since p lies outside S, there is $\lambda > 1$ such that $Q = \lambda$ at p. The plane π_p through p tangent to the surface $Q = \lambda$ is a plane that contains S fully on one side. The equation of the half-space bounded by π_p and containing S is

$$Q_x(p)(x - x_p) + Q_y(p)(y - y_p) + Q_z(p)(z - z_p) \leq 0,$$

where (x_p, y_p, z_p) are the coordinates of p. Since q also has to lie in this half-space, we obtain the constraint

$$Q_x(p)(x_q - x_p) + Q_y(p)(y_q - y_p) + Q_z(p)(z_q - z_p) \leq 0,$$

where (x_q, y_q, z_q) are the coordinates of q. A symmetric condition has to hold at q, namely

$$Q_x(q)(x_q - x_p) + Q_y(q)(y_q - y_p) + Q_z(q)(z_q - z_p) \geq 0.$$

Both inequalities are linear in the coefficients of Q. Thus if we represent each $S \in \mathcal{S}$ as a point in $\mathbb{R}^k$, where k is the number of distinct monomials appearing in the partial derivatives of the corresponding polynomials Q, the testing for the present condition amounts to two half-space range-searching queries in $\mathbb{R}^k$.

Of course, this method is not ideal if k is large, but it is nevertheless a fairly general technique. It works well for spheres: indeed, if the equation of a sphere $S_i \in \mathcal{S}$ is $(x - a_i)^2 + (y - b_i)^2 + (z - c_i)^2 = r_i^2$, then the above two constraints become

$$(x_p - a_i)(x_q - x_p) + (y_p - b_i)(y_q - y_p) + (z_p - c_i)(z_q - z_p) \leq 0$$

and

$$(x_q - a_i)(x_q - x_p) + (y_q - b_i)(y_q - y_p) + (z_q - c_i)(z_q - z_p) \geq 0,$$

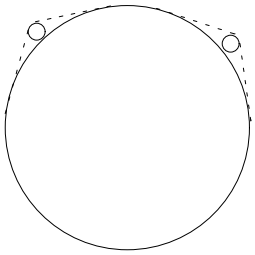


FIG. 10. *A disc appearing more than once along the upper convex hull.*

which, interestingly enough, are identical to the testing of the condition that the center of S_i lies inside the slab Σ_e, as used in our previous algorithm.

(3) At this stage, we have reduced our problem to that of detecting an intersection between a query line ℓ and the objects of $\mathcal{S}$ (or, more precisely, of some canonical subset of $\mathcal{S}$). We next need to further reduce it to the problem of detecting an intersection between ℓ and the planar upper convex hull of the cross-sections of the objects of $\mathcal{S}$ in the vertical plane V passing through ℓ, as we did for spheres. To do this we wish to partition $\mathcal{S}$ into two subsets; none of the objects in the first subset should intersect V in a cross-section that lies fully above ℓ, and none of the objects in the second subset should intersect V in a cross-section that lies fully below ℓ. We have shown that this step can be performed for spheres using an appropriate half-space range searching on the set of centers of the spheres.

(4) In the last stage, we need to decide whether our query line ℓ passes above the upper convex hull of the cross-sections of the objects of (a canonical subset of) $\mathcal{S}$ in the vertical plane V passing through ℓ. However, our solution for this stage is fairly general and can be adapted to apply to objects other than spheres. Indeed, a close inspection of our technique shows that the only place where we have used specific geometric properties of spheres is in the definition of the maps $M^{\pm}(S_i)$, where each map records the spheres that are adjacent to a sphere S_i along the upper hull, provided that the intersection of such a sphere with V is larger than the intersection of S_i with V. This was done in order to overcome the technical difficulty that the sphere following (or preceding) S_i along the hull need not be unique since one intersection disc may appear more than once along the hull; see Figure 10. Here is a way to overcome this difficulty if we assume that the objects of $\mathcal{S}$ are pairwise disjoint. In this case, their cross-sections in any vertical plane are pairwise-disjoint convex regions. We say that one such region D_i is *wider* than another region D_j if the length of the projection of D_i on the x-axis is larger than that of D_j. We now define the maps $M^{\pm}(S_i)$ in complete analogy to their definition in the case of spheres, with the proviso that a bridge connecting a cross-section D_i with a cross-section D_j is recorded in the appropriate map of the narrower of the two regions. It is easily verified that there can be at most one bridge connecting D_i with a wider D_j which follows D_i in the upper hull and at most one bridge connecting D_i with a wider region preceding D_j along the hull. Hence the maps are all well defined, and we can proceed in exactly the same manner as for spheres. If the objects in $\mathcal{S}$ can intersect, we need to devise an alternative method for recording all bridges on the upper hull in a well-defined manner. Assuming that this is indeed possible, the remainder of our algorithm applies to more general classes of objects, with only some obvious and simple modifications. We only need to require that each object in $\mathcal{S}$ has constant description complexity; this will

suffice to show that the overall complexity of the maps $M^{\pm}(S_i)$ and (the vertically decomposed) $N(S_i)$ is only $O(r^{3+\varepsilon})$. This implies that this stage of the algorithm can be performed on a set of n objects with $O(n^{3+\varepsilon})$ preprocessing and storage and with $O(n^{\varepsilon})$ query time. It is interesting that the most complex stage of our algorithm in the case of spheres is the most general stage, which applies with comparable efficiency to general classes of objects.

We now combine all these stages into a multilevel data structure, as we did for spheres. The preprocessing time and storage of the overall structure are roughly dominated by those of the most "expensive" stage, as already mentioned above and as will be illustrated below.

We next illustrate how the various stages of the algorithm can be performed for two specific classes of objects: triangles and ellipsoids.

3.1. The case of triangles. Most of the stages of the algorithm are easy to perform in the case of triangles. The first stage is void, assuming nondegenerate position of the query segment. In the second stage, it suffices to find the set of all triangles such that the two endpoints of the query segment are separated by the plane containing the triangle. This is easy to do using half-space range searching in three dimensions. The fourth stage applies in full generality, as we have noted, and the technical difficulty concerning relative sizes of the intersection objects (line segments in this case) is easy to handle; we leave details of this to the reader.

The most expensive stage, in the case of triangles, is the third one. Here we want to find all triangles passing below (resp. above) a query line or intersecting it. For each triangle Δ, we can find an edge s so that it suffices to determine whether the query line ℓ passes above (resp. below) the line containing s. This in turn can be done by mapping the problem into a half-space range-searching problem in Plücker 5-space, which can be done using $O(n^{4+\varepsilon})$ preprocessing and storage; see [15] for details. Thus the third stage is the bottleneck in obtaining an efficient ray-shooting method for triangles, and in this case, our approach does not seem to yield a better solution than those already known [5].

3.2. The case of ellipsoids. We next consider the case where our objects are ellipsoids whose axes are parallel to the coordinate axes. Each ellipsoid $S_i \in \mathcal{S}$ can be represented by the equation $A_i(x-a_i)^2 + B_i(y-b_i)^2 + C_i(z-c_i)^2 = 1$ for appropriate parameters A_i, B_i, C_i, a_i, b_i, and c_i, where A_i, B_i, and C_i are positive.

As noted above, the first stage can be applied in a general setting, which includes the case of ellipsoids. The second stage calls for the testing of the inequalities

$$A_i(x_p - a_i)(x_q - x_p) + B_i(y_p - b_i)(y_q - y_p) + C_i(z_p - c_i)(z_q - z_p) \le 0$$

and

$$A_i(x_q - a_i)(x_q - x_p) + B_i(y_q - b_i)(y_q - y_p) + C_i(z_q - c_i)(z_q - z_p) \ge 0.$$

These are homogeneous linear inequalities in the six parameters $(A_i, A_i a_i, B_i, B_i b_i, C_i, C_i c_i)$, and so this stage can be performed in projective 5-space using $O(n^{5+\varepsilon})$ storage and preprocessing and logarithmic query time.

Remark. Since this stage is the bottleneck in the case of ellipsoids, it is useful to observe that we can make this stage more efficient if our ellipsoids satisfy additional properties. For example, if they are all ellipsoids of revolution (about their vertical axis), then $A_i = B_i$ and the above inequalities become homogeneous linear inequalities

in projective 4-space, so they can be tested in $O(\log n)$ time using only $O(n^{4+\varepsilon})$ storage and preprocessing.

The third stage can be carried out as follows. Suppose that the query line ℓ is represented by the equations $y = \xi_1 x + \xi_2$ and $z = \xi_3 x + \xi_4$. Take an ellipsoid S_i and project it onto the xy-plane, obtaining the ellipse $S_i^* : A_i(x - a_i)^2 + B_i(y - b_i)^2 = 1$. The intersection of S_i^* with the projection $\ell^* : y = \xi_1 x + \xi_2$ of ℓ leads to the equation

$$A_i(x^2 - 2a_i x + a_i^2) + B_i(\xi_1^2 x^2 - 2(b_i - \xi_2)\xi_1 x + (b_i - \xi_2)^2) = 1,$$

or

$$(A_i + B_i\xi_1^2)x^2 - 2(A_i a_i + B_i b_i \xi_1 - B_i \xi_1 \xi_2)x + (A_i a_i^2 + B_i(b_i - \xi_2)^2 - 1) = 0.$$

The midpoint between the two intersection points (if they exist) thus satisfies

$$x_m = \frac{A_i a_i + B_i b_i \xi_1 - B_i \xi_1 \xi_2}{A_i + B_i \xi_1^2}.$$

It suffices to compare $z_m = \xi_3 x_m + \xi_4$ with c_i; if $z_m > c_i$, then the cross-section $S_i \cap V$ cannot lie fully above ℓ, and if $z_m < c_i$, then this cross-section cannot lie fully below ℓ. This yields the classification of the ellipsoids that we seek.

The required comparison is equivalent to the testing of the sign of the expression

$$A_i a_i \xi_3 + A_i \xi_4 + B_i b_i \xi_1 \xi_3 + B_i(\xi_1^2 \xi_4 - \xi_1 \xi_2 \xi_3) - A_i c_i - B_i c_i \xi_1^2,$$

which is a linear homogeneous expression in the six coefficients $(A_i a_i, A_i, B_i b_i, B_i, A_i c_i, B_i c_i)$. Thus the testing in this stage reduces to half-space range searching in projective 5-space. Moreover, since we are looking for a solution that uses large storage and small query time, the problem becomes, in a dual setting, that of point location among an arrangement of hyperplanes in 5-space. Since the query points lie on a four-dimensional low-degree algebraic surface (they are parametrized by the four parameters $(\xi_1, \ldots, \xi_4)$), we can exploit the zone theorem of Aronov et al. [9] to solve this problem using $O(n^{4+\varepsilon})$ storage and preprocessing with logarithmic query time (see [5] for more details).

The fourth stage is applicable in the general setting, and the technical difficulty concerning multiple bridges from the same cross-section can be overcome for pairwise-disjoint ellipsoids, as described above. Hence, putting it all together, we obtain the following result.

THEOREM 3.1. *Ray shooting amidst n axis-parallel pairwise-disjoint ellipsoids in 3-space can be performed in $O(n^{\varepsilon})$ time using $O(n^{5+\varepsilon})$ preprocessing time and storage. This can be reduced to $O(n^{4+\varepsilon})$ for ellipsoids of revolution.*

4. Conclusion. In this paper, we have developed a new and improved technique for ray shooting amidst spheres in three dimensions. Our method requires $O(n^{3+\varepsilon})$ preprocessing time and storage and performs a ray-shooting query in time $O(n^{\varepsilon})$ for any $\varepsilon > 0$, improving previous solutions by roughly an order of magnitude (in terms of preprocessing and storage costs). We have also shown that our technique can be extended to obtain a general algorithm for ray shooting amidst convex objects (of constant description complexity) in 3-space, and we have demonstrated this extension for the case of triangles (where the resulting algorithm is no worse than known solutions) and for the case of axis-parallel ellipsoids, where we obtain an algorithm that appears to be new and yields fast queries using $O(n^{5+\varepsilon})$ preprocessing and storage.

The weakness of our algorithm is that it does not seem to yield a good trade-off between storage and query time (in contrast with the less efficient solution of [4], which does have such a trade-off). Of course, we can combine our solution with that of [4] to obtain some trade-off. Readers familiar with this technique can easily check that the resulting algorithm yields a query time of $O(n^{9/8+\varepsilon}/s^{3/8})$ for $n \leq s \leq n^3$ using $O(s^{1+\varepsilon})$ storage and preprocessing. It would be interesting (and we pose it as an open problem) to design an algorithm for ray shooting amidst spheres, which takes close to linear storage and can answer ray-shooting queries in time close to $O(n^{2/3})$.

Another set of open problems is to apply our general technique for various specific classes of objects, attempting to find solutions to each stage that are as efficient as possible. In particular, can the algorithm be improved for the case of ellipsoids to require only $O(n^{4+\varepsilon})$ storage?

Finally, no lower bounds are known for the ray-shooting problem. Is our solution for the case of spheres close to optimal in the worst case?

REFERENCES

[1] P. K. Agarwal, *Ray shooting and other applications of spanning trees with low stabbing number*, SIAM J. Comput., 21 (1992), pp. 540–570.

[2] P. K. Agarwal, B. Aronov, and M. Sharir, *Computing envelopes in four dimensions with applications*, in Proc. 10th ACM Symposium on Computational Geometry, ACM, New York, 1994, pp. 348–358.

[3] P. K. Agarwal, L. Guibas, M. Pellegrini, and M. Sharir, *Ray shooting amidst spheres*, unpublished note.

[4] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, SIAM J. Comput., 22 (1993), pp. 794–806.

[5] P. K. Agarwal and J. Matoušek, *Range searching with semi-algebraic sets*, Discrete Comput. Geom., 11 (1994), pp. 393–418.

[6] P. K. Agarwal and M. Sharir, *Applications of a new space partitioning technique*, Discrete Comput. Geom., 9 (1993), pp. 11–38.

[7] P. K. Agarwal and M. Sharir, *Ray shooting amidst convex polyhedra and polyhedral terrains in three dimensions*, SIAM J. Comput., 25 (1996), pp. 100–116.

[8] P. K. Agarwal, M. Sharir, and P. Shor, *Sharp upper and lower bounds for the length of general Davenport–Schinzel sequences*, J. Combin. Theory Ser. A, 52 (1989), pp. 224–278.

[9] B. Aronov, M. Pellegrini, and M. Sharir, *On the zone of a surface in a hyperplane arrangement*, Discrete Comput. Geom., 9 (1993), pp. 177–186.

[10] R. Bar Yehuda and S. Fogel, *Good splitters with applications to ray shooting*, in Proc. 2nd Canadian Conference on Computational Geometry, Ottawa, Ontario, 1990, pp. 81–85.

[11] M. de Berg, D. Halperin, M. H. Overmars, J. Snoeyink, and M. van Kreveld, *Efficient ray shooting and hidden surface removal*, Algorithmica, 12 (1994), pp. 30–53.

[12] B. Chazelle, *Bounds on the complexity of polytope range searching*, J. Amer. Math. Soc., 2 (1989), pp. 637–666.

[13] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink, *Ray shooting in polygons using geodesic triangulations*, Algorithmica, 12 (1994), pp. 54–68.

[14] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *A singly exponential stratification scheme for real semi-algebraic varieties and its applications*, Theoret. Comput. Sci., 84 (1991), pp. 77–105.

[15] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Stolfi, *Lines in space: Combinatorics and algorithms*, Algorithmica, 15 (1996), pp. 428–447.

[16] B. Chazelle and L. Guibas, *Visibility and intersection problems in plane geometry*, Discrete Comput. Geom., 4 (1989), pp. 551–589.

[17] B. Chazelle, M. Sharir, and E. Welzl, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, Algorithmica, 8 (1992), pp. 407–430.

[18] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl, *Combinatorial complexity bounds for arrangements of curves and spheres*, Discrete Comput. Geom., 5 (1990), pp. 99–160.

[19] D. Dobkin and D. Kirkpatrick, *Determining the separation of preprocessed polyhedra: A unified approach*, in Proc. 17th International Colloquium on Automata, Languages and Programming, Springer-Verlag, Berlin, 1991, pp. 400–413.

[20] H. Edelsbrunner, L. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir, *Arrangements of curves in the plane: Topology, combinatorics, and algorithms*, Theoret. Comput. Sci., 92 (1992), pp. 319–336.

[21] L. Guibas, M. Overmars, and M. Sharir, *Ray shooting, implicit point location, and related queries in arrangements of segments*, Technical Report 433, Department of Computer Science, New York University, New York, 1989.

[22] D. Halperin and M. Sharir, *New bounds for lower envelopes in three dimensions with applications to visibility of terrains*, Discrete Comput. Geom., 12 (1994), pp. 313–326.

[23] D. Haussler and E. Welzl, *Epsilon nets and simplex range searching*, Discrete Comput. Geom., 2 (1987), pp. 127–151.

[24] J. Hershberger and S. Suri, *A pedestrian approach to ray shooting: Shoot a ray, take a walk*, in Proc. 4th ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1993, pp. 54–63.

[25] J. Matoušek, *Approximations and optimal geometric divide-and-conquer*, in Proc. 23rd ACM Symposium on Theory of Computing, ACM, New York, 1991, pp. 506–511.

[26] J. Matoušek, *Range searching with efficient hierarchical cuttings*, Discrete Comput. Geom., 10 (1993), pp. 157–182.

[27] N. Megiddo, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 30 (1983), pp. 852–865.

[28] M. Sharir, *Almost tight upper bounds for lower envelopes in higher dimensions*, Discrete Comput. Geom., 12 (1994), pp. 327–345.

[29] M. Sharir and P. K. Agarwal, *Davenport–Schinzel Sequences and Their Geometric Applications,* Cambridge University Press, Cambridge, UK, New York, Melbourne, 1995.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 675–677, June 1997

004

ON THE COMPLEXITY OF FINDING A MINIMUM CYCLE COVER OF A GRAPH*

CARSTEN THOMASSEN†

Abstract. We prove that the problem of finding a cycle cover of smallest total length is NP-hard. This confirms a conjecture of Itai, Lipton, Papadimitriou, and Rodeh from 1981.

Key words. complexity, minimum cycle cover

AMS subject classifications. 05C38, 68Q25

PII. S0097539794267255

1. Introduction. A *path* $x_1x_2\dots x_n$ is a graph with distinct vertices $x_1, x_2, \dots, x_n$ and edges $x_1x_2, x_2x_3, \dots, x_{n-1}x_n$. If we add the edge x_nx_1, we obtain a *cycle*. The *length* of a path or a cycle is the number of edges in it. A *cycle cover* of a graph G is a collection of cycles in G such that every edge is in at least one cycle. The *length* of a cycle cover is the sum of lengths of the cycles. The smallest length of a cycle cover of G is denoted $\mathrm{cc}(G)$.

There are several good upper bounds on $\mathrm{cc}(G)$ that are computable by polynomial-time algorithms; for references, see [4, 5]. However, we show that the question "Is $\mathrm{cc}(G) \le k$?" is *NP*-complete as conjectured by Itai et al. [3]. The problem is also mentioned in [4, 5] and discussed in the survey of Bondy [1].

2. Covers by paths and cycles. The subgraph induced by the vertices p_1, $p_2, \dots, p_9$ in Figure 1 together with a new vertex p_{10} joined to p_1, p_2, p_3 is the Petersen graph P.

LEMMA 2.1. $\mathrm{cc}(P) > 20$.

Proof. Consider a cycle cover of length $\mathrm{cc}(P)$. Since all vertices of P have odd degree, each vertex of P must be incident with at least one edge which is covered an even number of times. Hence $\mathrm{cc}(P) \ge 20$. Suppose equality holds. Then each vertex of P is incident with precisely one edge which is covered twice. Thus P minus the edges that are covered twice is a graph—say H—in which each vertex has degree 2. Hence H is the union of disjoint cycles. Since P has no cycle of length 10 and no cycle of length smaller than 5, H is the union of two disjoint cycles C_1 and C_2, each of length 5. If C: $z_1z_2z_3\dots$ is a cycle in the cycle cover and z_1z_2 is in C_1, say, then z_2z_3 goes from C_1 to C_2, z_3z_4 is in C_2, etc. Hence C has the same number of edges in C_1 as in C_2. Moreover, the edges of $C \cap C_1$ must be nonadjacent and hence C has at most two edges in C_1. It follows that C has length 8. Then, however, 8 divides $\mathrm{cc}(P)$, a contradiction. □

It is easy to show that $\mathrm{cc}(P) = 21$, but we shall not use this fact.

We denote by $H(k,q)(k \ge 1, q \ge 1)$ the graph obtained from the graph of Figure 1 by replacing each edge y_iz_i by a path of length k $(i = 1, 2, 3)$ and each edge of the form $p_ip_j(1 \le i < j \le 9)$ by a path of length q. By a *path-cycle cover* of $H(k,q)$, we mean a collection of cycles and paths covering the edges of $H(k,q)$ such that each path in the collection connects two vertices in $\{x_1, x_2, x_3\}$. If s_i is the number of

*Received by the editors May 9, 1994; accepted for publication (in revised form) June 28, 1995.
http://www.siam.org/journals/sicomp/26-3/26725.html
†Mathematical Institute, Technical University of Denmark, DK-2800 Lyngby, Denmark (c.thomassen@mat.dtu.dk).

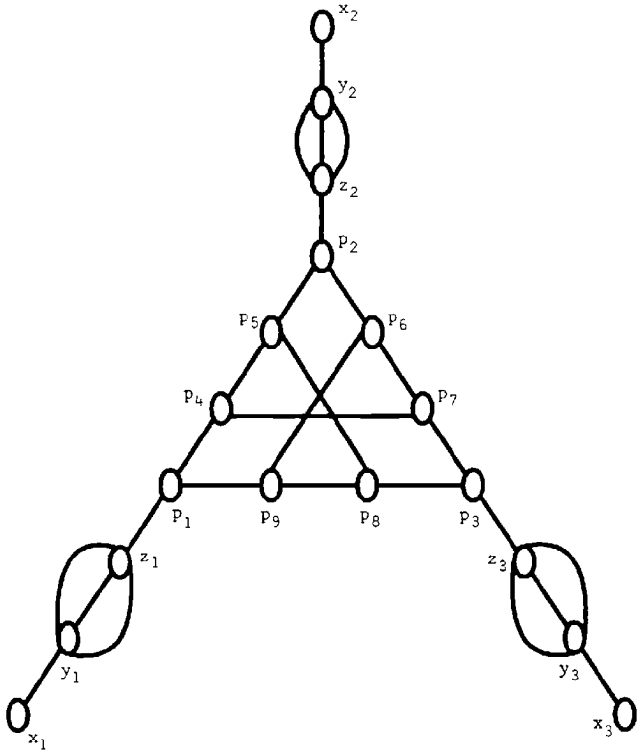


FIG. 2.1.

paths with an end vertex in x_i $(i = 1, 2, 3)$, we speak of an (s_1, s_2, s_3)-*cover*. Because of symmetry, we can assume that $1 \leq s_1 \leq s_2 \leq s_3$.

LEMMA 2.2. *If $k \geq 16(q+1) > 80$, then the sum of lengths of the paths and cycles in an (s_1, s_2, s_3)-cover of $H(k, q)$ is at least $16q + 10k + 12$. Equality can be achieved only for $(s_1, s_2, s_3) = (1, 2, 3)$.*

Proof. The paths $x_1y_1z_1p_1p_9p_6p_7p_4p_5p_8p_3z_3y_3x_3$, $x_2y_2z_2p_2p_5p_4p_1p_9p_8p_3z_3y_3x_3$, $x_2y_2z_2p_2p_6p_7p_3z_3y_3x_3$ and two cycles $y_1z_1y_1$ and $y_2z_2y_2$ form a $(1, 2, 3)$-cover of $H(k, q)$ with total length $16q + 10k + 12$.

Now suppose there exists an (s_1, s_2, s_3)-cover of total length $\leq 16q + 10k + 12$ such that $(s_1, s_2, s_3) \neq (1, 2, 3)$. Since $s_1 + s_2 + s_3$ is the total number of ends of the paths, $s_1 + s_2 + s_3$ is even and hence some s_i $(1 \leq i \leq 3)$ is even. Therefore one of the three paths between y_i and z_i is covered twice. If there exists a $j \in \{1, 2, 3\} \backslash \{i\}$ such that s_j is also even, then one of the paths between y_j and z_j is also covered twice and hence the total length of the path-cycle cover is at least $11k$, which is greater than $16q + 10k + 12$, a contradiction. Thus precisely one s_i is even.

A similar argument shows that the total length of the path-cycle cover is at least $11k$ if $s_3 \geq 5$. Hence $1 \leq s_1 \leq s_2 \leq s_3 \leq 4$.

Suppose now that $s_3 = 4$. Then either both of p_3p_8 and p_3p_7 are covered twice or one is covered three times. Since s_1 and s_2 are odd and every vertex in $\{p_1, p_2, \ldots, p_9\}$ is incident with an edge covered an even number of times, it follows that the total length of the path-cycle cover is at least $17q + 10k + 12$, a contradiction. Thus $1 \leq s_1 \leq s_2 \leq s_3 \leq 3$. This leaves the possibilities $(s_1, s_2, s_3) = (2, 3, 3)$ or $(1, 1, 2)$.

In the former case, the total length of the path-cycle cover is at least $16q + 10k + 16$, a contradiction.

Assume finally that $(s_1, s_2, s_3) = (1, 1, 2)$. It follows from Lemma 1 that the path-cycle cover restricted to the graph induced by $\{p_1, p_2, \ldots, p_9\}$ has total length at least $17q$. Hence the path-cycle cover has total length at least $17q + 10k + 8$, a contradiction. □

3. Cycle covers and the 3-edge-coloring problem. The 3-*edge-coloring* problem is the following: Given a cubic graph G (i.e., every vertex of G has degree 3), is it possible to assign a color in $\{1, 2, 3\}$ to each edge such that no two edges of the same color are incident with the same vertex? Holyer [2] proved that the 3-edge-color problem is *NP*-complete.

THEOREM 3.1. *The* 3*-edge-color problem can be reduced by a polynomial-time transformation to the question "Is* $\mathrm{cc}(G) \leq k$*?". Since the former is NP-complete, so is the latter.*

Proof. Let G be a cubic graph (which we would like to 3-edge-color). Let n be the number of vertices of G. We insert a vertex of degree 2 on every edge of G. Call the resulting graph G_1. If v is a vertex of degree 3 in G_1, then we delete v and replace it by a copy of $H(100,5)$ in such a way that x_1, x_2, and x_3 get degree 2 in the resulting graph, which we call M. We claim that G is 3-edge-colorable if and only if

$$\mathrm{cc}(M) \leq 1092n.$$

Every path-cycle cover of $H(100,5)$ has total length at least 1092 by Lemma 2. Hence $\mathrm{cc}(M) \geq 1092n$. Moreover, if equality holds, then each $H(100,5)$ is covered by a $(1, 2, 3)$-cover, and this results in a 3-edge-coloring of G. (An edge e in G is colored by i, where $i \in \{1, 2, 3\}$, and the new vertex of degree 2 on e is contained in precisely i cycles in the cycle cover of M.) Conversely, if G has a 3-edge-coloring, then we let $C_1, C_2, \ldots, C_r$ denote the cycles of color 1, 3 and $C'_1, C'_2, \ldots, C'_s$ denote the cycles of color 2, 3. Now the system $C_1, C_2, \ldots, C_r, C'_1, C'_1, C'_2, C'_2, \ldots, C'_s, C'_s$ covers each edge of color i precisely i times $(i = 1, 2, 3)$. By modifying this collection of cycles using the paths in the beginning of the proof of Lemma 2, we obtain a cycle cover of M of length $1092n$. □

REFERENCES

[1] J. A. BONDY, *Small cycle double covers of graphs*, in Cycles and Rays, G. Hahn, G. Sabidussi, and R. E. Woodrow, eds., Kluwer Academic Publishers, Dordrecht, the Netherlands, 1990, pp. 21–40.

[2] I. HOLYER, *The NP-completeness of edge-coloring*, SIAM J. Comput., 10 (1981), pp. 718–720.

[3] A. ITAI, R. J. LIPTON, C. H. PAPADIMITRIOU, AND M. RODEH, *Covering graphs with simple circuits*, SIAM J. Comput., 10 (1981), pp. 746–750.

[4] G. FAN, *Covering graphs by cycles*, SIAM J. Discrete Math., 5 (1992), pp. 491–496.

[5] C. ZHAO, *Smallest* $(1, 2)$*-Eulerian weight and shortest cycle covering*, J. Graph Theory, 18 (1994), pp. 153–160.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 678–692, June 1997

005

AN OPTIMAL ALGORITHM FOR SCANNING ALL SPANNING TREES OF UNDIRECTED GRAPHS*

AKIYOSHI SHIOURA†, AKIHISA TAMURA‡, AND TAKEAKI UNO§

Abstract. Let G be an undirected graph with V vertices and E edges. Many algorithms have been developed for enumerating all spanning trees in G. Most of the early algorithms use a technique called "backtracking." Recently, several algorithms using a different technique have been proposed by Kapoor and Ramesh (1992), Matsui (1993), and Shioura and Tamura (1993). They find a new spanning tree by exchanging one edge of a current one. This technique has the merit of enabling us to compress the whole output of all spanning trees by outputting only relative changes of edges. Kapoor and Ramesh first proposed an $O(N+V+E)$-time algorithm by adopting such a "compact" output, where N is the number of spanning trees. Another algorithm with the same time complexity was constructed by Shioura and Tamura. These are optimal in the sense of time complexity but not in terms of space complexity because they take $O(VE)$ space. We refine Shioura and Tamura's algorithm and decrease the space complexity from $O(VE)$ to $O(V+E)$ while preserving the time complexity. Therefore, our algorithm is optimal in the sense of both time and space complexities.

Key words. optimal algorithm, spanning trees, undirected graphs

AMS subject classifications. 05C30, 68R10

PII. S0097539794270881

1. Introduction.

Let G be an undirected graph with V vertices and E edges. A spanning tree of G is defined as a connected subgraph of G which contains all vertices but no cycle. In this paper, we consider the enumeration of all spanning trees in an undirected graph. Many algorithms for solving this problem have been developed, e.g., [7, 8, 4, 5, 6, 9], and these may be divided into several types.

The first type [7, 8, 4], to which many of the early algorithms belong, uses a technique called "backtracking." This is a useful technique for listing the kinds of subgraphs, e.g., cycles, paths, and so on. Gabow and Myers [4] refined the algorithms of Minty [7] and Read and Tarjan [8]. Their algorithm uses $O(NV+V+E)$ time and $O(V+E)$ space, where N is the number of all spanning trees. If we enumerate all spanning trees by outputting all edges of each spanning tree, their algorithm is optimal in terms of time and space complexities.

Recently, several algorithms [5, 6, 9] that use another technique have been developed. These algorithms find a new spanning tree by exchanging one pair of edges instead of backtracking. Furthermore, if we enumerate all spanning trees by outputting only relative changes of edges between spanning trees, we can compress the size of output to $\Theta(N+V)$, and hence the total time complexity may be reduced. In fact, Kapoor and Ramesh [5] proposed an $O(N+V+E)$ time and $O(VE)$-space algorithm by adopting such a "compact" output, which is optimal in the sense of time complexity. On the other hand, Matsui [6] developed an $O(NV+V+E)$-time and $O(V+E)$-space algorithm for enumerating all spanning trees explicitly, by applying the reverse-search scheme [3]. Reverse search is a scheme for general enumeration

* Received by the editors July 11, 1994; accepted for publication (in revised form) July 10, 1995. http://www.siam.org/journals/sicomp/26-3/27088.html

† Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1 Oh-okayama, Meguro-ku, Tokyo 152, Japan (shioura@is.titech.ac.jp).

‡ Department of Computer Science and Information Mathematics, University of Electro-Communications, 1-5-1 Chofugaoka, Chofu-shi, Tokyo 182, Japan (tamura@im.uec.ac.jp).

§ Department of Systems Science, Tokyo Institute of Technology, 2-12-1 Oh-okayama, Meguro-ku, Tokyo 152, Japan (uno@is.titech.ac.jp).

problems (see [1, 2]). Shioura and Tamura [9] also developed an algorithm generating a compact output with the same time and space complexities as the Kapoor–Ramesh algorithm by using the reverse-search technique. The Kapoor–Ramesh algorithm and the Shioura–Tamura algorithm, however, are not efficient in terms of space complexity because they take $O(VE)$ space.

The main aim of this paper is to obtain an algorithm that generates a compact output and is optimal in the sense of both time and space complexities by refining the Shioura–Tamura algorithm. When the process goes to a lower-level node of the computation tree of the original algorithm, some edge set can be efficiently divided without requiring extra information. However, in order to efficiently restore such an edge set when the process goes back to the higher-level node, the algorithm requires extra $O(E)$ information. Since the depth of the computation tree is $V-1$, it takes $O(VE)$ space. We propose a useful property for efficiently restoring the edge set and a technique for restoring it which uses extra $O(V)$ space in all, while the time complexity remains $O(N+V+E)$.

In section 2, we explain the technique for enumeration of spanning trees and compact outputs. In section 3, we define a nice child–parent relationship between spanning trees and propose a naïve algorithm. In section 4, we show some properties which are useful for efficient manipulation of data structures in our implementation. Our implementation is presented in section 5, and the time and space complexities are analyzed.

2. Compact output. Let G be an undirected graph (not necessary simple) with V vertices $\{v_1, \ldots, v_V\}$ and E edges $\{e_1, \ldots, e_E\}$. We define two types of edge sets which are necessary for our algorithm, so-called fundamental cuts and fundamental cycles. Let T be a spanning tree of G. Throughout this paper, we represent a spanning tree by its edge set of size $V-1$. For any edge $f \in T$, the deletion of f from T yields two connected components. The *fundamental cut* associated with T and f is defined as the set of edges connecting these components and is denoted by $Cut(T \backslash f)$. Likewise, we define the *fundamental cycle* associated with T and $g \notin T$ as the set of edges contained in the unique cycle of $T \cup g$. We will denote it as $Cyc(T \cup g)$. By definition, $T \backslash f \cup g$ is a spanning tree for any $f \in T$ and any $g \in Cut(T \backslash f)$. Similarly, for any $g \notin T$ and any $f \in Cyc(T \cup g)$, $T \cup g \backslash f$ is also a spanning tree. These properties are useful for enumerating spanning trees because by using fundamental cuts or cycles, we can construct a different spanning tree from a given one by exchanging exactly one edge.

Given a graph G, let $\mathcal{S}(G) = (\mathcal{T}, \mathcal{A})$ be the graph whose vertex set $\mathcal{T}$ is the set of all spanning trees of G and whose edge set $\mathcal{A}$ consists of all pairs of spanning trees which are obtained from each other by exchanging exactly one edge using some fundamental cut or cycle. For example, the graph $\mathcal{S}(G_1)$ of the left one, G_1, is shown in Figure 2.1.

Our algorithm finds all spanning trees of G by implicitly traversing some spanning tree $\mathcal{D}$ of $\mathcal{S}(G)$. In order to output all $(V-1)$ edges of each spanning tree, $\Theta(|\mathcal{T}| \cdot V) = \Theta(N \cdot V)$ time is required. However, if we output all edges of the first spanning tree and then only the sequence of exchanged edge pairs of G obtained by traversing $\mathcal{D}$, we need only $\Theta(|\mathcal{T}| + V) = \Theta(N+V)$ time because $|\mathcal{D}| = |\mathcal{T}|-1$ and exactly two edges of G are exchanged for each edge of $\mathcal{D}$. Furthermore, by scanning such a "compact" output, one can construct all spanning trees. Since we adopt such a compact output, it becomes desirable to find the next spanning tree from a current one efficiently in constant time.

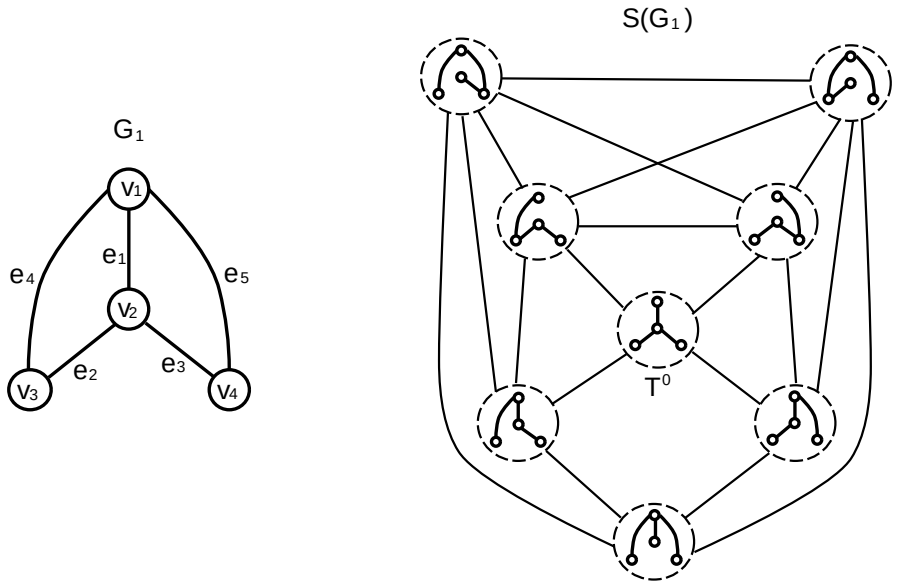


FIG. 2.1. *Graph G_1 and graph $\mathcal{S}(G_1)$.*

3. Basic ideas and the naïve algorithm. In this section, we explain the basic ideas and the naïve algorithm.

We define the total orders over the vertex set $\{v_1, \ldots, v_V\}$ and the edge set $\{e_1, \ldots, e_E\}$ of G by their indices as $v_1 < v_2 < \cdots < v_V$ and $e_1 < e_2 < \cdots < e_E$. Particularly, we call the smallest vertex v_1 the *root*. For each edge e, we call the smaller incident vertex the *tail*, denoted by $\partial^+ e$, and call the larger one the *head*, denoted by $\partial^- e$. Relative to a spanning tree T of G, if the unique path in T from the vertex v to the root v_1 contains a vertex u, then u is called an *ancestor* of v and v is a *descendant* of u. Similarly, for two edges e and f in T, we call e an *ancestor* of f and f a *descendant* of e if the unique path in T from f to the root v_1 contains e. A "depth-first spanning" tree of G is a spanning tree which is found by some depth-first search of G. It is known that a *depth-first spanning tree* is defined as a spanning tree such that for each edge of G, its one incidence vertex is an ancestor of the other.

In our algorithm, we make several assumptions regarding the vertex set and the edge set of G.

ASSUMPTION 1. *T^0 is a depth-first spanning tree of G.*

ASSUMPTION 2. $T^0 = \{e_1, \ldots, e_{V-1}\}$.

ASSUMPTION 3. *Any edge in T^0 is smaller than its proper descendants.*

ASSUMPTION 4. *Each vertex v is smaller than its proper descendants relative to T^0.*

ASSUMPTION 5. *For any two edges $e, f \notin T^0$, if $e < f$, then $\partial^+ e \leq \partial^+ f$.*

Vertices and edges of graph G_2 in Figure 3.1 satisfy these assumptions. In fact, one can find T^0 and sort vertices and edges of G in $O(V+E)$ time so that G satisfies the above assumptions by applying Tarjan's depth-first search [10]. We note that Assumptions 1, 2, and 3 are sufficient for the correctness of our algorithm. However, we further need Assumptions 4 and 5 for an efficient implementation.

For any nonempty subset S of $\{e_1, \ldots, e_E\}$, $\mathrm{Min}(S)$ denotes the smallest edge in

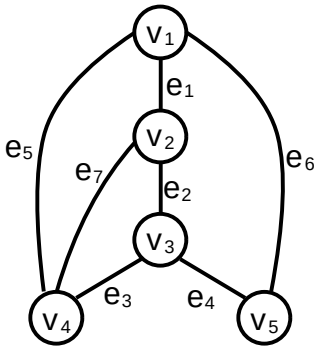


FIG. 3.1. *Graph* G_2.

S. For convenience, we assume that $\mathrm{Min}(\emptyset) = e_V$.

LEMMA 3.1 (see [9]). *Under Assumptions* 1 *and* 3*, for any spanning tree* $T^c \neq T^0$*, if* $f = \mathrm{Min}(T^0 \setminus T^c)$*, then* $Cyc(T^c \cup f) \cap Cut(T^0 \backslash f) \setminus f$ *contains exactly one edge.*

Proof. The set $T^0 \setminus f$ has exactly two components, one containing $\partial^- f$ and the other containing $\partial^+ f$. Therefore, the unique path $Cyc(T^c \cup f) \setminus f$ from $\partial^- f$ to $\partial^+ f$ in T^c contains at least one edge in $Cut(T^0 \backslash f)$. Hence $Cyc(T^c \cup f) \cap Cut(T^0 \backslash f) \setminus f \neq \emptyset$.

Since T^0 is a depth-first spanning tree, we may assume without loss of generality that the head of any edge is a descendant of its tail relative to T^0. Let e be the first edge from $\partial^- f$ on the path such that $e \in Cut(T^0 \backslash f)$. Then the head $\partial^- e$ is a descendant of $\partial^- f$ relative to T^0, and the tail $\partial^+ e$ is an ancestor of $\partial^+ f$. From Assumption 3 and the minimality of f, $\partial^+ e$ and $\partial^+ f$ are connected in $T^c \cap T^0$. Thus there is no edge contained in $Cut(T^0 \backslash f)$ between $\partial^+ e$ and $\partial^+ f$ in the path $Cyc(T^c \cup f) \setminus f$. Hence e is the only edge in $Cyc(T^c \cup f) \setminus f$ and $Cut(T^0 \backslash f)$. □

Consider the graph G_2 of Figure 3.1. Here let $T^0 = \{e_1, e_2, e_3, e_4\}$ and $T^c = \{e_4, e_5, e_6, e_7\}$. In graph G_2,

$$\begin{aligned} f &= \mathrm{Min}\{e_1, e_2, e_3\} = e_1, \\ Cyc(T^c \cup f) &= \{e_1, e_5, e_7\}, \\ Cut(T^0 \backslash f) &= \{e_1, e_5, e_6\}. \end{aligned}$$

Therefore, $Cyc(T^c \cup f) \cap Cut(T^0 \backslash f) \setminus f = \{e_5\}$.

Given a spanning tree $T^c \neq T^0$ and the edge $f = \mathrm{Min}(T^0 \setminus T^c)$, let g be the unique edge in $Cyc(T^c \cup f) \cap Cut(T^0 \backslash f) \setminus f$. Clearly, $T^p = T^c \cup f \backslash g$ is a spanning tree. We call T^p the *parent* of T^c and T^c a *child* of T^p. Lemma 3.1 guarantees that each spanning tree other than T^0 has a unique parent. Since $|T^p \cap T^0| = |T^c \cap T^0| + 1$ holds, T^0 is the ancestor of all spanning trees. For the graph G_1 in Figure 2.1, all child–parent pairs are shown by the arrows in Figure 3.2. Each arrow goes from a child to its parent. We can see that all arrows construct a spanning tree of $\mathcal{S}(G_1)$ rooted at T^0.

Let $\mathcal{D}$ be the spanning tree of $\mathcal{S}(G)$ consisting of all child–parent pairs of spanning trees. Our algorithm implicitly traverses $\mathcal{D}$ from T^0 by recursively scanning all children of a current spanning tree. Thus we must find all children of a given spanning tree, if they exist. The next lemma gives a useful idea for this.

LEMMA 3.2 (see [9]). *Let* T^p *be an arbitrary spanning tree of* G*, and let* f *and* g *be two distinct edges. Under Assumptions* 1*,* 2*, and* 3*,* $T^c = T^p \backslash f \cup g$ *is a child of* T^p

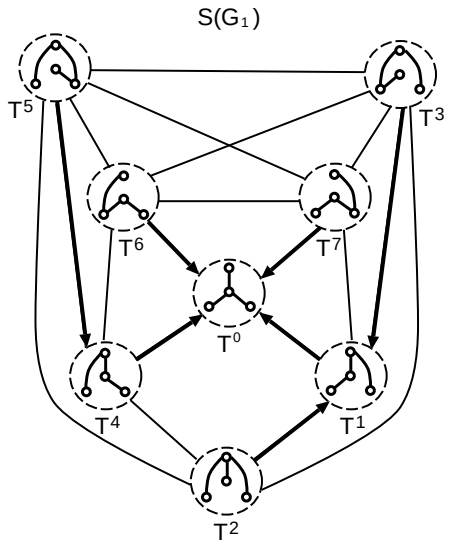


FIG. 3.2. *Child–parent relations in* $\mathcal{S}(G_1)$.

if and only if f *and* g *satisfy the following conditions:*

$$f < \mathrm{Min}(T^0 \setminus T^p) \quad \textit{and} \quad g \in Cut(T^p \backslash f) \cap Cut(T^0 \backslash f) \setminus f. \tag{3.1}$$

Proof. Under Assumptions 1 and 3, T^c is a child of T^p if and only if the following conditions hold:

$$T^c \text{ is a spanning tree different from } T^0; \tag{3.2}$$

$$f' = \mathrm{Min}(T^0 \setminus T^c) \quad \text{and} \quad g' \in Cyc(T^c \cup f') \cap Cut(T^0 \backslash f') \setminus f', \tag{3.3}$$

$$T^p = T^c \cup f' \backslash g'. \tag{3.4}$$

We first show that $f = f'$ and $g = g'$. From (3.2), (3.3), and (3.4), T^c and T^p are different spanning trees. Assume to the contrary that $f \notin T^p$; then $T^p \setminus f = T^p$. Since T^c is a spanning tree and $f \neq g$, we have $g \in T^p$ and $T^c = T^p \backslash f \cup g = T^p$, which is a contradiction. Thus $f \in T^p$ and $g \notin T^p$. From (3.4), $T^p = \{T^p \backslash f \cup g\} \cup f' \backslash g'$, and hence $f = f'$ and $g = g'$ must hold.

Conditions (3.2), (3.3), and (3.4) imply

$$f \in T^p \cap T^0 \quad \text{and} \quad g \notin T^p \cup T^0. \tag{3.5}$$

On the other hand, under Assumption 2, (3.1) implies (3.5). Moreover, (3.1) and (3.5) imply (3.2) and (3.4). All we have to do is to show that (3.1) and (3.3) are equivalent under conditions (3.2), (3.4), and (3.5).

From the definition of T^c and (3.5), $T^0 \setminus T^c = T^0 \setminus (T^p \backslash f \cup g) = (T^0 \setminus T^p) \cup \{f\}$. Hence $\mathrm{Min}(T^0 \setminus T^c) = \mathrm{Min}(\mathrm{Min}(T^0 \setminus T^p) \cup \{f\})$. This implies that $f = \mathrm{Min}(T^0 \setminus T^c)$ if and only if $f < \mathrm{Min}(T^0 \setminus T^p)$. Since T^p and $T^c = T^p \backslash f \cup g$ are distinct, $g \in Cyc(T^c \cup f)$ is equivalent to $g \in Cut(T^p \backslash f)$. Therefore, the second condition of (3.1) is equivalent to the second condition of (3.3). ∎

Let e_k be the largest edge less than $\mathrm{Min}(T^0 \setminus T^p)$. From Lemma 3.2, we can find all children of T^p if we know the edge sets $Cut(T^p \backslash e_j) \cap Cut(T^0 \backslash e_j) \setminus e_j$ for $j = 1, 2, \ldots, k$. Consider the graph $G = G_1$ defined in Figure 2.1 and $T^p = T^1$ (see Figure 3.2). In this case, e_1 and e_2 are the only edges smaller than $\mathrm{Min}(T^0 \setminus T^1) = e_3$ and

$$Cut(T^1 \backslash e_2) \cap Cut(T^0 \backslash e_2) \setminus e_2 = \{e_2, e_4\} \cap \{e_2, e_4\} \setminus e_2 = \{e_4\},$$
$$Cut(T^1 \backslash e_1) \cap Cut(T^0 \backslash e_1) \setminus e_1 = \{e_1, e_3, e_4\} \cap \{e_1, e_4, e_5\} \setminus e_1 = \{e_4\}.$$

Therefore, T^1 has only the two children, $T^1 \backslash e_2 \cup e_4$ and $T^1 \backslash e_1 \cup e_4$.

In the rest of paper, we abbreviate $Cut(T^p \backslash e_j) \cap Cut(T^0 \backslash e_j) \setminus e_j$ as $Entr(T^p, e_j)$ on the grounds that any edge in $Cut(T^p \backslash e_j) \cap Cut(T^0 \backslash e_j) \setminus e_j$ can be "entered" into T^p in place of e_j. From the above consideration, we can construct the following algorithm.

ALGORITHM all-spanning-trees(G);
input: a graph G with a vertex set $\{v_1, \ldots, v_V\}$ and an edge set $\{e_1, \ldots, e_E\}$;
begin
by using a depth-first search,
• find a depth-first spanning tree T^0 of G,
• sort vertices and edges to satisfy Assumptions 2, 3, 4, and 5;
output("$e_1, e_2, \cdots, e_{V-1}, tree,$") ;{output T^0}
find-children(T^0,$V-1$);
end.

PROCEDURE find-children(T^p,k);
input: a spanning tree T^p and an integer k with $e_k < \mathrm{Min}(T^0 \setminus T^p)$;
begin
if $k \leq 0$ **then** return;
for each $g \in Entr(T^p, e_k)$ **do begin**
{output all children of T^p not containing e_k}
$T^c := T^p \backslash e_k \cup g$;
output("$-e_k, +g, tree,$");
find-children(T^c,$k-1$); {find the children of T^c}
output("$-g, +e_k,$");
end;
find-children(T^p,$k-1$); {find the children of T^p not containing e_{k-1}}
end.

In this algorithm, procedure find-children( ) finds all children of each spanning tree. When it is called with two arguments T^p and k, it finds all children of T^p not containing an edge e_k. Whenever it finds such a child T^c, it recursively calls itself again to find all children of T^c. In this stage, arguments are set to T^c and $k-1$ because if $k > 1$, then e_{k-1} becomes the largest edge less than $\mathrm{Min}(T^0 \setminus T^c)$. If all children of T^p not containing e_k have been found, it recursively calls itself again to find all children of T^p not containing e_{k-1}. In this case, arguments are T^p and $k-1$. Initially, algorithm all-spanning-trees(G) calls find-children( ) with arguments T^0 and $V-1$, and all spanning trees of G are found. Figure 3.3 shows the enumeration tree of spanning trees in graph G_1.

THEOREM 3.3 (see [9]). *Algorithm* all-spanning-trees( ) *outputs each spanning tree exactly once.*

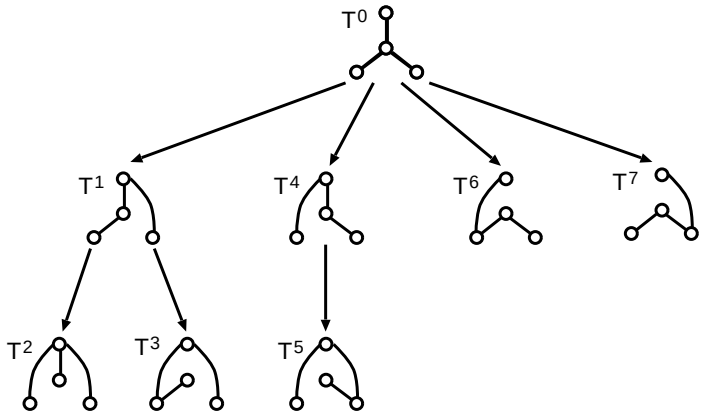


FIG. 3.3. *Enumeration tree of spanning trees in* G_1.

Proof. From Lemma 3.2, every spanning tree different from T^0 is output once for each time its parent is output. From Lemma 3.1, for any spanning tree T^c other than T^0, its parent always exists and is uniquely determined. Since T^0 is the ancestor of all spanning trees, the algorithm outputs each spanning tree exactly once. □

4. Manipulating data structures. In our algorithm, we define each state when we find all children of T^p not containing e_k by a pair (T^p, k). When we call procedure find-children(T^p, k), the current state becomes (T^p, k), and if we find a child T^c of T^p not containing e_k, the state moves to $(T^c, k-1)$. After all children of T^p not containing e_k have been found, the state moves to $(T^p, k-1)$. At the state (T^p, k), the entering edge set $Entr(T^p, e_k)$ is required to output all children of T^p not containing e_k. After the state moves to $(T^c, k-1)$ (or $(T^p, k-1)$), the entering edge set $Entr(T^c, e_{k-1})$ (or $Entr(T^p, e_{k-1})$) is required for the first time. The key point is finding an entering edge set $Entr(T^c, e_{k-1})$ (or $Entr(T^p, e_{k-1})$) efficiently. To construct an entering edge set efficiently, our implementation maintains the edge sets $Can(e_j; T^p, k)$ for $j = 1, \ldots, k$ defined below. Let T^p be a spanning tree and k be a positive integer with $e_k < \mathrm{Min}(T^0 \setminus T^p)$. For each edge e_j $(j = 1, \ldots, k)$, we define $Can(e_j; T^p, k)$ by

$$Can(e_j; T^p, k) = Entr(T^p, e_j) \setminus \bigcup_{h=j+1}^{k} Entr(T^p, e_h). \tag{4.1}$$

Here we use this notation in the sense that $Can(e_j; T^p, k)$ is a set of "candidates" of the entering edges $Entr(T^p, e_j)$ for a leaving edge e_j at the state (T^p, k). We can find $Entr(T^p, e_k)$ very easily by maintaining $Can(e_j; T^p, k)$ for $j = 1, \ldots, k$ because $Can(e_k; T^p, k) = Entr(T^p, e_k)$ from the definition in (4.1). When we find a child T^c of T^p, we update $Can(e_j; T^p, k)$ for $j = 1, \ldots, k$ to $Can(e_j; T^c, k-1)$ for $j = 1, \ldots, k-1$. On the other hand, after we have found all children of T^p not containing e_{k-1}, we construct $Can(e_j; T^p, k-1)$ for $j = 1, \cdots, k-1$ from $Can(e_j; T^p, k)$ for $j = 1, \ldots, k$. The efficiency of our implementation depends on how to maintain $Can(*; *, *)$ efficiently.

Figure 4.1 shows the states and edge sets $Can(*; *, *)$ during the enumeration of all spanning trees of G_1 in Figure 2.1. For example, at the initial state $(T^0, 3)$,

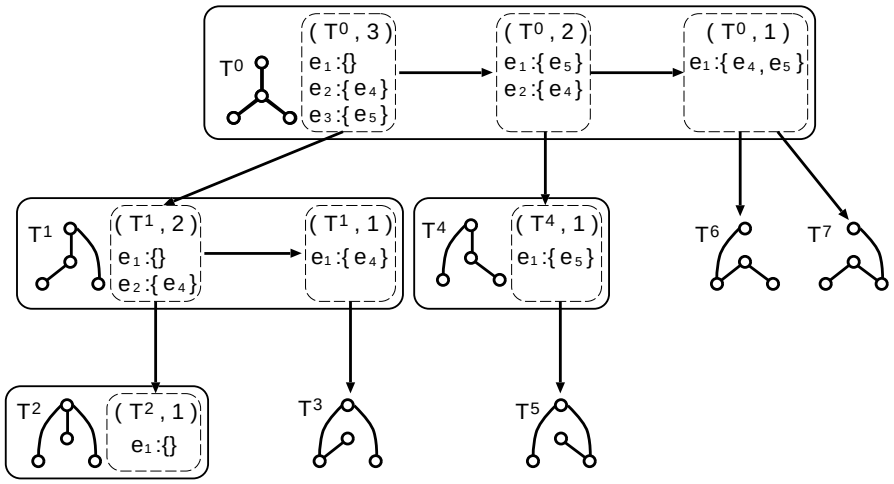


FIG. 4.1. *Movement of the state and* $Can(*;*,*)$.

$$Can(e_1;T^0,3)=\emptyset,$$
$$Can(e_2;T^0,3)=\{e_4\},$$
$$Can(e_3;T^0,3)=\{e_5\}.$$

At the succeeding states $(T^1,2)$ and $(T^0,2)$,

$$Can(e_1;T^1,2)=\emptyset,$$
$$Can(e_2;T^1,2)=\{e_4\},$$

and

$$Can(e_1;T^0,2)=\{e_5\},$$
$$Can(e_2;T^0,2)=\{e_4\}.$$

Here we consider how to maintain such edge sets. First, we show that the initial edge sets $Can(e_j;T^0,V-1)$ for $j=1,\ldots,V-1$ can be found easily.

LEMMA 4.1 (see [9]). *Under Assumptions* 1, 2, 3, *and* 4,

$$Can(e_j;T^0,V-1)=\{e \mid e\notin T^0,\ \partial^+e\le\partial^+e_j \text{ and } \partial^-e=\partial^-e_j\} \quad (j=1,\ldots,V-1) \tag{4.2}$$

Proof. Since $Entr(T^0,e_j)=Cut(T^0\backslash e_j)\setminus e_j$, $Can(e_j;T^0,V-1)$ can be written as

$$Can(e_j;T^0,V-1)=\left[Cut(T^0\backslash e_j)\setminus e_j\right]\setminus\bigcup_{h=j+1}^{V-1}\left[Cut(T^0\backslash e_h)\setminus e_h\right].$$

Under Assumptions 1 and 4, an edge $e\notin T^0$ belongs to $Cut(T^0\backslash e_j)$ if and only if ∂^-e is a descendant of ∂^-e_j and ∂^+e is an ancestor of ∂^+e_j relative to T^0. In addition, under Assumption 3, for $e\notin T^0$, e_j is the largest edge with $e\in Cut(T^0\backslash e_j)$ if and only if $\partial^-e=\partial^-e_j$ and $\partial^+e\le\partial^+e_j$. ∎

From Lemma 4.1, we can find $Can(e_j; T^0, V-1)$ for $j = 1, \ldots, V-1$ in $O(V+E)$ time by applying a depth-first search.

LEMMA 4.2. *For any spanning tree T^p and any positive integer k with $e_k < \mathrm{Min}(T^0 \setminus T^p)$, let g be an arbitrary edge in $Entr(T^p, e_k) \cup \{e_k\}$. Under Assumptions 1, 2, 3, and 4, the following relation holds for a spanning tree $T = T^p \backslash e_k \cup g$ and an edge e_j with $j < k$:*

$$Entr(T, e_j) = \begin{cases} Entr(T^p, e_j) & \text{if } e_j \in A, \\ Entr(T^p, e_j) \setminus Entr(T^p, e_k) & \text{otherwise,} \end{cases} \tag{4.3}$$

where A is the set of ancestors of the edge e_t in T^0 with $\partial^- e_t = \partial^+ g$ if it exists; otherwise, $A = \emptyset$.

Proof. We note that if $g \in Entr(T^p, e_k)$, then T is a child of T^p and if $g = e_k$, then $T = T^p$. Each descendant of $\partial^- e_k$ relative to T^p is a descendant of $\partial^- g$ relative to T, and vice versa. Therefore, for any $e_j \in A$, $Entr(T, e_j) = Entr(T^p, e_j)$. If $e_j \notin A$ is an ancestor of e_k, then $Entr(T, e_j) \subseteq Entr(T^p, e_j)$. More precisely, for any edge $e \in Entr(T^p, e_j)$ such that $\partial^- e$ is a descendant of $\partial^- e_k$ relative to T^p, e does not belong to $Entr(T, e_j)$, and the other edges obviously belong to $Entr(T, e_j)$. That is, $Entr(T, e_j) = Entr(T^p, e_j) \setminus Entr(T^p, e_k)$. If e_j is not an ancestor of e_k, $Entr(T, e_j) = Entr(T^p, e_j) = Entr(T^p, e_j) \setminus Entr(T^p, e_k)$ holds because $Entr(T^p, e_j) \cap Entr(T^p, e_k) = \emptyset$. □

LEMMA 4.3 (see [9]). *Let T^p be a spanning tree and let k be a positive integer with $e_k < \mathrm{Min}(T^0 \setminus T^p)$. Under Assumptions 1, 2, 3, and 4, for any edge $g \in Can(e_k; T^p, k) \cup \{e_k\}$ and for a spanning tree $T = T^p \backslash e_k \cup g$, the following relation holds*:

$$Can(e_j; T, k-1) = \begin{cases} Can(e_j; T^p, k) \cup [Can(e_k; T^p, k) \cap \{e \mid \partial^+ e < \partial^+ g\}] \\ \qquad \text{if } \partial^- e_j = \partial^+ g, \\ Can(e_j; T^p, k) \quad \text{if } \partial^- e_j \neq \partial^+ g. \end{cases} \tag{4.4}$$

Proof. From the assumptions, for two edges e and f with $e, f < \mathrm{Min}(T^0 \setminus T^p)$, e is an ancestor of f relative to T^0 if and only if e is an ancestor of f relative to T^p, so we will omit the phrase "relative to T^0 (or T^p)" for such edges. Let e_t be the edge with $\partial^- e_t = \partial^+ g$ if it exists, and let A be the set of edges in T^0 which are ancestors of e_t if e_t exists; otherwise, $A = \emptyset$. We prove (4.4) by using relation (4.3).

Case 1. If $e_j \notin A$, then

$$\begin{aligned} &Can(e_j; T, k-1) \\ &\quad = [Entr(T^p, e_j) \setminus Entr(T^p, e_k)] \\ &\qquad \setminus \left[\bigcup_{h=j+1,\, e_h \notin A}^{k-1} (Entr(T^p, e_h) \setminus Entr(T^p, e_k)) \cup \bigcup_{h=j+1,\, e_h \in A}^{k-1} Entr(T^p, e_h) \right] \\ &\quad = Entr(T^p, e_j) \setminus \bigcup_{h=j+1}^{k} Entr(T^p, e_h) = Can(e_j; T^p, k). \end{aligned}$$

Case 2. If $e_j \in A$, then

$$\begin{aligned} &Can(e_j; T, k-1) \\ &\quad = Entr(T^p, e_j) \setminus \left[\bigcup_{h=j+1,\, e_h \notin A}^{k-1} (Entr(T^p, e_h) \setminus Entr(T^p, e_k)) \cup \bigcup_{h=j+1,\, e_h \in A}^{k-1} Entr(T^p, e_h) \right] \end{aligned}$$

$$= Can(e_j; T^p, k) \bigcup \left[Entr(T^p, e_j) \cap \left(Entr(T^p, e_k) \setminus \bigcup_{h=j+1,\, e_h \in A}^{k-1} Entr(T^p, e_h) \right) \right].$$

If $e_j = e_t$, then there is no edge e_h with $j < h < k$ and $e_h \in A$. Therefore,

$$\begin{aligned} Can(e_j; T, k-1) &= Can(e_j; T^p, k) \bigcup [Entr(T^p, e_j) \cap Entr(T^p, e_k)] \\ &= Can(e_j; T^p, k) \bigcup \left[Can(e_k; T^p, k) \cap \{e \mid \partial^+ e < \partial^- e_t\} \right]. \end{aligned}$$

If e_j is a proper ancestor of e_t, then $Entr(T^p, e_j) \cap Entr(T^p, e_k) \subseteq Entr(T^p, e_t)$, and e_t satisfies $j < t < k$ and $e_t \in A$. Hence $Can(e_j; T, k-1) = Can(e_j; T^p, k)$. □

Lemma 4.3 guarantees that at most one of the sets $Can(*; T^p, k)$ is updated when we want to find all children of T^c or all children of T^p containing e_k. In Figure 4.1, when the state moves from $(T^0, 3)$ to $(T^0, 2)$, e_1 is the edge such that $\partial^- e_1 = \partial^+ e_3$ and the following equations hold:

$$\begin{aligned} Can(e_2; T^0, 2) &= Can(e_2; T^0, 3) \;=\; \{e_4\} \\ Can(e_1; T^0, 2) &= Can(e_1; T^0, 3) \cup \left[Can(e_3; T^0, 3) \cap \{e \mid \partial^+ e < \partial^+ e_3\} \right] \\ &= \emptyset \cup \left[\{e_5\} \cap \{e \mid \partial^+ e < v_2\} \right] \;=\; \{e_5\}. \end{aligned}$$

On the other hand, when the state moves from $(T^0, 3)$ to $(T^1, 2)$, no candidate edge set is updated because there is no edge with $\partial^- e_t = \partial^+ e_5$:

$$\begin{aligned} Can(e_2; T^1, 2) &= Can(e_2; T^0, 3) = \{e_4\}, \\ Can(e_1; T^1, 2) &= Can(e_1; T^0, 3) = \emptyset. \end{aligned}$$

In our implementation, we use the global variables `candi`$(*)$ and `leave`. At the state (T^p, k), variable `candi`(e_j) $(j{=}1, \ldots, k)$ represents the edge set $Can(e_j; T^p, k)$ and variable `leave` represents the edge set $\{e_j \mid j \le k \text{ and } Can(e_j; T^p, k) \neq \emptyset\}$. We can check in constant time whether or not the current spanning tree has children by checking to see if `leave` $\neq \emptyset$. Suppose that each edge set is represented as an ascending ordered list realized by a doubly linked list. We also use (i) a data structure for a given graph G so that two incidence vertices of any edge are found in constant time and (ii) a data structure for the initial spanning tree T^0 so that for any vertex v other than the root, the unique edge e with $\partial^- e = v$ is found in constant time. Recall that graph G satisfies the following assumption.

ASSUMPTION 5. *For any two edges $e, f \notin T^0$, if $e < f$, then $\partial^+ e \le \partial^+ f$.*

From this assumption, one can find the edge set $Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^+ g\}$ by searching the ordered list `candi`(e_k) from the beginning. Thus we can complete this in time proportional to the size of this edge set. Merging two edge sets can be executed in time proportional to the sum of the size of two edge sets. Therefore, it takes $O(|Can(e_t; T^p, k)| + |Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^+ g\}|)$ time to update edge sets `candi`$(*)$ when the current state (T^p, k) goes to a succeeding state $(T, k{-}1)$. If `candi`(e_t) changes from empty to nonempty, then we must insert an edge e_t into `leave`. Since `leave` is an ascending ordered list, we can complete it in $O(|\{e \in$ `leave` $| e < e_t\}|) = O(|\{e_j | j < t \text{ and } Can(e_j; T^p, k) \neq \emptyset\}|)$ time.

On the other hand, when the state goes back from $(T, k{-}1)$ to (T^p, k), we must reconstruct $Can(*; T^p, k)$ from $Can(*; T, k{-}1)$. To do this, we must restore the edges $Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^+ g\}$ from `candi`(e_t) to `candi`(e_k). In the Shioura–Tamura algorithm [9], such a restoration is efficiently executed by recording $Can(e_k; T^p, k) \cap$

$\{e|\partial^+e<\partial^+g\}$ before state (T^p,k) goes to $(T,k-1)$. However, this idea requires $O(VE)$ extra space since the depth of recursive calls of the algorithm is $O(V)$. In the rest of this section, we discuss our idea for reducing extra space.

Let $Head(e_j;T^p,k)$ denote the head set of edges contained in $Can(e_j;T^p,k)$. Then we have the following result.

LEMMA 4.4. *Under Assumptions* 1, 2, 3, *and* 4, *all head sets* $Head(e_j;T^p,k)$ *for* $j=1,\ldots,k$ *are mutually disjoint at any state* (T^p,k).

Proof. From Lemma 4.1, $Head(e_j;T^0,V-1)=\{\partial^-e_j\}$ at the initial state $(T^0,V-1)$ if $Can(e_j;T^0,V-1)$ is nonempty. Thus the assertion is true at the initial state.

We assume that the lemma holds at state (T^p,k) and prove that this holds at the next state $(T^p\backslash e_k\cup g,k-1)$, where $g\in Can(e_k;T^p,k)\cup\{e_k\}$. From Lemma 4.3, the following relation holds:

$$Head(e_j;T,k-1)=\begin{cases}Head(e_j;T^p,k)\cup HS & \text{if } \partial^-e_j=\partial^+g,\\ Head(e_j;T^p,k) & \text{if } \partial^-e_j\neq\partial^+g,\end{cases} \tag{4.5}$$

where HS is the head set of all edges in $Can(e_k;T^p,k)\cap\{e|\partial^+e<\partial^+g\}$. Because $HS\subseteq Head(e_k;T^p,k)$ and each $Head(e_j;T^p,k)$ for $j=1,\ldots,k-1$ does not intersect HS, all head sets $Head(e_j;T^p,k-1)$ for $j=1,\ldots,k-1$ are mutually disjoint. ∎

By Lemma 4.4, the head set HS of edges in $Can(g;T^p,k)\cap\{e|\partial^+e<\partial^+g\}$ has no intersection with any head set $Head(e_j;T^p,k)$ $(j=1,\ldots,k-1)$. Hence if we can find HS before restoring $\texttt{candi}(*)$, it is easy to pick up the edges $Can(e_k;T^p,k)\cap\{e|\partial^+e<\partial^+g\}=\{e\in Can(e_t;T,k-1)|\partial^-e\in HS\}$ from $Can(e_t;T,k-1)$.

In Figure 4.1, when the state goes back from $(T^0,1)$ to $(T^0,2)$, all edges in $Can(e_2;T^0,2)\cap\{e|\partial^+e<\partial^+e_2\}=\{e_4\}$ must be restored from $\texttt{candi}(e_1)=Can(e_1;T^0,1)=\{e_4,e_5\}$ to $\texttt{candi}(e_2)$. The head set of $Can(e_2;T^0,2)\cap\{e|\partial^+e<\partial^+e_2\}$ is equal to $\{v_3\}$. In this case, $e_4\in\texttt{candi}(e_1)$ is put back into $\texttt{candi}(e_2)$ to reconstruct $Can(e_2;T^0,2)$.

Our implementation uses the global variables $\texttt{head}(*)$ to represent each $Head(e_j;T^p,k)$ for $j=1,\ldots,k$ at state (T^p,k). Suppose that each head set is represented by a (not necessarily ascending) doubly linked list. From Lemma 4.4, we require $O(V)$ space for manipulating these head sets.

Now we describe two procedures for manipulating the data structures $\texttt{candi}(*)$, `leave`, and $\texttt{head}(*)$ when the current state (T^p,k) goes to a succeeding state $(T,k-1)$ or $(T,k-1)$ goes back to (T^p,k), respectively. The procedure for the first case is shown below.

PROCEDURE update-data-structure(e_k,g);
{the current state (T^p,k) goes to a succeeding state $(T,k-1)=(T^p\backslash e_k\cup g,k-1)$}
begin
  e_t := the edge in T^0 with $\partial^-e_t=\partial^+g$ if it exists, otherwise return;
  move $\{e\in\texttt{candi}(e_k)|\partial^+e<\partial^+g\}$ from $\texttt{candi}(e_k)$ to $\texttt{candi}(e_t)$;
  if $\texttt{candi}(e_t)$ changes from empty to nonempty **then** insert e_t into `leave`;
  HS := the head set of the edges in $\{e\in\texttt{candi}(e_k)|\partial^+e<\partial^+g\}$;
  for each maximal sublist of consecutive elements of HS in $\texttt{head}(e_k)$ **do begin**
    record the first element of the sublist and its position in $\texttt{head}(e_k)$ on a stack;
    delete the sublist from $\texttt{head}(e_k)$;
    add this to the end of $\texttt{head}(e_t)$;
  end;
  record the position of the first element of HS in $\texttt{head}(e_t)$ on a stack;
end.

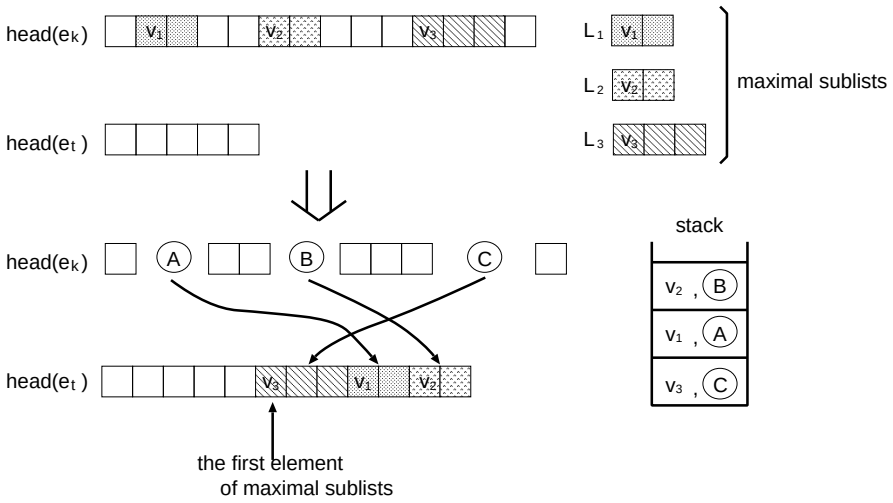


FIG. 4.2. *Update of* `head`$(*)$.

When the state changes from (T^p, k) to $(T, k-1)$, we must move the head set HS of all edges in $Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^+ g\}$ from `head`(e_k) to `head`(e_t). At this time, we do not move each element of HS one by one but move each maximal sublist of consecutive elements of HS in `head`(e_k) to `head`(e_t) as Figure 4.2. Then the extra space for recording positions of such maximal sublists is $O(V)$ in all because the number of maximal sublists is at most $|\texttt{head}(e_k) \setminus HS| + 1$, and `head`$(e_k) \setminus HS$ is unchanged until the state comes back to (T^p, k). It is easy to manipulate `head`$(*)$ in the same time as `candi`$(*)$ because $|HS| \leq |Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^+ g\}|$. Here we omit details. Thus the time complexity of the procedure is $O(|Can(e_t; T^p, k)| + |Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^+ g\}| + |\{e_j | j < t \text{ and } Can(e_j; T^p, k) \neq \emptyset\}|)$.

The second procedure restores data structures in the following way.

PROCEDURE restore-data-structure(e_k,g);
{the state $(T^p \backslash e_k \cup g, k-1)$ goes back to (T^p, k)}
begin
  e_t := the edge in T^0 with $\partial^- e_t = \partial^+ g$ if it exists, otherwise return;
  find HS by the record of the position of its first element in `head`(e_t);
  delete HS from `head`(e_t);
  move $\{e \in$ `candi`$(e_t) | \partial^- e \in HS\}$ from `candi`(e_t) to the beginning of `candi`(e_k);
  if `candi`(e_t) changes from nonempty to empty **then** delete e_t from `leave`;
  move each sublist in HS to the correct place in `head`(e_k)
    by using records on a stack;
end.

Since we recorded the first element of head vertices which were added to `head`(e_t), we can find HS in constant time. For each edge in `candi`(e_t), we can check in constant time whether it is in HS by marking all elements of HS in advance. Hence we can restore `candi`$(*)$ in $O(|Can(e_t; T, k-1)|) = O(|Can(e_t; T^p, k)| + |\{e \in Can(e_k; T^p, k) | \partial^+ e < \partial^+ g\}|)$ time. The deletion of an edge from `leave` is completed in constant time. The

head set HS is returned from $\texttt{head}(e_t)$ to $\texttt{head}(e_k)$ in time proportional to the number of maximal sublists by the information of the places in $\texttt{head}(e_k)$. Therefore, procedure restore-data-structure( ) takes $O(|Can(e_t; T^p, k)| + |\{e \in Can(e_k; T^p, k) | \partial^+ e < \partial^+ g\}|)$ time.

5. An optimal implementation and its analysis. Finally, we describe our efficient implementation and analyze its time and space complexities. Our implementation is shown below.

ALGORITHM all-spanning-trees(G);
input: a graph G with a vertex set $\{v_1, \ldots, v_V\}$ and an edge set $\{e_1, \ldots, e_E\}$;
begin
by using a depth-first search, (simultaneously) execute
- find a depth-first spanning tree T^0 of G,
- sort vertices and edges to satisfy assumptions 2, 3, 4, and 5,
- for each $e_j \in T^0$, $\texttt{candi}(e_j) := \{e | e \notin T^0, \partial^+ e \leq \partial^+ e_j \text{ and } \partial^- e = \partial^- e_j\}$,
- for each $e_j \in T^0$, $\texttt{head}(e_j) := \{\partial^- e_j\}$,
- $\texttt{leave} := \{e_j \in T^0 | \texttt{candi}(e_j) \neq \emptyset\}$;

output("$e_1, e_2, \ldots, e_{V-1}, tree$,"); {output T^0}
find-children( ); {of T^0}
end.

PROCEDURE find-children( ); {T^p:current spanning tree}
begin
if $\texttt{leave} = \emptyset$ **then** return;
$Q := \emptyset$;
$e_k :=$ the last entry of leave;
delete e_k from leave;
while $\texttt{candi}(e_k) \neq \emptyset$ **do begin**
$g :=$ the last entry of $\texttt{candi}(e_k)$;
delete g from $\texttt{candi}(e_k)$, and add g to the beginning of Q;
output("$-e_k, +g, tree$,"); {output $T^c := T^p \backslash e_k \cup g$}
update-data-structure(e_k,g);
find-children( ); {find children of T^c}
restore-data-structure(e_k,g);
output("$-g, +e_k$,"); {reconstruct $T^p := T^c \cup e_k \backslash g$}
end;
move all entries of Q to $\texttt{candi}(e_k)$;
update-data-structure(e_k,e_k);
find-children( ); {find children of T^p containing e_k}
restore-data-structure(e_k,e_k);
add e_k to the end of leave;
end.

Now we discuss the time complexity of our implementation. The next lemma is useful for analyzing the time complexity.

LEMMA 5.1 (see [9]). *Suppose that T is a spanning tree and that k is a positive integer with $e_k < \text{Min}(T^0 \setminus T)$. Under Assumptions* 1, 2, 3, *and* 4, *for any edge $g_j \in \{e_j\} \cup Can(e_j; T, k)$ $(j \leq k)$, $T' = T \setminus \{e_1, \ldots, e_k\} \cup \{g_1, \ldots, g_k\}$ is a spanning tree.*

Proof. Let $T^j = T \setminus \{e_j, \ldots, e_k\} \cup \{g_j, \ldots, g_k\}$ for $j = 1, \ldots, k$. Obviously, T^k is a spanning tree. We suppose that T^j is a spanning tree. If $j \geq 2$, from Lemma 4.3, $Can(e_{j-1}; T, j-1) \subseteq Can(e_{j-1}; T^j, j-1)$. Thus $T^{j-1} = T^j \backslash e_{j-1} \cup g_{j-1}$ is a spanning tree. □

In algorithm all-spanning-tree( ), the time required other than for calling find-children( ) is $O(V+E)$. At state (T^p, k), $O(\#$ of children of T^p not containing $e_k)$ time is taken to execute procedure find-children( ) other than for the maintenance of data structures. Now we consider the time complexities of the maintenance of data structures. From the discussion in section 4, it takes $O(|Can(e_t; T^p, k)| + |Can(e_k; T^p, k) \cap \{e|\partial^+ e < \partial^+ g\}| + |\{e_j | j < t \text{ and } Can(e_j; T^p, k) \neq \emptyset\}|)$ time to maintain data structures when the state changes between (T^p, k) and $(T^p \backslash e_k \cup g, k-1)$, where e_t is an edge with $\partial^- e_t = \partial^+ g$. We consider the following two cases.

Case A. Maintenance for finding children of T^c (i.e., $g \in Can(e_k; T^p, k)$).

Case B. Maintenance for finding children of T^p containing e_k (i.e., $g = e_k$).

Note that Case A occurs exactly one time for each spanning tree T^c other than T^0 and that Case B occurs at most one time for each spanning tree T^p and for each edge $e_k \in \{e | e_1 \leq e < \mathrm{Min}(T^0 \setminus T^p)\}$. In Case A, $|Can(e_t; T^p, k)| + |Can(e_k; T^p, k) \cap \{e|\partial^+ e < \partial^+ g\}|$ is bounded by the number of children of T^c not containing e_t. Moreover, for each edge e_j with $j < t$ and $Can(e_j; T^p, k) \neq \emptyset$, there is a child of T^c not containing e_j. Therefore, the time complexity in Case A is $O(\#$ of children of $T^c)$. In Case B, $|Can(e_k; T^p, k) \cap \{e|\partial^+ e < \partial^+ e_k\}|$ is bounded by the number of children of T^p not containing e_k. From Lemma 5.1, T^p has at least $|\{e \in Can(e_k; T^p, k) | \partial^+ e < \partial^+ e_k\}| \times |Can(e_t; T^p, k)|$ grandchildren which contain neither e_k nor e_t. Similarly, $|\{e_j | j < t \text{ and } Can(e_j; T^p, k) \neq \emptyset\}|$ is bounded by the number of grandchildren of T^p not containing e_k. Thus the time complexity in Case B is

$$O(\#\text{ of children of } T^p \text{ not containing } e_k) + O(\#\text{ of grandchildren of } T^p \text{ not containing } e_k).$$

We recall that procedure find-children( ) checks in constant time whether T^p has children. From the above discussion, the total required time of find-children( ) at state (T^p, k) is

$$O(\#\text{ of children and grandchildren of } T^p \text{ not containing } e_k).$$

Thus the total time complexity of our implementation is $O(N+V+E)$.

Finally, we consider the space complexity. At any state, the edge sets `candi`(e_j) $(j = 1, \ldots, V-1)$ have no intersection with each other, and neither do the head sets `head`(e_j) $(j = 1, \ldots, V-1)$. Thus we need $O(V+E)$ space for `candi` and $O(V)$ space for `head`. Obviously, the cardinality of `leave` is at most $V-1$. As we described in section 4, the size of the stack recording positions maximal sublists of HS is $O(V)$ in all. The total size of local variables Q in find-children( ) is $O(E)$ because each edge is stored in one of the global variables `candi`$(*)$ or local variables Q. Hence the space complexity of our implementation is $O(V+E)$.

THEOREM 5.2. *The time and space complexities of our implementation are $O(N+V+E)$ and $O(V+E)$, respectively.*

In this paper, we proposed an efficient algorithm for enumerating all spanning trees. This is optimal in sense of time and space complexities.

Acknowledgment. We are greatly indebted to Professor Yoshiko T. Ikebe of Science University of Tokyo for her kind and valuable comments on this manuscript.

REFERENCES

[1] D. Avis and K. Fukuda, *A basis enumeration algorithm for linear systems with geometric applications*, Appl. Math. Lett., 4 (1991), pp. 39–42.
[2] D. Avis and K. Fukuda, *A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra*, Discrete Comput. Geom., 8 (1992), pp. 295–313.
[3] D. Avis and K. Fukuda, *Reverse search for enumeration*, Discrete Appl. Math., 65 (1996), pp. 21–46.
[4] H. N. Gabow and E. W. Myers, *Finding all spanning trees of directed and undirected graphs*, SIAM J. Comput., 7 (1978), pp. 280–287.
[5] S. Kapoor and H. Ramesh, *Algorithms for enumerating all spanning trees of undirected and weighted graphs*, SIAM J. Comput., 24 (1995), pp. 247–265.
[6] T. Matsui, *An algorithm for finding all the spanning trees in undirected graphs*, Research Report, Department of Mathematical Engineering and Information Physics, University of Tokyo, Tokyo, 1993.
[7] G. J. Minty, *A simple algorithm for listing all the trees of a graph*, IEEE Trans. Circuit Theory, CT-12 (1965), p. 120.
[8] R. C. Read and R. E. Tarjan, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks, 5 (1975), pp. 237–252.
[9] A. Shioura and A. Tamura, *Efficiently scanning all spanning trees of an undirected graph*, J. Oper. Res. Soc. Japan, 38 (1995), pp. 331–344.
[10] R. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 693–707, June 1997

006

SIZE–DEPTH TRADEOFFS FOR THRESHOLD CIRCUITS*

RUSSELL IMPAGLIAZZO†, RAMAMOHAN PATURI†, AND MICHAEL E. SAKS‡

Abstract. The following size–depth tradeoff for threshold circuits is obtained: any threshold circuit of depth d that computes the parity function on n variables must have at least $n^{1+c\theta^{-d}}$ edges, where $c > 0$ and $\theta \leq 3$ are constants independent of n and d. Previously known constructions show that up to the choice of c and θ this bound is best possible. In particular, the lower bound implies an affirmative answer to the conjecture of Paturi and Saks that a bounded-depth threshold circuit that computes parity requires a superlinear number of edges. This is the first superlinear lower bound for an explicit function that holds for any fixed depth and the first that applies to threshold circuits with unrestricted weights.

The tradeoff is obtained as a consequence of a general restriction theorem for threshold circuits with a small number of edges: For any threshold circuit with n inputs, depth d, and at most kn edges, there exists a partial assignment to the inputs that fixes the output of the circuit to a constant while leaving $\lfloor n/(c_1k)^{c_2\theta^d} \rfloor$ variables unfixed, where $c_1, c_2 > 0$ and $\theta \leq 3$ are constants independent of n, k, and d.

A tradeoff between the number of gates and depth is also proved: any threshold circuit of depth d that computes the parity of n variables has at least $(n/2)^{1/2(d-1)}$ gates. This tradeoff, which is essentially the best possible, was proved previously (with a better constant in the exponent) for the case of threshold circuits with polynomially bounded weights in [K. Siu, V. Roychowdury, and T. Kailath, *IEEE Trans. Inform. Theory*, 40 (1994), pp. 455–466]; the result in the present paper holds for unrestricted weights.

Key words. threshold circuits, circuit complexity, lower bounds

AMS subject classification. 68Q15

PII. S0097539792282965

1. Introduction.

A fundamental problem in complexity theory is to prove lower bounds on the size and the depth of general Boolean circuits for specific problems of interest such as arithmetic operations, graph reachability, linear programming, and satisfiability [11, 8, 5]. Unfortunately, current research has not begun to provide lower bounds for such computationally significant problems in general models. For example, the best known lower bound on the size of Boolean circuits over the standard basis {AND, OR, NOT} for any problem in **NP** is a $4n - 4$ bound on the parity function [20]; over the basis of all two-input functions, the best known lower bound is $3n - 3$ [4].

Since proving bounds for general circuits seems very difficult, it is interesting to look at restricted families of circuits, for example, small-depth circuits over various bases. Some of these classes of circuits are interesting on their own. For example, the size and the depth required for unbounded-fan-in circuits over the basis {AND, OR, NOT} to compute a function f are the same as the number of processors (up to a polynomial factor) and the parallel time (up to a constant factor) required to compute f on a CREW PRAM model.

* Received by the editors August 1, 1992; accepted for publication (in revised form) July 11, 1995. http://www.siam.org/journals/sicomp/26-3/28296.html

† Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093 (rimpagliazzo@ucsd.edu, paturi@cs.ucsd.edu).

‡ Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093 and Department of Mathematics, Rutgers University, New Brunswick, NJ 08903 (saks@math.rutgers.edu). The research of this author was supported in part by NSF grant CCR-8911388, AFOSR grants 89-0512 and 90-0008, and DIMACS, which is funded by NSF grant STC-91-19999.

Another basis of interest is the family of linear threshold gates. Circuits over this basis, threshold circuits, have attracted interest as a model for neural networks [14, 12] and because of the potential that hardware implementations of threshold circuits might become a reality [15]. Bounded-depth threshold circuits are also appealing theoretically since they provide a surprisingly strong bounded-depth computational model. Indeed, it has been shown that basic operations like addition, multiplication, division, and sorting can be performed by bounded-depth polynomial-size threshold circuits [7, 17, 22, 5, 2, 6, 24, 27, 13]. On the other hand, unbounded-fan-in bounded-depth polynomial-size circuits over the standard basis (even when supplemented with mod p gates for prime p) cannot compute majority [5, 21, 25]. Therefore, separating the class of functions computable by bounded-depth polynomial-size threshold circuits, TC^0, from those computable by polynomial-time Turing machines would be an extremely interesting result in complexity theory.

In this paper, we give the first superlinear separation between bounded-depth threshold circuits and **P**. More precisely, our main result (Theorem 1 and its refinement, Theorem 3) says that for any threshold circuit with n inputs, depth d, and kn edges, there exists a partial assignment to the inputs that fixes the output of the circuit to a constant while leaving at least $\lfloor n/(c_1k)^{c_2\theta^d} \rfloor$ variables unfixed, where $c_1, c_2 > 0$ and $\theta \leq 3$ are constants. In particular, this implies (Corollary 2 and its refinement, Corollary 4) that any depth-d circuit that computes the parity function on n variables must have at least $n^{1+c\theta^{-d}}$ edges for the same θ and some constant $c > 0$, proving the conjecture of Paturi and Saks [18]. (The value of θ obtained in this paper is $1 + \sqrt{2} = 2.414\ldots$, as compared to the value $(1 + \sqrt{5})/2 = 1.618\ldots$ in the upper bound given in [18].) In particular, any linear-size threshold circuit for parity requires depth $\Omega(\log \log n)$, matching the upper bound given in [18].

The only lower bounds known previously for the number of edges needed to compute the parity function were for depth-2 and depth-3 circuits with polynomial-size weights. In [18], it is proved that $\Omega(n^2/\log^2 n)$ edges are required for depth-2 threshold circuits and $\Omega(n^{1.2}/\log^{5/3} n)$ edges are required for depth-3 circuits. These results are obtained by showing that small-size depth-2 and depth-3 threshold circuits can be approximated by low-degree rational functions. The results in this paper are more general in that they hold for threshold circuits with arbitrary weights and all depths. However, for the special cases mentioned above, our techniques yield weaker bounds.

Our proof uses a random restriction method as in [1, 9, 26, 10, 16]. However, unlike previous proofs, our proof uses a distribution on the partial assignments that depends on the structure of the circuit. The main restriction lemma (Lemma 3.1) shows that for any family of linear threshold gates on a common set of n variables with a total of δn edges, there is a partial assignment that leaves $n/(4\delta^2 + 2)$ variables free and makes every gate in the family dependent on at most one variable. Given a threshold circuit, this lemma can be applied to the set of gates at the first level in order to reduce the depth of the circuit by 1. A straightforward induction argument then yields the main result with $\theta = 3$. A more careful induction argument improves this to $\theta = 1 + \sqrt{2}$.

In fact, the restriction lemma applies to a more general class of functions than threshold functions, called *generalized monotone* functions. A Boolean function $f(\vec{x})$ is *generalized monotone* if $f(\vec{x}) = g(\vec{x} \oplus \vec{b})$ for some monotone Boolean function g and Boolean vector $\vec{b}$, where $\oplus$ represents the componentwise addition mod 2. (These functions are sometimes referred to in the literature as "unate.")

We also prove analogous results for the number of gates in a small-depth threshold

circuit. We prove a lemma (Lemma 3.2) that is analogous to Lemma 3.1 and says that for any family of N generalized monotone function gates on a common set of n variables there is a partial assignment that leaves $n/(N^2+1)$ variables free and fixes all of the functions. This result, together with a simple induction argument, proves Theorem 2—that for any threshold circuit with n inputs, depth d, and N gates, there exists a partial assignment to the inputs that fixes the output of the circuit to a constant while leaving at least $\lfloor n/2N^{2(d-1)} \rfloor$ variables unfixed. This theorem easily implies a $(n/2)^{1/2(d-1)}$ bound on the number of threshold gates required to compute parity by a depth-d threshold circuit (Corollary 3). A similar bound ($\Omega(dn^{1/d}/\log^2 n)$) was obtained previously in [23] in the special case of circuits with polynomial-size weights. Beigel [3] obtains similar bounds for a more general circuit model that allows any subexponential number of AND, OR, and NOT gates.

Section 2 contains definitions and some preliminary observations. In section 3, we state the main restriction theorem with $\theta = 3$ and show how it follows from Lemma 3.1. We also formalize the statement of Theorem 2 and show how it follows from Lemma 3.2. These two lemmas are proved in the succeeding two sections. In section 6, a more careful argument is used to improve the value of θ in the main restriction theorem to $1+\sqrt{2}$. In the last section, we present some related combinatorial results and discuss some possible strengthenings.

2. Preliminaries. A *threshold gate* with fan-in n is an $(n+1)$-tuple $g = (\vec{w}; b)$, where $\vec{w} \in \mathbf{R}^n$ and $b \in \mathbf{R}$. w_i is called the *weight* of variable i and b is called the *threshold value* for the gate. The Boolean function $f_g : \{0,1\}^n \to \{0,1\}$ computed by g is defined on input $(x_1, \ldots, x_n) = \vec{x} \in \{0,1\}^n$ by $f_g(\vec{x}) = \operatorname{sgn}(g(\vec{x}))$, where the weighted sum $g(\vec{x})$ is given by $g(\vec{x}) = \langle \vec{w}, \vec{x} \rangle - b = \sum_{i+1}^{n} w_i x_i - b$ and $\operatorname{sgn} : \mathbf{R} \to \{0,1\}$ is defined as

$$\operatorname{sgn}(\alpha) = \begin{cases} 1 & \text{if } \alpha > 0, \\ 0 & \text{otherwise.} \end{cases}$$

A Boolean function f which is representable as f_g for some threshold gate is called a *threshold function*.

A threshold circuit T on n inputs is a directed acyclic graph with a designated node (output) and exactly n *source* nodes, one for each input. Each nonsource node is labeled by a threshold gate with its fan-in equal to the in-degree of the node. The function $f_v(x_1, \ldots, x_n)$ computed by the node v is obtained by functional composition in the obvious way. The function $f_T : \{0,1\}^n \to \{0,1\}$ computed by T is the function computed by the designated output node.

The *gate complexity* of T is defined as the number of nonsource nodes of T. The *edge complexity* of T is defined as the number of edges in T.

The *level* of a node in a circuit T is defined inductively. The level of each source node is 0. The level of any other node i is one more than the maximum level of its immediate predecessors. The *depth* of T is the level of the output node. The circuit T is *layered* if the inputs to each gate are from gates of level one less.

It will be convenient to fix a variable set X of cardinality n and define an *assignment* of X to be a function $\alpha : X \to \{0,1\}$. Letting $\mathcal{A}(X)$ denote the set of assignments, we then view an n-variable Boolean function f as a function from $\mathcal{A}(X)$ to $\{0,1\}$. We say that f *depends on* variable $x \in X$ if there are two assignments α and β that differ only in their values at x such that $f(\alpha) \neq f(\beta)$. The set of variables that f depends on is denoted by $S(f)$, and $s(f) = |S(f)|$.

As usual, we write $\alpha \leq \beta$ if $\alpha(x) \leq \beta(x)$ for all $x \in X$, and we denote the complement of α by $\overline{\alpha}$. A monotone Boolean function h is one that satisfies $h(\alpha) \leq h(\beta)$ whenever $\alpha \leq \beta$. The sum $\alpha \oplus \beta$ of two assignments is defined by $(\alpha \oplus \beta)(x) = (\alpha(x) + \alpha(y)) \bmod 2$. A Boolean function f is a *generalized monotone* function if there exists an assignment β and a monotone function h such that $f(\alpha) = h(\alpha \oplus \beta)$. The assignment β is called an *orientation* of f. It is easy to see that any threshold function is a generalized monotone function g and β is an orientation of g if and only for each in variable x in $S(f)$, $\beta(x) = \operatorname{sgn}(w_x)$, where w_x is the weight of the variable x.

A *partial assignment* α of X is a function from a subset Y of X to $\{0,1\}$. The domain Y of α is denoted $\Delta(\alpha)$, and elements $x \in Y$ are said to be *assigned* or *fixed* by α. The variables in the set $\Phi(\alpha) = X - \Delta(\alpha)$ are said to be *unassigned* or *free*. We denote by $\mathcal{P}(X)$ the set of all partial assignments of X. This set contains $\mathcal{A}(X)$; if we wish to emphasize that an assignment α is in $\mathcal{A}(X)$, we say that it is a *total assignment*. If Y is a subset of variables and α is a total assignment, then α_Y denotes the partial assignment with domain Y and $\alpha_Y(x) = \alpha(x)$ for $x \in Y$.

If α and β are partial assignments such that $\Delta(\beta) \subseteq \Delta(\alpha)$ and $\beta(x) = \alpha(x)$ for $x \in \Delta(\beta)$, then we say that α *extends* or is an *extension* of β. If α and β are partial assignments that fix disjoint sets of variables, then the partial assignment $\alpha\beta$ is the unique minimal extension of both α and β. For a Boolean function f and a partial assignment α, the *restriction of f induced by α*, written as $f(\alpha)$, is the Boolean function with variable set $\Phi(\alpha)$ obtained by assigning the variables in $\Delta(\alpha)$ according to α.

An *ordering* of a set Y is a bijection $\Gamma : [|Y|] \to Y$, where $[k]$ denotes the set $\{1, 2, \ldots, k\}$. Given Γ, we refer to $\Gamma(i)$ as the ith element of Y. Also, $\Gamma(\leq i)$ denotes the set $\{\Gamma(j) : j \leq i \text{ and } j \in [|Y|]\}$ and $\Gamma(\geq i)$ denotes the set $\{\Gamma(j) : j \geq i \text{ and } j \in [|Y|]\}$.

The following simple lemma states the main property of the generalized monotone functions that is used in this paper.

LEMMA 2.1. *Let f be a nonconstant generalized monotone function on X with an orientation β, and let Γ be an ordering of X. Then there exists a $j \in \{0, 1, \ldots, n\}$ such that $f(\beta_{\Gamma(\leq j)})$ is identically 0 and $f(\bar{\beta}_{\Gamma(\geq j)})$ is identically 1.*

Proof. Label the elements of X as $x_1, \ldots, x_n$ according to the ordering Γ. Any assignment α is identified with the vector $(\alpha(x_1), \ldots, \alpha(x_n))$. Consider first the case that f is monotone, i.e., $\beta = 0^n$. Since f is not a constant function, we have $f(0^n) = 0$ and $f(1^n) = 1$. Let j be the least index such that $f(0^j 1^{n-j}) = 0$. This implies that $f(\beta_{\Gamma(\leq j)})$ is identically 0. $j \geq 1$ since f is not a constant function. Then $f(0^{j-1}1^{n-j+1}) = 1$, which implies that $f(\bar{\beta}_{\Gamma(\geq j)})$ is identically 1 by monotonicity since every total assignment that extends $\bar{\beta}_{\Gamma(\geq j)}$ is greater than or equal to $0^{j-1}1^{n-j+1}$.

In the case that f is not monotone, the desired result follows immediately by applying the previous argument to the monotone function $h(\alpha) = f(\alpha \oplus \beta)$. □

One useful consequence of this lemma is the following.

COROLLARY 1. *Any generalized monotone function f on n variables has a partial assignment α that leaves at least $\lfloor n/2 \rfloor$ variables free, such that $f(\alpha)$ is constant.*

3. Results. Our main result concerns the computational power of depth-d threshold circuits with a small number of edges.

THEOREM 1. *Let C be an n-input threshold circuit with depth d and nk edges, where $k \geq 1$. Let f denote the function computed by C. Then there exists a partial assignment α that leaves at least $\lfloor n/(2(3k)^{3^{d-1}-1}) \rfloor$ variables free such that $f(\alpha)$ is a*

constant function.

If f is the parity function, then $f(\alpha)$ is constant only if α is a total assignment. Thus it follows from Theorem 1 that if C is a depth-d circuit with nk edges that computes the parity function on n variables, then $n < 2(3k)^{3^{d-1}-1}$. This yields the following result.

COROLLARY 2. *Any threshold circuit of depth d that computes parity of n variables has at least $n^{1+1/(3^{d-1}-1)}/(3\sqrt{2})$ edges.*

The key to proving Theorem 1 is the following.

LEMMA 3.1 (main lemma). *Let F be a collection of generalized monotone functions on n variables and let $\delta = (1/n)\sum_{f\in F} s(f)$ (so the total support of the functions is $n\delta$). Then there exists a partial assignment α that leaves at least $n/(4\delta^2+2)$ variables free such that for every $f \in F$, $f(\alpha)$ depends on at most one variable.*

Proof of Theorem 1 *from main lemma.* We proceed by induction on the depth d of the circuit. If $d = 1$, the circuit consists of a single threshold gate and the conclusion follows from Corollary 1. For $d > 1$, let F be the family of functions corresponding to the gates at depth 1. By hypothesis, the sum of the fan-ins of these gates is at most nk. Lemma 3.1 implies that there is a partial assignment that leaves at least $n' = n/(4k^2+2) \geq n/(6k^2)$ variables free such that the induced restriction of each function in F depends on at most one variable. We may then collapse the first level of the circuit, i.e., if g is a gate at depth 2, then each input to g is either an input to the circuit or the output of a gate at level 1, which after the restriction is equal to a variable or its complement. Thus each gate g at depth 2 can now be reexpressed as a threshold gate that depends only on the original inputs. (Note that g may have several edges entering which depend on the same variable, but these can be combined into one edge by adjusting the weights of g.) Hence we obtain a depth-$(d-1)$ circuit C' on at least n' variables with at most $n'k'$ edges, where $k' = 6k^3$ and $nk = n'k'$. By the induction hypothesis, there exists a partial assignment to the variables of C' such that the number of free variables is at least

$$\left\lfloor \frac{n'}{2(3k')^{3^{d-2}-1}} \right\rfloor \geq \left\lfloor \frac{n/(6k^2)}{2(3(6k^3))^{3^{d-2}-1}} \right\rfloor \geq \left\lfloor \frac{n}{2(3k)^{3^{d-1}-1}} \right\rfloor,$$

as required to prove the theorem. □

Remark. Since the main lemma applies to generalized monotone functions, it might appear that Theorem 1 could be generalized to apply to circuits whose gates compute arbitrary generalized monotone functions. However, the proof fails to generalize because when the circuit is collapsed in the induction step, a level-2 gate may have more than one input corresponding to the same variable. In that case, it is not true that the gate computes a generalized monotone function of the original variables; indeed, it is easy to see that *every* n-variable Boolean function can be represented as a single generalized monotone function on $2n$ variables by identifying variables in pairs.

To bound the number of gates in a small-depth circuit instead of the number of edges, we use the following (simpler) relative of Lemma 3.1.

LEMMA 3.2. *Let F be a collection of generalized monotone functions on n variables. Then there exists a partial assignment α that leaves at least $\lfloor n/(|F|^2+1)\rfloor$ variables free such that for each $f \in F$, $f(\alpha)$ is a constant function.*

This leads to the following result for threshold circuits with a small number of gates. In this case, the result holds for generalized monotone functions.

THEOREM 2. *Let C be a circuit consisting of generalized monotone function gates of depth d on n inputs with at most N gates. Then there exists a partial assignment α leaving $\lfloor n/2N^{2(d-1)} \rfloor$ variables free such that $f_C(\alpha)$ is constant.*

Proof of Theorem 2 *from Lemma* 3.2. We proceed by induction on the depth d of the circuit, as in the proof of Theorem 1. If $d = 1$, the circuit consists of a single threshold gate and the conclusion follows from Corollary 1. For $d > 1$, consider the family F of threshold functions corresponding to the depth-1 gates. Note that $|F| \leq N - 1$. We apply Lemma 3.2 to F to obtain a partial assignment that leaves at least $n' = \lfloor n/(|F|^2 + 1) \rfloor \geq \lfloor n/N^2 \rfloor$ variables free such that the induced restriction of each function in F is constant. After the restriction, the only nonconstant inputs to the second-level gates are the inputs to the circuit. Thus the resulting circuit C' has depth at most $d-1$, at most N gates, and at least n' variables. By the induction hypothesis, there exists a partial assignment of the variables of C' which leaves at least $\lfloor n'/2(N)^{2(d-2)} \rfloor \geq \lfloor n/2(N)^{2(d-1)} \rfloor$ variables free (where the inequality follows from the fact that for positive integers n, A, and B, $\lfloor \lfloor n/A \rfloor / B \rfloor = \lfloor n/AB \rfloor$). □

Again using the fact that the only partial assignments that make the parity function constant are the total assignments, we deduce that the number N of gates of a depth-d parity circuit satisfies $2N^{2(d-1)} \geq n$, and thus we have the following.

COROLLARY 3. *Any circuit of depth d consisting of generalized monotone function gates that computes the parity of n inputs has at least $(n/2)^{1/2(d-1)}$ gates.*

Slightly stronger bounds (removing the half from the exponent of the lower bound) than those obtained in Theorem 2 and Corollary 3 were previously proved in [17, 23] for the case of threshold circuits with polynomially bounded weights.

It remains to prove Lemmas 3.1 and 3.2, and these proofs constitute the main part of the paper. The proofs of these lemmas are similar; both use a probabilistic method to demonstrate the existence of the required partial assignment.

The proof of Lemma 3.2 is somewhat simpler, so we present the proof in the next section. The proof of Lemma 3.1 will be presented in section 5.

4. Proof of Lemma 3.2. In the probabilistic arguments in this section and the next, we adopt the following notational convention. Random variables are denoted by placing a ˜ over the identifier. When we refer to a specific value that a random variable may assume, we denote that value by an identifier without a ˜.

We have a family F of Boolean generalized monotone functions on n variables and seek a partial assignment that makes all of the functions constant. It will be convenient to fix an indexing $f^1, f^2, \ldots, f^m$ of the functions in F. Let β^i be an orientation for f^i.

Fix an ordered partition $Y_1, Y_2, \ldots, Y_q$ of the variable set X into $q = m^2+1$ blocks of nearly equal size (each having $\lfloor n/q \rfloor$ or $\lfloor n/q \rfloor + 1$ variables). The desired partial assignment will be obtained by fixing the variables in all but one of the blocks. We describe a randomized procedure P which produces such a partial assignment $\tilde{\alpha}$ and show that with positive probability $f^i(\tilde{\alpha})$ is constant for all $i \in [m]$.

The procedure P is as follows. Let $\mathcal{U}$ be a symbol (meaning "unallocated"). Choose uniformly at random a 1–1 function $\tilde{M}$ from $[m] \times [m] \cup \{\mathcal{U}\}$ to $[q]$. Intuitively, we think of $\tilde{M}$ as "allocating" sets $Y_{\tilde{M}(i,1)}, \ldots, Y_{\tilde{M}(i,m)}$ to function f^i, while leaving set $Y_{\tilde{M}(\mathcal{U})}$ unallocated. In addition, choose a vector $(\tilde{t}_1, \tilde{t}_2, \ldots, \tilde{t}_m)$ uniformly from the set $\{0, 1, 2, \ldots, m\}^m$. For each $1 \leq i, j \leq m$, if $j \leq \tilde{t}_i$, then fix the variables in $Y_{\tilde{M}(i,j)}$ according to β^i, and if $j > \tilde{t}_i$, then fix the variables in $Y_{\tilde{M}(i,j)}$ according to $\bar{\beta}^i$. Thus all of the variables except those in the unique block $Y_{\tilde{M}(\mathcal{U})}$ are fixed. Call the resulting partial assignment $\tilde{\alpha}$.

The key property of this distribution is given by the following lemma.

LEMMA 4.1. *For each $h \in [m]$, the probability that $f^h(\tilde{\alpha})$ is not constant is at most $1/(m+1)$.*

It follows from this lemma that the probability that there exists $i \in [m]$ with $f^i(\tilde{\alpha})$ not constant is at most $m/(m+1)$. Thus there exists a particular α such that $f^i(\alpha)$ is constant for all $i \in [m]$, and this α satisfies the conclusion of Lemma 3.2.

Proof of Lemma 4.1. Fix $h \in [m]$. We define a modification P^h of the procedure P. It will be easy to see that this modified construction produces the same distribution; we then use the modified construction to verify the conclusion of the lemma.

The modified construction is as follows. Choose $\tilde{t'}_i \in \{0, \ldots, m\}$ uniformly at random for $i \neq h$, and pick $\tilde{t'}_h \in \{1, \ldots, m+1\}$ uniformly at random. Pick $\tilde{M'}$, a random 1–1 function from $[m] \times [m] \cup \{(h, m+1)\}$ to $[q]$. For $i \neq h$, assign variables in $Y_{\tilde{M'}(i,j)}$ as before, according to β^i if $j \leq \tilde{t'}_i$ and according to $\bar{\beta^i}$ otherwise. For $j < \tilde{t'}_h$, assign the variables in $Y_{\tilde{M'}(h,j)}$ according to β^h, and for $j > \tilde{t'}_h$, assign them according to $\bar{\beta^h}$. We leave the variables in $Y_{\tilde{M'}(h,\tilde{t'}_h)}$ unassigned.

As in the original procedure, each gate is allocated m random sets of variables, with one random set of variables being unallocated. For each gate, the number of these sets fixed according to the orientation of the gate is randomly chosen between 0 and m, and the rest are set according to the negation of the orientation. Thus the two distributions are identical. More formally, we could define $\tilde{M}(i,j) = \tilde{M'}(i,j)$ for $i \neq h$, $\tilde{M}(h,j) = \tilde{M'}(h,j)$ for $j < \tilde{t_h}'$, $\tilde{M}(h,j) = \tilde{M'}(h,j+1)$ for $\tilde{t'}_h < j \leq m$, and $\tilde{M}(\mathcal{U}) = \tilde{M'}(h, \tilde{t_h}')$ and define $\tilde{t}_i = \tilde{t'}_i$ for $i \neq h$, $\tilde{t}_h = \tilde{t'}_h - 1$. Then the distributions on $\tilde{M}$ and $\tilde{t}$ are identical to those in the original process, and all values M and $t_1, \ldots, t_m$ of these random variables, if chosen by the original process, would determine the same value of α as M' and the t'_i's do in the modified process.

Thus it will suffice to upper bound the probability that $f^h(\tilde{\alpha})$ is not constant when $\tilde{\alpha}$ is constructed according to P^h. For this, fix any value M' for $\tilde{M'}$, and fix values t'_i for $\tilde{t'}_i$, $i \neq h$. This determines the setting of $\tilde{\alpha}$ for all the variables in $Y_{M'(i,j)}$ for $i \neq h$. We will show that, given the above information, the probability that f^h is nonconstant when restricted by $\tilde{\alpha}$ is at most $1/m + 1$. Let g be f^h restricted to the variables in the blocks $Y_{h,j}$, $1 \leq j \leq m+1$, with the other variables set according to $\tilde{\alpha}$. (As we noted before, the value of $\tilde{\alpha}$ at all other variables has been fixed by the information that we are conditioning on.) g is a generalized monotone function with the same orientation β^h as f^h.

For each block $Y_{M'(h,j)}$, fix an arbitrary order on variables of the block; extend these orders to an ordering Γ on all the variables for g by ordering the blocks according to j. Then we can apply Lemma 2.1 to obtain an index l such that the functions $g(\beta^h_{\Gamma(\leq l)})$ and $g(\bar{\beta}^h_{\Gamma(\geq l)})$ are both constant. Let $0 \leq r \leq m+1$ be such that $\Gamma(l) \in Y_{M'(h,r)}$, i.e., the lth variable is in the rth block allocated to f^h. We claim that $g(\tilde{\alpha})$—and hence $f^h(\tilde{\alpha})$—is constant unless $\tilde{t'}_h = r$, an event which happens with probability $1/(m+1)$ (since $\tilde{t'}_h \in [m+1]$ is chosen independently from $\tilde{M'}$ and the $\tilde{t'}_i$'s for $i \neq h$.). If $\tilde{t'}_h > r$, all variables in blocks labeled r or less are fixed by $\tilde{\alpha}$ to β^h, so $\tilde{\alpha}$ extends $\beta^h_{\Gamma(\leq l)}$, so $g(\tilde{\alpha})$ is constant. Similarly, if $\tilde{t'}_h < r$, $\tilde{\alpha}$ extends $\bar{\beta^h}_{\Gamma(\geq l)}$ and $g(\tilde{\alpha})$ is constant. Thus with probability $1 - 1/(m+1)$, $f^h(\tilde{\alpha}) = g(\tilde{\alpha})$ is constant, as required to complete the proof of Lemma 4.1 and hence of Lemma 3.2. □

5. Proof of the main lemma. Again, index the functions in F as $f^1, \ldots, f^m$ and let β^i denote an orientation for f^i. For each variable x, let D_x be the subfamily of F that consists of those functions that depend on variable x and let $\delta_x = |D_x|$.

Thus the quantity $\delta = (1/n)\sum_{f\in F} s(f)$ in the lemma is also the average of the δ_x's. We seek a partial assignment α that leaves at least $n/(4\delta^2+2)$ variables free and such that for every $f \in F$, $f(\alpha)$ depends on at most one variable.

We will describe a randomized algorithm $A(L)$, where L is a positive real parameter, for constructing a partial assignment $\tilde{\alpha}$ and show that, for an appropriate choice of L, $\tilde{\alpha}$ has the desired properties with positive probability. The random procedure in the previous proof can be viewed as associating a fraction $m/(m^2+1)$ of the variables to each function and then fixing the variables associated with a function in a way that is determined by the orientation of the function. We will do something similar here; however, here we will require that the set of variables assigned to f^i is a subset of $S(f^i)$, the set of variables on which f^i depends.

PROCEDURE $A(L)$.

1. *Partition the variables.* (Intuitively, this step assigns each function a set of variables in proportion to its support size, leaving a few variables unassigned.) Construct a random partition of the variable set X into $m+1$ parts $\tilde{R}, \tilde{C}_1, \tilde{C}_2, \ldots, \tilde{C}_m$. For each variable x, the block of the partition containing x is determined independently according to the following rule. With probability $1/(1+L\delta_x)$, $x \in \tilde{R}$. Otherwise, x is assigned to block $\tilde{C}_{\tilde{i}(x)}$, where $\tilde{i}(x)$ is the index of a uniformly chosen element of D_x. In other words, for each $f^i \in D_x$, the probability that $x \in \tilde{C}_i$ is $L/(1+L\delta_x)$. Let $\tilde{r} = |\tilde{R}|$ and for each $i \in [m]$, let $\tilde{c}_i = |\tilde{C}_i|$.

2. *For each $i \in [m]$, fix all of the variables in $\tilde{C}_i$.* (Intuitively, this step fixes the variables assigned to each f^i so that any particular function f^i becomes constant with a good probability.) For each $i \in [m]$, choose $\tilde{b}_i$ uniformly at random from $\{0, 1, \ldots, \tilde{c}_i\}$. Choose a subset $\tilde{B}_i$ of $\tilde{C}_i$ uniformly from all $\tilde{b}_i$-element subsets of $\tilde{C}_i$. Let $\tilde{\gamma}^i$ denote the partial assignment which fixes the variables of $\tilde{B}_i$ according to β^i and fixes the variables of $\tilde{C}_i - \tilde{B}_i$ according to $\bar{\beta}^i$. Let $\tilde{\gamma}$ be the union of the partial assignments $\tilde{\gamma}^i, i \in [m]$.

3. *Fix some of the variables in $\tilde{R}$.* (Intuitively, this step cleans up the few remaining functions that are still nonconstant so that they depend on at most one variable.) For each $i \in [m]$, let $\tilde{T}_i$ denote the set of variables on which $f^i(\tilde{\gamma})$ depends, and if $\tilde{T}_i \neq \emptyset$, let $\tilde{T'}_i$ be an arbitrary subset containing all but one element of $\tilde{T}_i$; otherwise, $\tilde{T'}_i = \emptyset$. Let $\tilde{\alpha}$ be the restriction obtained from $\tilde{\gamma}$ by setting all the elements of each $\tilde{T'}_i$ to 1.

The third step above ensures that the partial assignment $\tilde{\alpha}$ has the required property that $f^i(\tilde{\alpha})$ depends on at most one variable for each i. Thus it remains to show that with positive probability the number $\tilde{\phi}$ of variables left free is sufficiently large.

The set of free variables of $\tilde{\alpha}$ consists of those variables in $\tilde{R}$ that are not fixed during step 3. Thus $\tilde{\phi} \geq \tilde{r} - \sum_{i=1}^{m} \max\{0, |\tilde{T}_i| - 1\}$. Our goal is to obtain a lower bound on the expectation of $\tilde{\phi}$. Note that $\mathbf{E}[\tilde{r}] = \sum_{x\in X} 1/(L\delta_x + 1)$.

The harder part is to upper bound the expectation of $a(|\tilde{T}_i|)$, where $a(m) = \max\{0, m-1\}$. The key lemma of this section is the following.

LEMMA 5.1. *For each $h \in [m]$,*

$$\mathbf{E}[a(|\tilde{T}_h|)] \leq \frac{1}{L} \sum_{x\in S(f^h)} \frac{1}{L\delta_x + 1}.$$

Assuming this lemma for the moment, we have

$$\begin{aligned}
\mathbf{E}[\tilde{\phi}] &\geq \mathbf{E}[\tilde{r}] - \sum_{i=1}^{m} \mathbf{E}[a(|\tilde{T}_i|)] \\
&\geq \sum_{x \in X} \frac{1}{(Ld_x + 1)} - \sum_{i=1}^{m} \frac{1}{L} \sum_{x \in S(f^i)} \frac{1}{L\delta_x + 1} \\
&\geq \sum_{x \in X} \frac{1}{(Ld_x + 1)} - \frac{1}{L} \sum_{x \in X} \sum_{f_i \in D_x} \frac{1}{L\delta_x + 1} \\
&\geq \sum_{x \in X} \frac{1}{(Ld_x + 1)} - \sum_{x \in X} \frac{\delta_x / L}{L\delta_x + 1} \\
&= \sum_{x \in X} \frac{1 - \delta_x / L}{L\delta_x + 1} \\
&\geq n \left(\frac{1 - \delta / L}{L\delta + 1} \right),
\end{aligned}$$

where the last inequality follows from the convexity of the function $\lambda(z) = (1 - z/L)/(Lz + 1)$ for positive z and the fact (Jensen's inequality) that the arithmetic mean of a convex function on a set is at least the function evaluated at the mean value of the set. Choosing the parameter $L = 2\delta$ to (approximately) maximize this quantity, we will have that the expectation of $\tilde{\phi}$ is at least $n/(4\delta^2 + 2)$. Thus among the partial assignments that could be produced by the procedure $A(2\delta)$, there must exist a partial assignment that leaves at least $n/(4\delta^2 + 2)$ variables free, as required to prove the main lemma.

It remains to prove Lemma 5.1. Fix $h \in [m]$. Let $\tilde{\chi}_h$ denote the random variable which is 1 if the hth function is not fixed after step 2, i.e., if $f^h(\tilde{\gamma})$ is not constant, and is 0 if $f^h(\tilde{\gamma})$ is constant. Clearly, $\tilde{T}_h$ is empty if $\tilde{\chi}_h = 0$, and otherwise $\tilde{T}_h$ is a subset of the set $\tilde{U}_h = \tilde{R} \cap S(f^h)$. Letting $\tilde{u}_h$ denote the cardinality of $|\tilde{U}_h|$, we have $a(|\tilde{T}_h|) \leq \tilde{\chi}_h a(\tilde{u}_h)$. Lemma 5.1 is an immediate consequence of the following Lemma.

LEMMA 5.2. *Let $h \in [m]$ and let q be an arbitrary nonnegative-valued function defined on the natural numbers. Then*

$$\mathbf{E}[\tilde{\chi}_h q(\tilde{u}_h)] \leq \frac{1}{L} \mathbf{E}[q(\tilde{u}_h + 1)].$$

Applying this lemma with $q = a$, the right-hand side of the inequality is just $\mathbf{E}[\tilde{u}_h]/L$, which by linearity of expectation is

$$\frac{1}{L} \sum_{x \in S(f^h)} \mathbf{P}[x \in \tilde{R}] = \frac{1}{L} \sum_{x \in S(f^h)} \frac{1}{L\delta_x + 1},$$

as required to prove Lemma 5.1.

Proof of Lemma 5.2. Let $\tilde{K} = \tilde{C}_h \cup \tilde{U}_h$, i.e., the set of variables on which f^h depends that are assigned to either $\tilde{C}_h$ or $\tilde{R}$. Let $\tilde{k} = |\tilde{K}|$.

Fix a particular instantiation C_i and B_i for all $i \neq h$ and let Ξ denote the event that $\tilde{C}_i = C_i$ and $\tilde{B}_i = B_i$ for all $i \neq h$. Note that Ξ determines the value K of $\tilde{K}$ and also determines $\tilde{\gamma}$ on all variables in $S(f^h) - \tilde{K}$. Thus let g be the function of the variables in K determined by restricting f^h according to $\tilde{\gamma}_i$ for each $i \neq h$.

We will show that for any such event Ξ,

$$\mathbf{E}[\tilde{\chi}_h q(\tilde{u}_h) \mid \Xi] \leq \frac{1}{L}\mathbf{E}[q(\tilde{u}_h + 1) \mid \Xi].$$

The lemma then follows by deconditioning the expectation.

Given Ξ, the variables in K are partitioned into the two sets $\tilde{C}_h$ and $\tilde{U}_h$ as follows: for $x \in K$, the conditional probability given Ξ that x is in $\tilde{R}$ (and hence in $\tilde{U}_h$) is $p = 1/(L+1)$, and otherwise (with probability $L/(L+1)$) x is in $\tilde{C}_h$. Furthermore, these events are independently determined for each $x \in K$. Thus the conditional distribution given Ξ of $\tilde{u}_h$ is a binomial distribution $B(k,p)$, i.e., $\mathbf{P}[\tilde{u}_h = i \mid \Xi] = \binom{k}{i}p^i(1-p)^{k-i}$. We have

$$\begin{aligned}\mathbf{E}[\tilde{\chi}_h q(\tilde{u}_h) \mid \Xi] &\leq \sum_{i=0}^{k} q(i)\mathbf{P}[\tilde{u}_h = i \mid \Xi]\mathbf{P}[g(\tilde{\gamma}) \text{ is not constant} \mid \Xi \wedge (\tilde{u}_h = i)] \\ &= \sum_{i=1}^{k} q(i)\binom{k}{i}p^i(1-p)^{k-i}\mathbf{P}[g(\tilde{\gamma}) \text{ is not constant} \mid \Xi \wedge (\tilde{u}_h = i)].\end{aligned}$$

We next determine an upper bound for $\mathbf{P}[g(\tilde{\gamma})\text{is not constant} \mid \Xi \wedge (\tilde{u}_h = i)]$.

The conditional distribution of $\tilde{C}_h, \tilde{B}_h$ given $\Xi \wedge (\tilde{u}_h = i)$ can be described as follows. $\tilde{C}_h$ is a uniformly chosen $(k-i)$-element subset of K, $\tilde{b}_h$ is chosen uniformly at random from $\{0, 1, \ldots, k-i\}$, and $\tilde{B}_h$ is a uniformly chosen $\tilde{b}_h$-element subset of $\tilde{C}_h$.

An alternative way to generate this same distribution on $\tilde{B}_h, \tilde{C}_h$ is as follows: Choose an order $\tilde{\Gamma}$ of the elements of K uniformly at random. Choose $\tilde{b}_h$ uniformly from $\{0, 1, \ldots, k-i\}$. Let $\tilde{B}_h$ be the first $\tilde{b}_h$ elements of K and let $\tilde{C}_h$ consist of $\tilde{B}_h$ together with the last $k - i - \tilde{b}_h$ elements of K. It is clear that this distribution is equivalent to the one described in the previous paragraph.

We want to determine the conditional probability that g is not constant given Ξ and $\tilde{u}_h = i$. Lemma 2.1 applied to the function g and the ordering $\tilde{\Gamma}$ of K implies that there is an index $\tilde{j} = \tilde{j}(\tilde{\Gamma})$ in $\{0, 1, \ldots, k\}$ such that $f(\beta^h_{\tilde{\Gamma}(\leq \tilde{j})})$ is identically 0 and $f(\bar{\beta}^h_{\tilde{\Gamma}(\geq \tilde{j})})$ is identically 1. Now observe that if $\tilde{b}_h$ is chosen to be greater than or equal to $\tilde{j}$, then the partial assignment $\tilde{\gamma}^h$ is an extension of $\beta^h_{\tilde{\Gamma}(\leq \tilde{j})}$ and $f(\tilde{\gamma}^h)$ is thus identically 0. Similarly, if $\tilde{b}_h$ is chosen to be less than $\tilde{j} - i$, then the partial assignment $\tilde{\gamma}^h$ is an extension of $\bar{\beta}^h_{\tilde{\Gamma}(\geq \tilde{j})}$ and $f(\tilde{\gamma}^h)$ is thus identically 1. Thus the only way that $\tilde{\Xi}_h$ can be nonzero is if $\tilde{b}_h$ satisfies $\tilde{j} - i \leq \tilde{b}_h \leq \tilde{j} - 1$, and since $\tilde{b}_h$ is chosen uniformly in the range $\{0, 1, \ldots, k-i\}$, this happens with probability at most $i/(k-i+1)$. We conclude that the conditional probability given $\Xi \wedge (\tilde{u}_h = i)$ that g is not constant is at most $i/(k-i+1)$. Using this probability, we can rewrite the expression for the conditional expectation of $\tilde{\chi}_h q(\tilde{u}_h)$ as

$$\begin{aligned}\mathbf{E}[\tilde{\chi}_h q(\tilde{u}_h) \mid \Xi] &\leq \sum_{i=1}^{k} q(i)\binom{k}{i}p^i(1-p)^{k-i}\frac{i}{k-i+1} \\ &= \frac{p}{1-p}\sum_{i=1}^{k} q(i)\binom{k}{i-1}p^{i-1}(1-p)^{k-(i-1)} \\ &= \frac{p}{1-p}\sum_{i'=0}^{k-1} q(i'+1)\binom{k}{i'}p^{i'}(1-p)^{k-i'}\end{aligned}$$

$$
\begin{aligned}
&= \frac{p}{1-p} \sum_{i'=0}^{k-1} q(i'+1)\mathbf{P}[\tilde{u}_h = i' \mid \Xi] \\
&\leq \frac{p}{1-p}\mathbf{E}[q(\tilde{u}_h + 1) \mid \Xi] \\
&= \frac{1}{L}\mathbf{E}[q(\tilde{u}_h + 1) \mid \Xi],
\end{aligned}
$$

as required to complete the proof of Lemma 5.2, which in turn completes the proofs of Lemma 5.1 and the main lemma. □

6. An improved lower bound. In this section, we present refined versions of Theorem 1 and Corollary 2 for which the parameter θ is reduced from 3 to $1 + \sqrt{2}$. In the following, our results are stated for layered threshold circuits. This is sufficient for our purposes since an arbitrary threshold circuit can be converted to a layered one that computes the same function by increasing the number of edges by a factor of at most d.

To state the improvement of Theorem 1, define ν_i for $i \geq 1$ to be the solution to the recurrence equation $\nu_{i+2} = 2\nu_{i+1} + \nu_i$ with the initial conditions $\nu_1 = 1$ and $\nu_2 = 3$. Note that the explicit expression for ν_i is of the form $A(1+\sqrt{2})^i + B(1-\sqrt{2})^i$, where $A \neq 0$ and B are easily determined constants, and so $\nu_i \in \Theta((1+\sqrt{2})^i)$.

THEOREM 3. *Let C be a layered depth-d threshold circuit with n inputs d and nk edges, where $k \geq 1$. Let f denote the function computed by C. Then there exists a partial assignment α that leaves at least $\lfloor n/4(11k)^{\nu_d - 1} \rfloor$ variables free such that $f(\alpha)$ is a constant function.*

As before, this theorem immediately implies a size–depth tradeoff for the parity function.

COROLLARY 4. *Any threshold circuit of depth $d \geq 2$ that computes parity of n variables has at least $(n/11)^{1+\frac{1}{\nu_d - 1}}$ edges.*

To motivate the proof of Theorem 3, we first summarize the main inductive argument of the previous proof. In each inductive step, the depth of the circuit is decreased by 1 by fixing some variables in order to eliminate the first level. The fraction of variables left unfixed after each step is inversely proportional to the square of the parameter δ, the ratio of the number of edges at the first level to the number of unfixed variables before the step. In analyzing the resulting recurrence, we upper bounded the number of edges at the first level by the total number of edges in the circuit.

The idea for improving this analysis is to substantially improve this upper bound on the number of edges at the first level, thereby increasing the fraction of variables that are known to survive each reduction step. It might seem that since the circuit is arbitrary, we cannot do better than to bound the number of edges at the first level by the total number of edges in the circuit. This is indeed true the first time the reduction is applied. However, it turns out that for all subsequent reduction steps, there is a better bound available. This is because the partial assignment produced by $A(L)$ in the proof of Lemma 3.1 has a very useful side effect: for each first-level gate whose output is fixed to a constant by the partial assignment, the edges leaving that gate can be eliminated from the circuit. We will show that with high probability the number of edges in the second level of the circuit (which becomes the first level) is decreased by a large amount. This allows us to keep a larger fraction of variables

unassigned when we recursively perform the reduction on the first level of the resulting circuit.

To make this idea precise, we need a modified version of Lemma 3.1.

LEMMA 6.1. *Let F be a collection of generalized monotone functions on n variables, and suppose that each function $f \in F$ has a nonnegative weight $w(f)$. Let $\delta = (1/n)\sum_{f\in F} s(f) \geq 1$ and $W = \sum_{f\in F} w(f)$. Then assuming that $n/(9\delta)^2 \geq 4$, there exists a partial assignment α that leaves at least $n/(9\delta)^2$ variables free such that*

1. *for every $f \in F$, $f(\alpha)$ depends on at most one variable*;
2. *$\sum_{f:f(\alpha)\ is\ not\ constant} w(f) \leq W/8\delta$.*

Note that when we apply this lemma in the inductive argument, the weights of the functions will correspond to the out-degree of the corresponding gates. The point is that the total number of edges remaining on the new first level can then be bounded above by $1/8\delta$ times the number of edges in the circuit.

Proof of Lemma 6.1. The proof is a modification of that of Lemma 3.1, and we retain the notation of that lemma. We use procedure $A(L)$ to generate the random restriction $\tilde{\alpha}$. We show the following:

1. With probability at least $1/2$, the sum of the weights of the nonconstant gates is at most $2W/L$.

2. Assuming that $n \geq 16(1 + L\delta)$, then with probability exceeding $1/2$, the number $\tilde{\phi}$ of free variables is at least $n(1 - 8\delta/L)/2(L\delta + 1)$.

If we choose $L = 16\delta$, then it follows immediately that with positive probability $\tilde{\alpha}$ satisfies the conclusion of the lemma, and thus such a partial assignment exists.

Thus it suffices to prove the two claims. For the first claim, using the notation of Lemma 5.2, the sum of the weights of the nonconstant gates can be bounded above by

$$\sum_{h=1}^{m} \tilde{\chi}_h w(f^h).$$

The expectation of a generic term of the sum can be bounded above by $w(f^h)/L$ using Lemma 5.2 with q being the constant function $w(f^h)$. Thus the expectation of the sum is at most W/L. By Markov's inequality, with probability greater than $1/2$, the sum does not exceed $2W/L$.

We now verify the second claim. As in the proof of the main lemma, we write $\tilde{\phi} \geq \tilde{r} - \tilde{s}$, where $\tilde{s}$ denotes $\sum_{i=1}^{m} \max\{0, |\tilde{T}_i| - 1\}$.

Recall that $\tilde{r}$ is the cardinality of $\tilde{R}$. For each $x \in X$, let $\tilde{r}_x$ denote the random variable that is 1 if $x \in \tilde{R}$ and is 0 otherwise; then $\tilde{r} = \sum_{x\in X} \tilde{r}_x$. Thus as observed previously, $\mathbf{E}[\tilde{r}] = \sum_{x\in X} 1/(1 + L\delta_x)$, which is at least $n/(1 + L\delta)$ by the convexity of the function $\lambda(y) = 1/(1 + Ly)$ for nonnegative y. Furthermore, since the variables $\tilde{r}_x$ are mutually independent, we may use Chernoff-type bounds (see, e.g., Theorem 2 of [19]) to bound the probability that $\tilde{r}$ is less than half its mean: $\mathbf{P}[\tilde{r} < \mathbf{E}[\tilde{r}]/2] < e^{-\mathbf{E}[\tilde{r}]/8} = e^{-n/8(1+L\delta)}$. In particular, for $n \geq 16(1 + L\delta)$, this is less than $1/4$.

On the other hand, by Markov's inequality, $\mathbf{P}[\tilde{s} > 4\mathbf{E}[\tilde{s}]] \leq 1/4$. Combining this with the previous inequality, we get $\mathbf{P}[(\tilde{r} \geq \mathbf{E}[\tilde{r}]/2) \wedge (\tilde{s} \leq 4\mathbf{E}[\tilde{s}])] > 1/2$, which implies that $\mathbf{P}[\tilde{r} - \tilde{s} \geq \mathbf{E}[\tilde{r}]/2 - 4\mathbf{E}[\tilde{s}]] > 1/2$. As noted above, $\mathbf{E}[\tilde{r}] \geq n/(1 + L\delta)$, and from Lemma 5.1, $\mathbf{E}[\tilde{s}] \leq \sum_{x\in X} (\delta_x/L)/(L\delta_x + 1)$ which is at most $(\delta/L)/(L\delta + 1)$ (by the concavity of the function $\lambda(y) = (y/L)/(Ly + 1)$ for positive y). Substituting these bounds, we get $\mathbf{P}[\tilde{\phi} \geq n(1 - 8\delta/L)/2(L\delta + 1)] > 1/2$. □

COROLLARY 5. *Let C be a depth-d layered threshold circuit with n inputs and nk edges, where $k \geq 1$. Let f denote the function computed by C. For $i \geq 0$, let*

$\rho_i = (11k)^{\nu_{i+1}-1}$. Then for each $i \in \{0, 1, \ldots, d-1\}$ such that $n \geq 4\rho_i$, there exists a partial assignment α^i that leaves at least n/ρ_i variables free such that $f(\alpha^i)$ can be computed by a layered circuit C_i of depth $d - i$.

Proof. For $i = 0$, we take α^i to be the trivial restriction and $C_0 = C$. For $i \in [d-1]$, we will use Lemma 6.1 repeatedly to define partial assignments α^i and circuits C_i that have depth $d - i$. For C_i, we let n_i denote the number of inputs, m_i denote the number of edges entering the level-1 gates, and F_i denote the family of functions computed by the level-1 gates.

For $0 \leq i \leq d-2$, we construct C_{i+1} from C_i as follows: Apply Lemma 6.1 to the set F_i with $w(f)$ equal to the fan-out (in C_i) of the gate into level 2. Note that the quantity δ in this application of the lemma is equal to m_i/n_i. Thus the hypothesis of the lemma holds as long as $n_i \geq 4(9(m_i/n_i))^2$. Assuming this holds, then after the application of the lemma, we can eliminate the level-1 gates to produce C_{i+1}.

It remains to verify that for $0 \leq i \leq d-2$, $n_i \geq 4(9(m_i/n_i))^2$ (so that Lemma 6.1 can be applied in constructing C_{i+1}) and $n_{i+1} \geq n/\rho_{i+1}$ (which is the conclusion of the corollary). From the conclusion of Lemma 6.1, $n_{i+1} \geq n_i/(9(m_i/n_i))^2$. Thus if we define $p_0 = n$ and $p_i = n_{i-1}^3/(9m_{i-1})^2$ for $i \geq 1$, then the condition for applying the lemma to construct C_{i+1} for $0 \leq i \leq d-2$ is $p_{i+1} \geq 4$ and the conclusion of the lemma gives $n_{i+1} \geq p_{i+1}$. Furthermore, Lemma 6.1 implies that the number of edges into the level-1 gates of C_{i+1} is at most $1/8\delta$ times the number of edges into the level-2 gates of C_i, and hence $m_{i+1} \leq nkn_i/8m_i$. Squaring this and multiplying both sides by $81p_{i+2}n_i$, we obtain $n_i n_{i+1}^3 \leq (81nk/8)^2 p_{i+2}p_{i+1}$. Substituting the bounds $n_{i+1} \geq p_{i+1}$ and $n_i \geq p_i$ yields the following recurrence entirely in terms of p_i: for $i \geq 0$,

$$p_{i+2} \geq \left(\frac{8}{81nk}\right)^2 p_{i+1}^2 p_i \geq \left(\frac{1}{11nk}\right)^2 p_{i+1}^2 p_i$$

with the initial conditions $p_0 = n$ and $p_1 \geq n^3/(9nk)^2 \geq n/(11k)^2$.

If we set $l_i = \log p_i$, we get a linear recurrence which is easily shown to imply $p_i \geq n/\rho_i$. (Alternatively, this inequality can be verified directly by induction on i.)

Thus for each $i \in \{0, 1, \ldots, d-2\}$, if $n \geq 4\rho_{i+1}$, then $p_{i+1} \geq 4$ and $n_{i+1} \geq p_{i+1} \geq n/\rho_{i+1}$, as required. □

We can now finish the proof of Theorem 3. If $n < 4\rho_{d-1}$ then we can choose any total assignment for α and the conclusion holds trivially. Otherwise, we may apply Corollary 5 with $i = d-1$ to find a partial assignment α^{d-1} with at least n/ρ_{d-1} unfixed variables such that the resulting restricted function can be computed by a single threshold gate. Applying Corollary 1, we need to fix at most half the remaining variables to make the function constant. □

7. Final remarks and open problems. This paper gives the first nontrivial lower bounds, for threshold circuits with arbitrary weights and any fixed depth, on the number of edges and gates needed to compute an explicit function. The results show that there are functions $\epsilon(d)$ and $\gamma(d)$ such that any depth-d threshold circuit that computes parity on n variables must have at least $n^{1+\epsilon(d)}$ edges and $n^{\gamma(d)}$ gates. In our case, the functions $\epsilon(d)$ and $\gamma(d)$ tend to 0 as d tends to ∞. An apparently difficult challenge would be to prove an n^ϵ lower bound, with $\epsilon > 1$ a constant independent of depth, on the number of gates needed to compute some explicit function.

For each fixed depth, there is a gap between the bounds provided by our results and the best constructions for parity circuits. For instance, for depth-2 circuits, the

result in this paper gives an $\Omega(n^{3/2})$ bound on the number of edges and an $n^{1/2}$ bound on the number of gates, while the best construction requires $O(n^2)$ edges and $O(n)$ gates. One way to reduce this gap is to improve Lemma 3.1 by increasing the number of variables left free in the restrictions.

PROBLEM 1. *What is the smallest exponent r such that the conclusions of Lemmas* 3.1 *and* 6.1 *hold with $n/(4\delta^2+2)$ replaced by $\Omega(n/\delta^r)$?*

The best possible r is at least 1, as shown by the family $F = \{T_i : 0 \le i \le n\}$ of n-variable functions, where T_i is the function which is 1 on inputs with at least i 1's. If the conclusion holds for $r = 1$, then this would lead to an $\Omega(n^2)$-edge lower bound for depth-2 circuits that compute parity and, more generally, to an improvement in the value of θ in the main theorem to $\theta = (1+\sqrt{5})/2$. This would exactly match the value of θ in the known upper bounds. Note that for purposes of applications to circuits, it would suffice to consider the above problem for families of threshold functions rather than for generalized monotone functions.

It is interesting also to look for a similar improvement to Lemma 3.2.

PROBLEM 2. *What is the smallest exponent r such that the conclusion of Lemma* 3.2 *holds with $n/(4\delta^2+2)$ replaced by $\Omega(n/|F|^r)$?*

Again, the best lower bound on r we have is 1. Any value of $r < 2$ would give a corresponding improvement in Theorem 2: the number of variables left free would be $\Omega(n/N^{r(d-1)})$.

For the special case of monotone functions, it is easy to show that Lemma 3.2 has such a strengthening.

PROPOSITION 1. *Let F be a collection of monotone functions on n variables. Then there exists a partial assignment α that leaves at least $\lfloor n/(|F|+1)\rfloor$ variables free such that for each $f \in F$, $f(\alpha)$ is a constant function.*

Proof. Fix an ordering Γ for the variables X and for each $f \in F$, let $j(f)$ be the index promised by Lemma 2.1. Order the functions as $f_1, f_2, \ldots, f_m$ so that $j_1 \le j_2 \le \cdots \le j_m$, where $j_i = j(f_i)$, and let $j_0 = 0$ and $j_{m+1} = n+1$. Let i be an index such that $j_{i+1} - j_i$ is maximum (and hence at least $(n+1)/(m+1)$) and let α be the assignment which sets all variables in $\Gamma(\le j_i)$ to 0 and $\Gamma(\ge j_{i+1})$ to 1. Then $f_h(\alpha)$ is identically 0 for all $h \le i$ and $f_h(\alpha)$ is identically 1 for all $h \ge i+1$. The number of free variables of α is $j_{i+1} - j_i - 1 \ge \lfloor n/(m+1)\rfloor$. □

REFERENCES

[1] M. AJTAI, *Σ_1^1-formulae on finite structures*, Ann. Pure Appl. Logic, 24 (1983), pp. 1–48.

[2] P. BEAME, S. A. COOK, AND H. J. HOOVER, *Log depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994–1003.

[3] R. BEIGEL, *When do extra majority gates help?*, in Proc. 24th ACM Symposium on Theory of Computing, ACM, New York, 1992, pp. 450–454.

[4] N. BLUM, *A Boolean function requiring 3n network size*, Theoret. Comput. Sci., 28 (1984), pp. 337–345.

[5] R. BOPPANA AND M. SIPSER, *The complexity of finite functions*, in Handbook of Theoretical Computer Science, Vol. A, Elsevier Science Publishers, Amsterdam, New York, 1990, pp. 757–804.

[6] J. BRUCK, AND R. SMOLENSKY, *Polynomial threshold functions, AC^0 functions and spectral norms*, in Proc. 31st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 632–641.

[7] A. CHANDRA AND L. STOCKMEYER, *Constant depth reducibility*, SIAM J. Comput., 13 (1984), pp. 423–439.

[8] P. E. DUNNE, *The Complexity of Boolean Functions*, Academic Press, New York, 1988.

[9] M. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13–28.

[10] J. Håstad, *Almost optimal lower bounds for small depth circuits*, in Proc. 18th ACM Symposium on Theory of Computing, ACM, New York, 1986, pp. 6–20.
[11] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, Reading, MA, 1979.
[12] G. E. Hinton, *Connectionist learning procedures*, Technical Report CMU-CS-87-115, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1987.
[13] T. Hofmeister, W. Hohberg, and S. Köhling, *Some notes on threshold circuit and multiplication in depth* 4, Inform. Process. Lett., 39 (1991), pp. 219–225.
[14] M. Minsky and S. A. Papert, *Perceptrons*, expanded ed., MIT Press, Cambridge, MA, 1988.
[15] C. Mead, *Analog VLSI and Neural Systems*, Addison–Wesley, Reading, MA, 1989.
[16] W. Maass, G. Schnitger, and E. D. Sontag, *On the computational power of sigmoid versus Boolean threshold circuits*, in Proc. 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 767–776.
[17] I. Parberry and G. Schnitger, *Parallel computation with threshold functions*, J. Comput. System Sci., 36 (1988), pp. 278–302.
[18] R. Paturi and M. E. Saks, *Approximating threshold circuits by rational functions*, Inform. and Comput., 112 (1994), pp. 257–272.
[19] P. Raghavan, *Probabilistic construction of deterministic algorithms: Approximating packing integer programs*, in Proc. 27th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 10–18.
[20] N. P. Red'kin, *A roof of minimality of circuits consisting of functional elements*, Problemy Kibernet., 23 (1973), pp. 83–102 (in Russian); Systems Theory Res., 23 (1973), pp. 85–103 (in English).
[21] A. A. Razborov, *Lower bounds on the size of bounded depth networks over a complete basis with logical addition*, Mat. Zametki, 41 (1986), pp. 598–607 (in Russian); Math. Notes Acad. Sci. USSR, 41 (1986), pp. 333–338 (in English).
[22] J. Reif, *On threshold circuits and polynomial computations*, in Proc. 2nd Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 118–125.
[23] K. Siu, V. Roychowdury, and T. Kailath, *Rational approximation techniques for analysis of neural networks*, IEEE Trans. Inform. Theory, 40 (1994), pp. 455–466.
[24] K.-Y. Siu, J. Bruck, and T. Kailath, *Depth-efficient neural networks for division and related problems*, IEEE Trans. Inform. Theory, 39 (1993), pp. 946–956.
[25] R. Smolensky, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, in Proc. 19th ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 77–82.
[26] A. C.-C. Yao, *Circuits and local computation*, in Proc. 21st ACM Symposium on Theory of Computing, ACM, New York, 1989, pp. 186–196.
[27] A. C.-C. Yao, *On ACC and threshold circuits*, in Proc. 31st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 619–627.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 708–732, June 1997

007

BOUNDS FOR THE COMPUTATIONAL POWER AND LEARNING COMPLEXITY OF ANALOG NEURAL NETS*

WOLFGANG MAASS†

Abstract. It is shown that high-order feedforward neural nets of constant depth with piecewise-polynomial activation functions and arbitrary real weights can be simulated for Boolean inputs and outputs by neural nets of a somewhat larger size and depth with Heaviside gates and weights from $\{-1, 0, 1\}$. This provides the first known upper bound for the computational power of the former type of neural nets. It is also shown that in the case of first-order nets with piecewise-linear activation functions one can replace arbitrary real weights by rational numbers with polynomially many bits without changing the Boolean function that is computed by the neural net. In order to prove these results, we introduce two new methods for reducing nonlinear problems about weights in multilayer neural nets to linear problems for a transformed set of parameters. These transformed parameters can be interpreted as weights in a somewhat larger neural net.

As another application of our new proof technique we show that neural nets with piecewise-polynomial activation functions and a constant number of analog inputs are probably approximately correct (PAC) learnable (in Valiant's model for PAC learning [*Comm. Assoc. Comput. Mach.*, 27 (1984), pp. 1134–1142]).

Key words. neural networks, analog computing, threshold circuits, circuit complexity, learning complexity

AMS subject classifications. 68Q05, 68Q15, 68T05, 92B20, 94C05

PII. S0097539793256041

1. Introduction.

We examine in this paper the computational power and learning complexity of high-order analog feedforward neural nets $\mathcal{N}$, i.e., of circuits with analog computational elements in which certain parameters are treated as programmable parameters. We focus on neural nets $\mathcal{N}$ of bounded depth in which each gate g computes a function from $\mathbf{R}^m$ into $\mathbf{R}$ of the form $\langle y_1, \ldots, y_m \rangle \mapsto \gamma^g(Q^g(y_1, \ldots, y_m))$. We assume that, for each gate g, γ^g is some fixed piecewise-polynomial activation function (also called response function). This function is applied to some polynomial $Q^g(y_1, \ldots, y_m)$ of bounded degree with arbitrary real coefficients, where $y_1, \ldots, y_m$ are the real-valued inputs to gate g. One usually refers to the degree of the polynomial Q^g as the *order* of the gate g. It should be noted that (following the conventions in the neural net literature) the order of a gate g does not refer to the degree of its activation function γ^g. We will specify bounds for that degree separately.

The coefficients ("weights") of Q^g are the *programmable variables* of $\mathcal{N}$ whose values may arise from some learning process.

We are primarily interested in the case where the neural net $\mathcal{N}$ computes (respectively, learns) a Boolean-valued function. For that purpose we assume that the real-valued output of the output gate g_{out} of $\mathcal{N}$ is "rounded off." More precisely, we assume that there is an *outer threshold* T_{out} (which belongs to the programmable parameters of $\mathcal{N}$) such that the output of $\mathcal{N}$ is 1 whenever the real-valued output z of g_{out} satisfies $z \geq T_{\text{out}}$ and 0 if $z < T_{\text{out}}$. In some results of this paper we also assume that the input $\langle x_1, \ldots, x_n \rangle$ of $\mathcal{N}$ is Boolean valued. It should be noted that this does

*Received by the editors September 22, 1993; accepted for publication (in revised form) July 12, 1995.

http://www.siam.org/journals/sicomp/26-3/25604.html

†Institute for Theoretical Computer Science, Technische Universität Graz, Klosterwiesgasse 32/2, A-8010 Graz, Austria (maass@igi.tu-graz.ac.at).

not affect the capacity of $\mathcal{N}$ to carry out, on its intermediate levels (i.e., in its "hidden units"), computation over reals, whose real-valued results are then transmitted to the next layer of gates.

Circuits of this type have rarely been considered in computational complexity theory, and they give rise to the principal question whether these intermediate *analog* computational elements will allow the circuit to compute more complex Boolean functions than a circuit with a similar layout but *digital* computational elements. Note that circuits with analog computational elements have an extra source of potentially unlimited parallelism at their disposal, since they can execute operations on numbers of arbitrary bit-length in one step, and they can transmit numbers of arbitrary bit-length from one gate to the next.

One already knows quite a bit about the special case of such neural nets $\mathcal{N}$, where each gate g is a linear threshold gate. In this case each polynomial $Q^g(y_1, \ldots, y_m)$ is of degree ≤ 1 (i.e., a weighted sum), and each activation function γ^g in $\mathcal{N}$ is the Heaviside function (also called hard limiter) $\mathcal{H}$ defined by

$$\mathcal{H}(y) = \begin{cases} 1 & \text{if } y \geq 0, \\ 0 & \text{if } y < 0 \end{cases}$$

(e.g., see [R], [Ni], [Mu], [MP], [PS], [HMPST], [GHR], [SR], [SBKH], [BH], [A], [B], [L]). The analog versus digital issue does not arise in this case, since the output of each gate is a single bit. Still, it requires some work to bound the potential power of arbitrary weights (in the weighted sums) for the computation of Boolean functions on such circuit. Since there are only finitely many Boolean circuit inputs, it is obvious that only rational weights have to be considered. The key result for the analysis of these circuits was the discovery of Muroga [Mu] that it is sufficient to consider for a linear threshold gate with m Boolean inputs only weights $\alpha_1, \ldots, \alpha_m$ and a bias α_0 that are integers of size $2^{O(m \log m)}$. (This upper bound is optimal according to a recent result of Håstad [Has].) With the help of this a priori bound on the relevant bit-length of weights, it is easy to show that the same arrays $(F_n)_{n \in \mathbf{N}}$ of Boolean functions $F_n : \{0,1\}^n \to \{0,1\}$ are computable by arrays $(\mathcal{N}_n)_{n \in \mathbf{N}}$ of neural nets of depth $O(1)$ and size $O(n^{O(1)})$ with linear threshold gates, no matter whether one uses as weights arbitrary reals, rationals, integers, or elements of $\{-1, 0, 1\}$; see [Mu], [CSV], [HMPST], [GHR], [MT]. The resulting class of arrays $(F_n)_{n \in \mathbf{N}}$ of Boolean functions is called (nonuniform) TC^0 (see [HMPST], [J]).

In comparison, very little is known about upper bounds for the computational power and the learning complexity of feedforward neural nets whose gates g employ more general types of activation functions γ^g. This holds in spite of the fact that "real neurons and real physical devices have continuous input-output relations" [Ho]. In the analysis of information processing in natural neural systems, one usually views the firing rate of a neuron as its current output. Such firing rates are known to change between a few and several hundred spikes per second (see Chap. 20 in [MR]). Hence the activation function γ^g of a gate g that models such a neuron should have a graded response. It should also be noted that the customary learning algorithms for artificial neural nets (such as backward propagation [RM]) are based on gradient descent methods which require that all gates g employ *smooth* activation functions γ^g.

In addition, it has frequently been pointed out that it is both biologically plausible and computationally relevant to consider gates g that pass to γ^g instead of a weighted sum $\sum_{i=1}^m \alpha_i y_i + \alpha_0$ some polynomial $Q^g(y_1, \ldots, y_m)$ of bounded degree, where $y_1, \ldots, y_m$ are circuit inputs or outputs of the immediate predecessors

of g. Such gates are called sigma-pi units or high-order gates in the literature (see p. 73 and Chap. 10 in [RM]; see also [DR], [H], [PG], [MD]). From the point of view of approximation theory there has been particular interest in the case in which $Q^g(y_1, \ldots, y_m) = \sum_{i=1}^m \alpha_i (y_i - c_i)^2$ measures a "distance" of its input $\langle y_1, \ldots, y_m \rangle$ from some "center" $\langle c_1, \ldots, c_m \rangle$ (the latter may be determined through a learning process). Apparently Theorems 3.1 and 4.3 of this paper provide the first upper bounds for the computational power and learning complexity of high-order feedforward neural nets with non-Boolean activation functions.

The power of feedforward neural nets with other activation functions besides $\mathcal{H}$ has previously been investigated in [RM, Chap. 10], [S1], [S2], [H], [MSS], [DS], [SS]. It was shown in [MSS] for a very general class of activation functions γ^g that neural nets $(\mathcal{N}_n)_{n \in \mathbf{N}}$ of constant depth and size $O(n^{O(1)})$ with real weights of size $O(n^{O(1)})$ and output separation $\Omega(1/n^{O(1)})$ (between the unrounded circuit outputs for rejected and accepted inputs) can compute only Boolean functions in TC^0. It follows from a result of Sontag [S2] that the assumptions on the weight size and separation are essential for this upper bound: he constructed an arbitrarily smooth monotone function Θ (which can be made to satisfy the conditions on γ^g in the quoted result of [MSS]) and neural nets $\mathcal{N}_n$ of size 2 (!) with activation function Θ such that $\mathcal{N}_n$ can compute with sufficiently large weights *any* Boolean function $F_n : \{0,1\}^n \to \{0,1\}$ (hence $\mathcal{N}_n$ has VC dimension 2^n).

These results leave open the question about the computational power and learning complexity of feedforward neural nets with arbitrary weights that employ "natural" analog activation functions γ^g. For example there has previously been no upper bound for the set of Boolean functions computable by analog neural nets with the very simple piecewise-linear activation function π defined by

$$\pi(y) = \begin{cases} 0 & \text{if } y \leq 0, \\ y & \text{if } 0 \leq y \leq 1, \\ 1 & \text{if } y \geq 1. \end{cases}$$

([L] refers to a gate g with $\gamma^g = \pi$ as a threshold logic element.) On the other hand, there exist results which suggest that such upper bound would be nontrivial. It has already been shown in [MSS] that constant size neural nets of depth 2 with activation function π and small integer weights can compute *more* Boolean functions than constant size neural nets of depth 2 with linear threshold gates (and arbitrary weights). [DS] exhibits an even stronger increase in computational power for the case of quadratic activation functions.

Hence even *simple* non-Boolean activation functions provide more computational power to a neural net than the Heaviside function. However it has been an open problem by *how much* they can increase the computational power (in the presence of arbitrary weights). From the technical point of view, this difficulty in proving an upper bound for the computational power was caused by the lack of an upper bound on the amount of information that can be encoded in such a neural net by the assignment of weights. For the case of neural nets with Heaviside gates, this upper bound on the information capacity of weights is provided by the quoted result of Muroga [Mu]. However, this problem is substantially more difficult for neural nets with piecewise-linear activation functions. For this model it is no longer sufficient to analyze a single gate with Boolean inputs and outputs. Even if the inputs and outputs of the neural net are Boolean valued, the signals that are transmitted between the hidden units are real valued. Furthermore, one can give no a priori bound on the

precision required for such analog signals between hidden units, since one has no control over the maximal size of weights in the neural net. Obviously a large weight will magnify any imprecision. Note also that a computation on a multilayer neural net of the type considered here involves *products* of weights from subsequent levels. Hence, if some of the weights are arbitrarily large, one needs arbitrarily high precision for the other weights.

The main technical contribution of this paper is two new methods for reducing nonlinear problems about weights in multilayer neural nets to linear problems for a transformed set of parameters. These two methods are presented in sections 2 and 3 of this paper. We introduce in section 2 of this paper a method that allows us to prove an upper bound for the information capacity of weights for neural nets with piecewise-linear activation functions (hence in particular for π). It is shown that for the computation of Boolean functions on neural nets $\mathcal{N}_n$ of constant depth and polynomially in n many gates (where n is the number of input variables) it is sufficient to use as weights rational numbers with polynomially in n many bits. As a consequence, one can simulate any such analog neural net by a digital neural net of constant depth and polynomial size with the Heaviside activation function (i.e., linear threshold gates) and binary weights (i.e., weights from $\{0, 1\}$). This result also implies that the VC dimension of $\mathcal{N}_n$ can be bounded above by a polynomial in n.

In section 3 we introduce another proof technique that allows us to derive the same two consequences for neural nets with piecewise-*polynomial* activation functions and nonlinear gate inputs $Q^g(y_1, \ldots, y_m)$ of bounded degree. These results show that in spite of the previously quoted evidence for the superiority of non-Boolean activation functions in neural nets, there is some limit to their computational power as long as the activation functions are piecewise-polynomial. On the other hand the polynomial upper bound on the VC dimension of such neural nets may be interpreted as *good news*: it shows that neural nets of this type can in principle be trained with a sequence of examples that is not too long.

We conclude in section 4 with a positive result for learning on neural nets in Valiant's model [V] for probably approximately correct learning (PAC learning). We consider the problem of learning on neural nets with a fixed number of analog (i.e., real-valued) input variables. We exploit here the implicit linearization of the requirements for the desired weight assignment that is achieved in the new proof techniques from sections 2 and 3. In this way one can show that such neural nets are properly PAC learnable in the case of piecewise-linear activation functions and PAC learnable with a hypothesis class that is given by a somewhat larger neural net in the case of piecewise-polynomial activation functions. Another application of our parameter transformation method from section 2 to PAC learning has subsequently been given by Koiran [K94].

The results of this paper were first announced in [M92], and an extended abstract of these results appeared in [M93a]. Another result of [M93a], the construction of neural nets whose VC dimension is superlinear in the number of weights, has subsequently been improved to apply for depth 3 also. A full version of that proof appears in [M93b].

DEFINITION 1.1. *A* network architecture *(or neural net) $\mathcal{N}$ of order k is a labeled acyclic directed graph $\langle V, E\rangle$. Its nodes of fan-in* 0 *are labeled by the input variables $x_1, \ldots, x_n$. Each node g of fan-in $m > 0$ is called a computation node (or gate) and is labeled by some activation function $\gamma^g : \boldsymbol{R} \to \boldsymbol{R}$ and some polynomial $Q^g(y_1, \ldots, y_m)$ of degree $\leq k$. Furthermore, $\mathcal{N}$ has a unique node of fan-out 0 which is called the*

output node of $\mathcal{N}$ and which carries as an additional label a certain real number T_{out} (called the outer threshold of $\mathcal{N}$).

The coefficients of all polynomials $Q^g(y_1, \ldots, y_m)$ for gates g in $\mathcal{N}$ and the outer threshold T_{out} are called the programmable parameters *of $\mathcal{N}$. Assume that $\mathcal{N}$ has w programmable parameters and that some numbering of these has been fixed. Then each assignment $\underline{\alpha} \in \boldsymbol{R}^w$ of reals to the programmable parameters in $\mathcal{N}$ defines an analog circuit $\mathcal{N}^{\underline{\alpha}}$, which computes a function $\underline{x} \mapsto \mathcal{N}^{\underline{\alpha}}(\underline{x})$ from $\boldsymbol{R}^n$ into $\{0,1\}$ in the following way: assume that some input $\underline{x} \in \boldsymbol{R}^n$ has been assigned to the input nodes of $\mathcal{N}$. If a gate g in $\mathcal{N}$ has m immediate predecessors in $\langle V, E\rangle$ which output $y_1, \ldots, y_m \in \boldsymbol{R}$, then g outputs $\gamma^g(Q^g(y_1, \ldots, y_m))$. Finally, if g_{out} is the output gate of $\mathcal{N}$ and g_{out} gives the real-valued output z (according to the preceding inductive definition), we define*

$$\mathcal{N}^{\underline{\alpha}}(\underline{x}) := \begin{cases} 1 & \text{if } z \geq T_{\text{out}}, \\ 0 & \text{if } z < T_{\text{out}}, \end{cases}$$

where T_{out} is the outer threshold that has been assigned by $\underline{\alpha}$ to g_{out}.

Any parameters that occur in the definitions of the activation functions γ^g of $\mathcal{N}$ are referred to as architectural parameters *of $\mathcal{N}$.*

DEFINITION 1.2. *A function $\gamma : \boldsymbol{R} \to \boldsymbol{R}$ is called* piecewise-polynomial *if there are thresholds $t_1, \ldots, t_k \in \boldsymbol{R}$ and polynomials $P_0, \ldots, P_k$ such that $t_1 < \cdots < t_k$ and for each $i \in \{0, \ldots, k\}$: $t_i \leq x < t_{i+1} \Rightarrow \gamma(x) = P_i(x)$ (we set $t_0 := -\infty$ and $t_{k+1} := \infty$).*

If k is chosen minimal for γ, we refer to k as the number of polynomial pieces of γ; to $P_0, \ldots, P_k$ as the polynomial pieces of γ; and to $t_1, \ldots, t_k$ as the thresholds of γ. Furthermore we refer to $t_1, \ldots, t_k$ together with all coefficients in the polynomials $P_0, \ldots, P_k$ as the parameters *of γ. The maximal degree of $P_0, \ldots, P_k$ is called the degree of γ. If the degree of γ is ≤ 1 then we call γ* piecewise-linear*, and we refer to $P_0, \ldots, P_k$ as the linear pieces of γ.*

If γ occurs as activation function γ^g of some network architecture $\mathcal{N}$, then one refers to the parameters of γ as architectural parameters *of $\mathcal{N}$.*

Note that *we do not require that γ is continuous* (*or monotone*). It should also be pointed out that according to Definition 1.1 the order k of a neural net does *not* bound the degrees of the polynomial pieces of its activation functions. Finally, we would like to mention that in contrast to [MSS], we do not require here any minimal distance between the real-valued network outputs z and the outer threshold T_{out}.

DEFINITION 1.3. *Assume that $\mathcal{N}$ is an arbitrary network architecture with n inputs and w programmable parameters and that $S \subseteq \boldsymbol{R}^n$ is an arbitrary set. Then one defines the* VC dimension *of $\mathcal{N}$ over S in the following way:*

$$\begin{aligned} \textit{VC dimension}(\mathcal{N}, S) := \max\{|S'| \,|\, & S' \subseteq S \textit{ has the property that for every function} \\ & F : S' \to \{0,1\} \textit{ there exists a parameter assignment} \\ & \underline{\alpha} \in \boldsymbol{R}^w \textit{ such that } \forall\, \underline{x} \in S' (\mathcal{N}^{\underline{\alpha}}(\underline{x}) = F(\underline{x}))\}. \end{aligned}$$

Remark 1.4. VC dimension is an abbreviation for Vapnik–Chervonenkis dimension. It has been shown in [BEHW] (see also [BH], [A]) that the VC dimension of a neural net $\mathcal{N}$ essentially determines the number of examples that are needed to train $\mathcal{N}$ (in Valiant's model for PAC learning [V]). Sontag [S2] has shown that the VC dimension of a neural net can be drastically increased by using activation functions with non-Boolean output instead of the Heaviside function $\mathcal{H}$. The methods described in

this paper were used in [M93a] to give the first proof of a polynomial upper bound for the VC dimension of the here-considered neural nets. Subsequently [GJ] have shown that such bounds for the VC dimension can also be derived more directly via Milnor's theorem. However their method does not yield upper bounds for the *computational* power of these neural nets.

2. A bound for the information—capacity of weights in neural nets with piecewise-linear activation functions. We consider for arbitrary $a \in \mathbf{N}$ the following set of rationals with up to a bits before and after the comma:

$$\mathbf{Q}_a := \left\{ r \in \mathbf{Q} \;\middle|\; r = s \cdot \sum_{i=-a}^{a-1} b_i \cdot 2^i \quad \text{for } b_i \in \{0,1\},\ i = -a, \ldots, a-1, \text{ and } s \in \{-1, 1\} \right\}.$$

Note that for any $r \in \mathbf{Q}_a : |r| \leq 2^a \leq 2^{2a} \cdot \min\{|r'| \mid r' \in \mathbf{Q}_a \text{ and } r' \neq 0\}$.

THEOREM 2.1. *Consider an arbitrary network architecture $\mathcal{N}$ of order 1 over a graph $\langle V, E \rangle$ with n input nodes in which every computation node has fan-out ≤ 1. Assume that each activation function γ^g in $\mathcal{N}$ is piecewise-linear with parameters from $\boldsymbol{Q}_a$. Let $w := |V| + |E| + 1$ be the number of programmable parameters in $\mathcal{N}$.*

Then for every $\underline{\alpha} \in \boldsymbol{R}^w$ there exists a vector $\underline{\alpha}' = \langle \frac{s_1}{t}, \ldots, \frac{s_w}{t} \rangle \in \boldsymbol{Q}^w$ with integers $s_1, \ldots, s_w, t$ of absolute value $\leq (2w+1)!\ 2^{2a(2w+1)}$ such that $\forall \underline{x} \in \boldsymbol{Q}_a^n (\mathcal{N}^{\underline{\alpha}}(\underline{x}) = \mathcal{N}^{\underline{\alpha}'}(\underline{x}))$. In particular $\mathcal{N}^{\underline{\alpha}'}$ computes the same Boolean function as $\mathcal{N}^{\underline{\alpha}}$.

Remark 2.2. The condition of Theorem 2.1 that all computation nodes in $\mathcal{N}$ have fan-out ≤ 1 is automatically satisfied for $d \leq 2$. For larger d one can simulate any network architecture $\mathcal{N}$ of depth d with s nodes by a network architecture $\mathcal{N}'$ with $\leq \frac{s}{s-1} \cdot s^{d-1} \leq \frac{3}{2} s^{d-1}$ nodes and depth d that satisfies this condition (replace each computation node with fan-out k by k identical nodes with fan-out 1, starting from the output layer). Hence this condition is not too restrictive for network architectures of a constant depth d.

It should also be pointed out that there is in the assumption of Theorem 2.1 no explicit bound on the number of linear pieces of γ^g (apart from the requirement that its thresholds are from $\mathbf{Q}_a$). For example, these activation functions may consist of 2^a linear pieces (with discontinuous jumps in between). Furthermore γ^g is not required to be monotone.

Finally, it should be mentioned that a corresponding version of Theorem 2.1 also holds for rational numbers that do not have a finite binary representation, i.e., for all rationals from $\mathbf{Q}'_a := \{r \in \mathbf{Q} : r$ is the quotient of integers of bit-length $\leq a\}$ instead of $\mathbf{Q}_a$.

Remark 2.3. Previously, one had *no* upper bound for the computational power (or for the VC dimension) of multilayer neural nets $\mathcal{N}$ with arbitrary weights and analog computational elements (i.e., activation functions with non-Boolean output). Theorem 2.1 implies that any $\mathcal{N}$ of the considered type can compute with the help of arbitrary parameter assignments $\underline{\alpha} \in \mathbf{R}^w$ at most $2^{O(aw^2 \log w)}$ different functions from $\mathbf{Q}_a^n$ into $\{0,1\}$, hence VC dimension $(\mathcal{N}, \mathbf{Q}_a^n) = O(w^2(a + \log w))$ (see Remark 3.9 for a slightly better bound and for a related bound for the case of inputs from $\mathbf{R}^n$).

Furthermore Theorem 2.1 implies that one can *replace* all *analog computations inside* $\mathcal{N}$ by *digital arithmetical operations on not too large integers* (the proof gives an upper bound of $O(wa + w \log w)$ for their bit-length). It is well known that each of

these digital arithmetical operations (multiple addition, multiplication, division) can be carried out on a circuit of small constant depth with $O(a^{O(1)} \cdot w^{O(1)})$ MAJORITY-gates, hence also on a network architecture of depth $O(1)$ and size $O(a^{O(1)} \cdot w^{O(1)})$ with Heaviside gates and weights from $\{-1, 0, 1\}$ (see [CSV], [PS], [HMPST], [GHR], [SR], [SBKH]). Thus one can simulate for inputs from $\{0,1\}^n$ any depth d network architecture $\mathcal{N}$ as in Theorem 2.1 with arbitrary parameter assignments $\underline{\alpha} \in \mathbf{R}^w$ by a network architecture of depth $O(d)$ and size $O(a^{O(1)} \cdot w^{O(1)})$ with Heaviside gates and weights from $\{-1, 0, 1\}$. The same holds for inputs from $\mathbf{Q}_a^n$ if they are given to $\mathcal{N}$ in digital form.

The size of this simulating digital neural net with Heaviside gates is polynomial in the number of real-valued parameters of the simulated analog neural net $\mathcal{N}$ but exponential in the depth of $\mathcal{N}$. Subsequent to [M93a], Koiran [K93] has proven a complementary simulation result, where the size of the simulating digital neural net is exponential in the number of real-valued parameters in $\mathcal{N}$ but subexponential in the depth of $\mathcal{N}$.

Proof of Theorem 2.1. In the special case where $\gamma^g = \mathcal{H}$ for all gates in $\mathcal{N}$ this result is well known [Mu]. It follows by applying separately to each gate in $\mathcal{N}$ the following result.

LEMMA 2.4 (folklore; see [MT] for a proof). *Consider a system $A\underline{x} \leq \underline{b}$ of some arbitrary finite number of linear inequalities in l variables. Assume that all entries in A and $\underline{b}$ are integers of absolute value $\leq K$.*

If this system has any solution in $\boldsymbol{R}^l$, then it has a solution of the form $\langle \frac{s_1}{t}, \ldots, \frac{s_l}{t} \rangle$, where $s_1, \ldots, s_l, t$ are integers of absolute value $\leq (2l+1)!\, K^{2l+1}$.

Sketch of the proof for Lemma 2.4. Let k be the number of inequalities in $A\underline{x} \leq \underline{b}$. One writes each variable in $\underline{x}$ as a difference of two nonnegative variables and adds to each inequality a "slack variable." In this way one gets an equivalent system

$$A'\underline{x}' = \underline{b}, \quad \underline{x}' \geq \underline{0}, \tag{1}$$

over $l' := 2l + k$ variables for some $k \times l'$ matrix A'. The k columns of A' for the k slack variables in $\underline{x}'$ form an identity matrix. Hence A' has rank k.

The assumption of the lemma implies that (1) has a solution over $\mathbf{R}$. Hence by Carathéodory's theorem (Corollary 7.1i in [Sch]) one can conclude that there is also a solution over $\mathbf{R}$ of a system

$$A''\underline{x}'' = \underline{b}, \quad \underline{x}'' \geq \underline{0}. \tag{2}$$

Subsequent to the first publication of the techniques of this article in [M93a], Koiran [K93] has proven a complementary result without assumption that the depth is bounded but where one has to assume that the number of real-valued parameters in the given neural net $\mathcal{N}$ is bounded by a constant where A'' consists of k linearly independent columns of A'. Since A'' has full rank, (2) has in fact a unique solution that is given by Cramer's rule: $x_j'' = \det(A_j'')/\det A''$ for $j = 1, \ldots, k$, where A_j'' results form A'' by replacing its jth column by $\underline{b}$. Since all except up to $2l$ columns of A'' contain exactly one 1 and else only 0's, we can bring each of the matrices A'', A_j'' by permutations of rows and columns into a form

$$B = \begin{pmatrix} C & 0 \\ D & I \end{pmatrix},$$

where C is a square matrix with $2l+1$ rows. Hence the determinant of B is an integer of absolute value $\leq (2l+1)!\, K^{2l+1}$. □

The difficulty of the proof of Theorem 2.1 lies in the fact that with *analog* computational elements one can no longer treat each gate separately, since intermediate values are no longer integers. Furthermore, the total computation of $\mathcal{N}$ can in general *not* be described by a system of *linear* inequalities, where the w variable parameters of $\mathcal{N}$ are the variables in the inequalities (and the fixed parameters of $\mathcal{N}$ are the constants). This becomes obvious if one just considers the composition of two very simple analog gates g_1 and g_2 on levels 1 and 2 of $\mathcal{N}$, whose activation functions γ_1, γ_2 satisfy $\gamma_1(y) = \gamma_2(y) = y$. Assume $x = \sum_{i=1}^{n} \alpha_i x_i + \alpha_0$ is the input to gate g_1, and g_2 receives as input $\sum_{j=1}^{m} \alpha'_j y_j + \alpha'_0$, where $y_1 = \gamma_1(x) = x$ is the output of gate g_1. Then g_2 outputs $\alpha'_1 \cdot (\sum_{i=1}^{n} \alpha_i x_i + \alpha_0) + \sum_{j=2}^{m} \alpha'_j y_j + \alpha'_0$. Obviously this term is not linear in the weights $\alpha'_1, \alpha_1, \ldots, \alpha_n$. Hence if the output of gate g_2 is compared with a fixed threshold at the next gate, the resulting inequality is not linear in the weights of the gates in $\mathcal{N}$.

If the activation functions of all gates in $\mathcal{N}$ were linear (as in the example for g_1 and g_2), then there would be no problem because a composition of linear functions is linear. However for *piecewise*-linear activation functions it is not sufficient to consider their composition, since intermediate results have to be compared with boundaries between linear pieces of the next gate.

We introduce in this paper a new method in order to handle this difficulty. We simulate $\mathcal{N}^{\underline{\alpha}}$ by another neural net $\hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}$ (which one may view as a "normal form" for $\mathcal{N}^{\underline{\alpha}}$) that uses the same graph $\langle V, E\rangle$ as $\mathcal{N}$ but different activation functions and different values $\underline{\beta}$ for its variable parameters. The activation functions of $\hat{\mathcal{N}}[\underline{c}]$ depend on $|V|$ new parameters $\underline{c} \in \mathbf{R}^{|V|}$, which we call *scaling parameters* in the following. Although this new neural net has the disadvantage that it requires $|V|$ additional parameters $\underline{c}$, it has the advantage that we can choose in $\hat{\mathcal{N}}[\underline{c}]$ all weights on edges between computation nodes to be from $\{-1, 0, 1\}$. Since these weights from $\{-1, 0, 1\}$ are already of the desired bit-length, we can treat them as constants in the system of inequalities that describes computations of $\hat{\mathcal{N}}[\underline{c}]$. Therefore, all variables that appear in the inequalities that describe computations of $\hat{\mathcal{N}}[\underline{c}]$ (the variables for weights of gates on level 1, the variables for the biases of gates on all levels, the variable for the outer threshold, *and the new variables for the scaling parameters* $\underline{c}$) appear only *linearly* in those inequalities. Hence we can apply Lemma 2.4 to the system of inequalities that describes the computations of $\hat{\mathcal{N}}$ for inputs from $\mathbf{Q}_a^n$ and thereby get a rational solution $\underline{\beta}'$, $\underline{c}'$ for all variable parameters in $\hat{\mathcal{N}}$. Finally we observe that we can transform $\hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'}$ back into the original neural net $\mathcal{N}$ with an assignment of rational numbers $\underline{\alpha}'$ to all variable parameters in $\mathcal{N}$.

We will now fill in some of the missing details. Consider the gate function γ of an arbitrary gate g in $\mathcal{N}$. Since γ is piecewise-linear, there are fixed parameters $t_1 < \cdots < t_k, a_0, \ldots, a_k, b_0, \ldots, b_k$ in $\mathbf{Q}_a$ (which may be different for different gates g) such that with $t_0 := -\infty$ and $t_{k+1} := +\infty$ one has $\gamma(x) = a_i x + b_i$ for $x \in \mathbf{R}$ with $t_i \leq x < t_{i+1}$; $i = 0, \ldots, k$. For an arbitrary scaling parameter $c \in \mathbf{R}^+$ we associate with γ the following piecewise-linear activation function γ^c: the thresholds of γ^c are $c \cdot t_1, \ldots, c \cdot t_k$, and its output is $\gamma^c(x) = a_i x + c \cdot b_i$ for $x \in \mathbf{R}$ with $c \cdot t_i \leq x < c \cdot t_{i+1}$; $i = 0, \ldots, k$ (set $c \cdot t_0 := -\infty$, $c \cdot t_{k+1} := +\infty$). Thus for all reals $c > 0$ the function γ^c is related to γ through the equality: $\forall\, x \in \mathbf{R}(\gamma^c(c \cdot x) = c \cdot \gamma(x))$.

Assume that $\underline{\alpha} \in \mathbf{R}^w$ is some arbitrarily given assignment to the variable parameters in $\mathcal{N}$. We transform $\mathcal{N}^{\underline{\alpha}}$ into a normal form $\hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}$, in which all weights on edges between computation nodes are from $\{-1, 0, 1\}$ such that $\forall\, \underline{x} \in \mathbf{R}^n(\mathcal{N}^{\underline{\alpha}}(\underline{x}) = \hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}(\underline{x}))$. We proceed inductively from the output level towards the input level.

Assume that the output gate g_{out} of $\mathcal{N}^{\underline{\alpha}}$ receives as input $\sum_{i=1}^{m} \alpha_i y_i + \alpha_0$, where $\alpha_1, \ldots, \alpha_m, \alpha_0$ are the weights and the bias of g_{out} (under the assignment $\underline{\alpha}$) and $y_1, \ldots, y_m$ are the (real-valued) outputs of the immediate predecessors $g_1, \ldots, g_m$ of g. For each $i \in \{1, \ldots, m\}$ with $\alpha_i \neq 0$ such that g_i is not an input node we replace the activation function γ_i of g_i by $\gamma_i^{|\alpha_i|}$, and we multiply the weights and the bias of gate g_i with $|\alpha_i|$. Finally we replace the weight α_i of gate g_{out} by

$$sgn(\alpha_i) := \begin{cases} 1 & \text{if } \alpha_i > 0, \\ -1 & \text{if } \alpha_i < 0. \end{cases}$$

This operation has the effect that the multiplication with $|\alpha_i|$ is carried out *before* the gate g_i (rather than after g_i, as done in $\mathcal{N}^{\underline{\alpha}}$) but that the considered output gate g_{out} still receives the same input as before. The analogous operation is then inductively carried out for the predecessors g_i of g_{out} (note, however, that the weights of g_i are no longer the original ones from $\mathcal{N}^{\underline{\alpha}}$, since they have been changed in the preceding step). We exploit here the assumption that each gate has fan-out ≤ 1.

Let $\underline{\beta}$ consist of the new weights on edges adjacent to input nodes, the resulting biases of all gates in $\hat{\mathcal{N}}$, and the (unchanged) outer threshold T_{out}. Let $\underline{c}$ consist of the resulting scaling factors at the gates of $\mathcal{N}$. Then we have $\forall\, \underline{x} \in \mathbf{R}^n (\mathcal{N}^{\underline{\alpha}}(\underline{x}) = \hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}(\underline{x}))$.

Finally we have to replace all *strict* inequalities of the form "$s_1 < s_2$" that are needed to describe the computation of $\hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}$ for some input $\underline{x} \in \mathbf{Q}_a^n$ by inequalities of the form "$s_1 + 1 \leq s_2$". This concerns inequalities of the form $s < c \cdot t_i$, where $c \cdot t_i$ is the threshold of some gate g in $\hat{\mathcal{N}}[\underline{c}]$ and s is its gate input, inequalities of the form $s < T_{\text{out}}$ where s is the output of g_{out}, and inequalities of the form $0 < c$ for each scaling parameter c. In order to achieve this stronger separation it is sufficient to multiply all parameters $\underline{\beta}$, $\underline{c}$ in $\hat{\mathcal{N}}$ by a sufficiently large constant K. For simplicity we write again $\underline{\beta}$, $\underline{c}$ for the resulting parameters. We now specify a system $\mathcal{A}\underline{z} \leq \underline{b}$ of linear inequalities in w variables $\underline{z}$ that play the role of the w parameters $\underline{\beta}$, $\underline{c}$ in the computations of $\hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}$ for all inputs $\underline{x}$ from $\mathbf{Q}_a^n$. The constants of these inequalities are the coordinates of all inputs $\underline{x} \in \mathbf{Q}_a^n$, the parameters of the activation functions γ in $\mathcal{N}$, the constants $-1, 1$ that occur in $\hat{\mathcal{N}}$ as weights of edges between computation nodes, and the constants 1 that arise from the replacement of strict inequalities "$s_1 < s_2$" by "$s_1 + 1 \leq s_2$."

For each fixed input $\underline{x} \in \mathbf{Q}_a^n$ one places into the system $\mathcal{A}\underline{z} \leq \underline{b}$ up to two linear inequalities for each gate g in $\mathcal{N}$. These inequalities are defined by induction on the depth of g. If g has depth 1, $t_1 < \cdots < t_k$ are the thresholds of its activation functions γ in $\mathcal{N}$, and its input $\sum_{i=1}^{n} \alpha_i x_i + \alpha_0$ in $\hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}$ satisfies $c \cdot t_j \leq \sum_{i=1}^{n} \alpha_i x_i + \alpha_0$ and $\sum_{i=1}^{n} \alpha_i x_i + \alpha_0 + 1 \leq c \cdot t_{j+1}$, then one adds these two inequalities to the system (more precisely, if $j = 0$ or $j = k$ then only one inequality is needed since the other one is automatically true).

If g' is a successor gate of g, it receives from g for some specific $j \in \{0, \ldots, k\}$ an output of the form $a_j \cdot (\sum_{i=1}^{n} \alpha_i x_i + \alpha_0) + c \cdot b_j$ (where c is the scaling factor of gate g). Note that this term is linear, since a_j, b_j are fixed parameters of gate g'. In this way one can express for circuit input $\underline{x}$ the input $I(\underline{x})$ of gate g' as a linear term in the weights, biases, and scaling factors of its preceding gates. (We exploit here that in $\hat{\mathcal{N}}$ the weight on the edge between g' and each predecessor gate is a fixed parameter from $\{-1, 0, 1\}$, not a variable.) If this input $I(\underline{x})$ satisfies in $\hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}$ the inequalities $c' \cdot t'_{j'} \leq I(\underline{x})$ and $I(\underline{x}) + 1 \leq c' \cdot t'_{j'+1}$ (where $t'_1 < \cdots < t'_{k'}$ are the thresholds of g' in $\mathcal{N}$, and c' is the scaling factor of g' in $\hat{\mathcal{N}}$), then one adds these two

inequalities to the system $\mathcal{A}\underline{z} \leq \underline{b}$ (respectively, only one if $j' = 0$ or $j' = k'$). Note that all resulting inequalities are linear, in spite of the fact that the system contains variables for the biases of *all* gates. It should also be pointed out that the definition of this system of inequalities is more involved than it may first appear, since the sum of terms $I(\underline{x})$ depends on the chosen inequalities for all predecessor gates (e.g., on j in the example above). Hence a precise definition has to be similar to that of the proof of Theorem 3.1.

It is clear that the resulting system $\mathcal{A}\underline{z} \leq \underline{b}$ has a solution in $\mathbf{R}^w$, since $\underline{z} := \langle \underline{\beta}, \underline{c} \rangle$ is a solution. Hence we can apply Lemma 2.4, which provides a solution $\underline{z}'$ of the form $\langle \frac{s_i}{t} \rangle_{i=1,\ldots,w}$ with integers $s_1, \ldots, s_w, t$ of absolute value $\leq (2w+1)!\ 2^{2a(2w+1)}$. Let $\hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'}$ be the neural net $\hat{\mathcal{N}}$ with this new assignment $\langle \underline{\beta}', \underline{c}' \rangle := \underline{z}'$ of "small" parameters. By definition we have $\forall\, \underline{x} \in \mathbf{Q}_a^n (\mathcal{N}^{\underline{\alpha}}(\underline{x}) = \hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'})$. We show that one can transform this neural net $\hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'}$ into a net $\mathcal{N}^{\underline{\beta}'}$ with the *same activation functions* as $\mathcal{N}^{\underline{\alpha}}$ but a new assignment $\underline{\alpha}'$ of rational parameters (that can easily be computed from $\underline{\beta}'$, $\underline{c}'$). This transformation proceeds inductively from the input level towards the output level. Consider some gate g on level 1 in $\hat{\mathcal{N}}$ that uses (for the new parameter assignment $\underline{c}'$) the scaling factor $c > 0$ for its activation function γ^c. Then we replace the weights $\alpha_1, \ldots, \alpha_n$ and bias α_0 of gate g in $\hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'}$ by $\frac{\alpha_i}{c}, \ldots, \frac{\alpha_n}{c}, \frac{\alpha_0}{c}$ and γ^c by γ. Furthermore if $r \in \{-1, 0, 1\}$ was in $\hat{\mathcal{N}}$, the weight on the edge between g and its successor gate g, we assign to this edge the weight $c \cdot r$. Note that g' receives in this way from g the same input as in $\hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'}$ (for every circuit input). Assume now that $\alpha'_1, \ldots, \alpha'_m$ are the weights that the incoming edges of g' get assigned in this way, that α'_0 is the bias of g' in the assignment $\underline{z}' = \langle \underline{\beta}', \underline{c}' \rangle$, and that $c' > 0$ is the scaling factor of g' in $\hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'}$. Then we assign the new weights $\frac{\alpha'_1}{c'}, \ldots, \frac{\alpha'_m}{c'}$ and the new bias $\frac{\alpha'_0}{c'}$ to g', and we multiply the weight on the outgoing edge from g' by c'.

By construction we have that $\forall\, \underline{x} \in \mathbf{R}^n (\mathcal{N}^{\underline{\alpha}'}(\underline{x}) = \hat{\mathcal{N}}[\underline{c}']^{\underline{\beta}'}(\underline{x}))$; hence $\forall\, \underline{x} \in \mathbf{Q}_a^n (\mathcal{N}^{\underline{\alpha}'}(\underline{x}) = \mathcal{N}^{\underline{\alpha}}(\underline{x}))$. □

3. Upper bounds for neural nets with piecewise-polynomial activation functions.

THEOREM 3.1. *Consider an arbitrary array $(\mathcal{N}_n)_{n \in \boldsymbol{N}}$ of high-order network architectures $\mathcal{N}_n$ of depth $O(1)$ with n inputs and $O(n^{O(1)})$ gates in which the gate function γ^g of each gate g is piecewise-polynomial of degree $O(1)$ with $O(n^{O(1)})$ polynomial pieces, with arbitrary reals as architectural parameters.*

Then there exists an array $(\hat{\mathcal{N}}_n)_{n \in \boldsymbol{N}}$ of first-order network architectures $\hat{\mathcal{N}}_n$ of depth $O(1)$ with n inputs and $O(n^{O(1)})$ gates such that each gate g in $\hat{\mathcal{N}}_n$ uses as its activation function the Heaviside function $\mathcal{H}$ (i.e., g is a linear threshold gate) and such that for each assignment $\underline{\alpha}_n$ of arbitrary reals to the programmable parameters in $\mathcal{N}_n$ there is an assignment $\tilde{\underline{\alpha}}_n$ of $O(n^{O(1)})$ numbers from $\{-1, 0, 1\}$ to the programmable parameters in $\hat{\mathcal{N}}_n$ such that $\forall\, \underline{x} \in \{0,1\}^n (\mathcal{N}_n^{\underline{\alpha}_n}(\underline{x}) = \tilde{\mathcal{N}}_n^{\tilde{\underline{\alpha}}_n}(\underline{x}))$.

Hence for any assignment $(\underline{\alpha}_n)_{n \in \boldsymbol{N}}$ of real-valued parameters the Boolean functions that are computed by $(\mathcal{N}_n^{\underline{\alpha}_n})_{n \in \boldsymbol{N}}$ are in TC^0. In particular, VC dimension $(\mathcal{N}_n, \{0,1\}^n) = O(n^{O(1)})$.

Remark 3.2.

(a) The proof of Theorem 3.1 shows that one can replace in its claim the Boolean domain $\{0,1\}^n$ by $\{-K, \ldots, K\}^n$ for any $K \in \mathbf{N}$.

(b) Theorem 3.1 yields no bound for the computational power of neural nets with the activation function $\sigma(y) = 1/(1 + e^{-y})$. However it provides bounds for

the case where the activation functions are spline approximations to σ of arbitrarily high degree d, provided that $d \in \mathbf{N}$ is fixed.

Proof of Theorem 3.1. This proof is quite long and involved, even for the simplest nonlinear case where the activation functions consist of two polynomial pieces of degree 2. Note that in contrast to the model in [SS] the magnitude of the given weights in $\mathcal{N}_n$ may grow arbitrarily fast as a function of n.

We first note that one can eliminate all nonlinear polynomials $\mathbf{Q}^g$ as arguments of activation functions by introducing intermediate gates with linear gate inputs and quadratic activation functions. One exploits here the obvious fact that $y \cdot z = \frac{1}{2}((y + z)^2 - y^2 - z^2)$. In this way one can transform the given network architectures into *first-order* network architectures which still satisfy the assumptions of Theorem 3.1. One should note, however, that this transformation does *not* affect the degrees of polynomial pieces in the activation functions.

Subsequently we transform each given network architecture $\mathcal{N}_n$ into a normal form $\hat{\mathcal{N}}_n$ of constant depth and size $O(n^{O(1)})$ in which all gates g have fan-out ≤ 1 and in which all gates g use as activation functions γ^g piecewise-polynomial functions of the following special type: γ^g consists of up to three pieces, of which at most one is not identically 0 and in which the nontrivial piece outputs the constant 1 or computes a power $y \mapsto y^k$ (where $k \in \mathbf{N}$ satisfies $k = O(1)$). The preceding "normalization" of activation functions is easy to achieve, since every activation function of a gate in $\mathcal{N}_n$ can be written as linear combination of activation functions of this normalized type. The transformation from $\mathcal{N}_n$ to $\hat{\mathcal{N}}_n$ can be carried out in such a way that for every assignment $\underline{\alpha}_n$ of real values to the programmable parameters of $\mathcal{N}_n$ there exists an assignment $\underline{\beta}_n$ of real numbers to the programmable parameters of $\hat{\mathcal{N}}_n$ such that

$$\forall \underline{x} \in \{0,1\}^n (\mathcal{N}_n^{\underline{\alpha}_n}(\underline{x}) = \hat{\mathcal{N}}_n^{\underline{\beta}_n}(\underline{x})),$$

and such that any strict inequality "$s_1 < s_2$" that arises in the computation of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ for some input $\underline{x} \in \{0,1\}^n$ (when one compares some subresult of that computation with a threshold of the activation function of some gate or with the outer threshold of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$) can be replaced by the stronger inequality "$s_1 + 1 \leq s_2$."

It would also be possible to push all nontrivial weights to the gates on level 1 in correspondence to the construction in the proof of Theorem 2.1. However, in the present context this additional operation does not eliminate nonlinear conditions on the weights. Assume for example that g is a gate on level 1 with input $\alpha_1 x_1 + \alpha_2 x_2$ and activation function $\gamma^g(y) = y^2$. Then this gate g outputs $\alpha_1^2 x_1^2 + 2\alpha_1\alpha_2 x_1 x_2 + \alpha_2^2 x_2^2$. Hence the variables α_1, α_2 will not occur linearly in an inequality which describes the comparison of the output of g with some threshold of a gate at the next level.

Although it does not eliminate nonlinear conditions on the weights if one pushes all weights toward level 1, the resulting network provides some notational advantage because all weights between computation nodes can be treated as constants (with three possible values). This approach has therefore been chosen in [M92] and [M93a]. However, this approach is disadvantageous if one wants to apply the method of this proof in the context of agnostic PAC learning on analog neural nets [M93c]. In this application one has to be able to control the bit-length of the (rational) weights. Therefore, one cannot afford to push all weights toward level 1, since this may increase the bit-length of weights in an unbounded manner. For example, if one pushes the weight 2 through a gate g with activation function $\gamma^g(y) = y^2$, then this weight is changed to $\sqrt{2}$ (since $2\gamma^g(y) = \gamma^g(\sqrt{2} \cdot y)$).

Since the nonlinearity of the conditions on the weights cannot be eliminated in the same way as for Theorem 2.1, we have to introduce an alternative method. We fix an arbitrary assignment $\underline{\beta}_n$ of real numbers to the programmable parameters of $\hat{\mathcal{N}}_n$. We introduce for the system of inequalities $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ (that describes the computations of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ for all inputs $\underline{x} \in \{0,1\}^n$) new variables v for all nontrivial parameters in $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ (i.e., for the weights and bias of each gate g, for the outer threshold T_{out}, and for the *thresholds* t_1^g, t_2^g *of each gate* g). In addition we introduce new variables for all *products* of such parameters that arise in the computation of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$. We have to keep the inequalities linear in order to apply Lemma 2.4. Hence we cannot demand in these inequalities that the value of the variable $v_{v_1^g, v_2^g}$ (that represents the product of α_1^g and α_2^g) is the product of the values of the variables v_1^g and v_2^g (that represent the weights α_1^g, respectively, α_2^g). We solve this problem by describing in detail in the linear inequalities $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ which *role* the product of α_1^g and α_2^g plays in the computations of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ for inputs from $\{0,1\}^n$. It turns out that this can be done in such a way that it does not matter whether a solution $\mathcal{A}$ of $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ assigns to the variable $v_{v_1^g, v_2^g}$ a value $\mathcal{A}(v_{v_1^g, v_2^g})$ that is equal to the product of the values $\mathcal{A}(v_1^g)$ and $\mathcal{A}(v_2^g)$ (that are assigned by $\mathcal{A}$ to the variables v_1^g and v_2^g). In any case $\mathcal{A}(v_{v_1^g, v_2^g})$ is forced to behave like the product of $\mathcal{A}(v_1^g)$ and $\mathcal{A}(v_2^g)$ in the computations of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$.

We would like to emphasize that the parameters $\underline{\beta}_n$ do not occur as constants in the system $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ of inequalities. They are also replaced by variables. The reason the real-valued parameters $\underline{\beta}_n$ occur nevertheless in our notation $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ of inequalities is the following. These inequalities consist of conditions which demand that for any input $\underline{x} \in \{0,1\}^n$ the computation on the neural net proceeds exactly as for the parameter assignment $\underline{\beta}_n$ (i.e., the same inequalities with thresholds of the piecewise-polynomial activation functions are satisfied and the same pieces of the activation functions are used at each gate as in the computation with parameter assignment $\underline{\beta}_n$).

Before we present the formal definitions and proofs, we give a high-level description of the proof idea and the purpose of the formal definitions. These preceding informal remarks should be real *interactively* with the subsequent formal part.

In more abstract terms, one may view any solution $\mathcal{A}$ of $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ as a model of a certain linear fragment $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ of the theory of the role of the parameters $\underline{\beta}_n$ in the computations of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ on inputs from $\{0,1\}^n$. Such a model $\mathcal{A}$ (which will be given by Lemma 2.4) may be viewed as some type of nonstandard model of the theory of computations of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$, since it replaces products of weights by values that one might call nonstandard products. Such a nonstandard model $\mathcal{A}$ does not provide a new assignment of (small) weights to the network architecture $\hat{\mathcal{N}}_n$, only to a nonstandard version $\mathcal{M}_n^{\mathcal{A}}$ of the neural net $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$. However the linear fragment $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \{0,1\}^n)$ can be chosen in such a way that $\mathcal{M}_n^{\mathcal{A}}$ computes the same Boolean function as $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$. Furthermore, if $\mathcal{A}$ consists of a solution with rational numbers as given by Lemma 2.4, then $\mathcal{M}_n^{\mathcal{A}}$ can be simulated by a constant-depth polynomial-size Boolean circuit whose gates g are all MAJORITY-gates (i.e., $g(y_1, \ldots, y_m) = 1$ if $\sum_{i=1}^m y_i \geq m/2$; otherwise $g(y_1, \ldots, y_m) = 0$). This implies that the Boolean functions

that are computed by $(\mathcal{M}_n^{\mathcal{A}})_{n\in\mathbf{N}}$ are in TC^0. However, by construction these are the same Boolean functions that are computed by $(\mathcal{N}_n^{\underline{\alpha}_n})_{n\in\mathbf{N}}$.

We will now describe the details of the previously sketched proof of Theorem 3.1. We will simply write $\mathcal{N}$ instead of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ (where $\underline{\beta}_n$ is some assignment of real numbers to the programmable parameters of the network architecture $\hat{\mathcal{N}}_n$). We will define for each gate g in $\mathcal{N}$ by induction on the depth of g,

(1) in Definition 3.3 a set V^g of variables and a set M^g of formal terms that are needed to describe the operation of gate g.

[The intuition is here that one writes for any network input $\underline{x}$ the output of g as a sum of products (of programmable parameters, of architectural parameters, and of components of $\underline{x}$). *Which* of these terms will occur for a specific circuit input $\underline{x}$ will depend on the course of the computation in $\mathcal{N}$ up to gate g: for different inputs the involved gates may use different pieces of their activation function. The set M^g contains a separate formal term for each product that *may* possibly occur in this sum. Each term in M^g consists of a variable $w \in V^g$ (that represents a programmable or architectural parameter of $\mathcal{N}$ or some product of these) and of a product $P \equiv \pm \boldsymbol{x}_1^{j_1} \cdot \ldots \cdot \boldsymbol{x}_n^{j_n}$ of input *variables* $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$.]

(2) in Definition 3.4 for any fixed network input $\underline{x} \in \mathbf{R}^n$ a set $L^g(\underline{x})$ of linear inequalities associated with gate g (with variables from $V^{\mathcal{N}} := \cup\{V^{g'} | g'$ is a gate of $\mathcal{N}\}$) that hold for the computation of $\mathcal{N}$ on input $\underline{x}$ if all formal terms $t \in M^g$ are replaced by their actual value $W(t, \underline{x})$ for the given parameter assignment in $\mathcal{N}$. We also define in Definition 3.4 a set $S^g(\underline{x})$ of formal terms whose sum represents the input of g and a set $T^g(\underline{x}) \subseteq M^g$ of formal terms whose sum represents the output of g for circuit input $\underline{x}$.

[$L^g(\underline{x})$ specifies in particular which piece of γ^g is used by gate g for network input $\underline{x}$.]

(3) in Definition 3.6 for any input set $S \subseteq \mathbf{R}^n$, any solution $\mathcal{A}$ of the resulting system $L(\mathcal{N}, S) := \cup\{L^g(\underline{x}) \mid \underline{x} \in S$ and g is a gate in $\mathcal{N}\}$ of linear inequalities, and any term $t \in M^g$ a network architecture $\mathcal{M}_{g,t}^{\mathcal{A}}$ that decides for any network input $\underline{x} \in S$ whether t occurs as a summand in the output of g in $\mathcal{N}$.

[For any input $\underline{x} \in S$ the network architectures $(\mathcal{M}_{g,t}^{\mathcal{A}})_{t\in M^g}$ together compute the characteristic function of the set $T^g(\underline{x}) \subseteq M^g$ which represents the output of gate g in $\mathcal{N}$. In this way one can replace in a recursive manner the analog computations in $\mathcal{N}$ by digital manipulations of formal terms, with "nonstandard products" of weights in place of real products.]

One verifies in Lemma 3.5 that $L(\mathcal{N}, S)$ describes correctly the role of the parameters $\underline{\beta}_n$ in the computations of $\mathcal{N} := \hat{\mathcal{N}}_n^{\underline{\beta}_n}$ for inputs $\underline{x} \in S$. Unfortunately, $L(\mathcal{N}, S)$ does not provide a complete description of the properties of the parameters $\underline{\beta}_n$ in these computations, since it represents only a linear fragment of their theory. Nevertheless, one can prove with the help of Lemmas 3.7 and 3.8 that for *any* solution $\mathcal{A}$ of $L(\mathcal{N}, S)$ the network architectures $\mathcal{M}_{g,t}^{\mathcal{A}}$ carry out a truthful simulation of the corresponding initial segments of $\mathcal{N}$.

We would like to point out a difference to the proof of Theorem 2.1 regarding the treatment of architectural parameters. In the proof of Theorem 3.1 the programmable parameters $\underline{\alpha}_n$ of $\mathcal{N}_n^{\underline{\alpha}_n}$ and the architectural parameters of the given network architecture $\mathcal{N}_n$ (the thresholds of activation functions γ^g and the coefficients of the polynomial pieces of γ^g) are all changed simultaneously in the transformation to $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$. Consequently, $\underline{\beta}_n$ denotes the values of *all* nontrivial parameters in $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ (i.e., of all

programmable and architectural parameters). As a consequence of this treatment of parameters one can allow in the given network architectures $\mathcal{N}_n$ of Theorem 3.1 *arbitrary reals* as architectural parameters (i.e., for the thresholds and coefficients of the polynomial pieces of the given activation functions γ^g).

We refer to an analog network architecture $\mathcal{N}$ with the properties of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ as a network architecture in *normal form.* This means that $\mathcal{N}$ is a first-order network architecture whose gates have fan-out ≤ 1; all gates g in $\mathcal{N}$ use as activation function γ^g a piecewise-polynomial function that consists of three pieces, of which at most one piece is not identically 0 and in which the nontrivial piece (if it exists) outputs the constant 1 or computes a power $y \mapsto y^k$ for some $k \in \mathbf{N}$.

In order to simplify our notation, we assume that for a network architecture $\mathcal{N}$ in normal form the nontrivial piece of the activation function γ^g of each gate g is defined over a half-open interval $[t_1^g, t_2^g)$ with certain reals $t_1^g < t_2^g$. It is easy to see that the subsequent proof can also be carried out without this simplifying assumption. We also assume without loss of generality (w.l.o.g.) that $\mathcal{N}$ is leveled, i.e., each gate g in $\mathcal{N}$ has the property that all paths in $\mathcal{N}$ from an input node to g have the same length.

DEFINITION 3.3. *Assume that $\mathcal{N}$ is a network architecture in normal form with n input variables $x_1, \ldots, x_n$, where arbitrary reals have been assigned to all parameters of $\mathcal{N}$. We define by induction on the depth of g for each gate g in $\mathcal{N}$ a set V^g of variables, a value $W(v)$ for each variable $v \in V^g$ (that arises from the assignment $\underline{\beta}_n$ in $\hat{\mathcal{N}}_n^{\underline{\beta}_n} := \mathcal{N}$), and a set M^g of (formal) terms. Each element of M^g is of the form $v \cdot P$, where $v \in V^g$ is a variable and P is some formal polynomial term of the form $\pm \boldsymbol{x}_1^{j_1} \cdot \ldots \cdot \boldsymbol{x}_n^{j_n}$, with $j_1, \ldots, j_n \in \boldsymbol{N}$. The here-occurring* formal variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ *for the input components should be distinguished from the* concrete values $x_1, \ldots, x_n \in \boldsymbol{R}$ *for these variables that are considered later (starting in Definition* 3.4).

We consider first the case where g has depth 1. *If γ^g gives on its nontrivial piece $[t_1^g, t_2^g)$ the constant* 1 *as output, we set*

$$V^g := \{v_0^g, \ldots, v_n^g\} \cup \{v_{\text{const}}^g\} \cup \{v_{\text{I}}^g, v_{\text{II}}^g\} \text{ and } M^g := \{v_{\text{const}}^g\}.$$

We define $W(v_i^g) := \alpha_i^g$ for $i = 0, \ldots, n$, $W(v_{\text{const}}^g) := \alpha^g$, $W(v_{\text{I}}^g) := t_1^g$, and $W(v_{\text{II}}^g) := t_2^g$ ($\alpha_1^g, \ldots, \alpha_n^g$ are the weights and α_0^g is the bias of gate g in $\mathcal{N}$, α^g is the weight on the edge that leaves g, and t_1^g, t_2^g are the thresholds of the activation function γ^g). In the other case γ^g computes a power $y \mapsto y^k$ on its nontrivial piece. Then we introduce for each ktuple $\langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in (\{v_0^g\} \cup \{v_1^g \cdot \boldsymbol{x}_1, \ldots, v_n^g \cdot \boldsymbol{x}_n\})^k$ a new variable $v_{w_1,\ldots,w_k}^g$ in V^g and a term $v_{w_1,\ldots,w_k}^g \cdot \prod_{i=1}^k P_i$ in M^g. We assume here that a formal multiplication $P \cdot P'$ for formal terms P, P' of the form $\pm \boldsymbol{x}_1^{j_1} \cdot \ldots \cdot \boldsymbol{x}_n^{j_n}$ is defined in the obvious way. We define

$$V^g := \{v_0^g, \ldots, v_n^g\} \cup \{v_{\text{I}}^g, v_{\text{II}}^g\} \cup \{v_{w_1,\ldots,w_k}^g \mid \langle w_1, \ldots, w_k \rangle \in \{v_0^g, \ldots, v_n^g\}^k\}.$$

We set $W(v_{w_1,\ldots,w_k}^g) := \alpha^g \cdot \prod_{i=1}^k W(w_i)$, and we define $W(v)$ for the other variables as before. We define

$$M^g := \left\{ v_{w_1,\ldots,w_k}^g \cdot \prod_{i=1}^k P_i \mid \langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in (\{v_0^g\} \cup \{v_1^g \cdot \boldsymbol{x}_1, \ldots, v_n^g \cdot \boldsymbol{x}_n\})^k \right\}.$$

[The terms in M^g denote the summands that one gets from the output $(\alpha_0^g + \sum_{i=1}^m \alpha_i^g \cdot x_i)^k$ of γ^g by multiplying this output with the weight α^g on the next edge and then rewriting it as a sum of products.]

We now consider the case where g is a gate on level $l+1$, with edges from the gates $g_1, \ldots, g_m$ on level l leading into g. Assume that $\alpha_1^g, \ldots, \alpha_m^g$ are in $\mathcal{N}$, the weights on these edges, that α^g is the weight on the edge out of g, and that α_0^g is the bias of g. If g is an output gate (i.e., g has fan-out 0) then we set $\alpha^g := 1$. If γ^g outputs the constant 1 on its nontrivial piece, we set

$$V^g := \{v_0^g, v_{\text{const}}^g\} \cup \{v_{\text{I}}^g, v_{\text{II}}^g\} \text{ and } M^g := \{v_{\text{const}}^g\}.$$

We set $W(v_0^g) := \alpha_0^g$, $W(v_{\text{const}}^g) := \alpha^g, W(v_{\text{I}}^g) := t_1^g$, and $W(v_{\text{II}}^g) := t_2^g$. If γ^g computes the power $y \mapsto y^k$ on its nontrivial piece, we introduce for each $k$tuple

$$\langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in \left(\{v_0^g\} \cup \bigcup_{j=1}^m M^{g_j} \right)^k$$

a new variable $v_{w_1,\ldots,w_k}^g$ in V^g and a term $v_{w_1,\ldots,w_k}^g \cdot \prod_{i=1}^k P_i$ in M^g. Thus we set

$$V^g := \{v_0^g\} \cup \{v_{\text{I}}^g, v_{\text{II}}^g\} \cup \{v_{w_1,\ldots,w_k}^g \mid \langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in \left(\{v_0^g\} \cup \bigcup_{j=1}^m M^{g_j} \right)^k,$$

for arbitrary polynomial terms $P_1, \ldots, P_k$ and variables $w_i \in (\{v_0^g\} \cup \bigcup_{j=1}^m V^{g_j})\}$.

We define $W(v_{w_1,\ldots,w_k}^g) := \alpha^g \cdot \prod_{i=1}^k W(w_i)$, $W(v_0^g) := \alpha_0^g$, $W(v_{\text{I}}^g) := t_1^g$, $W(v_{\text{II}}^g) := t_2^g$.
We set

$$M^g := \left\{ v_{w_1,\ldots,w_k}^g \cdot \prod_{i=1}^k P_i \mid \langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in \left(\{v_0^g\} \cup \bigcup_{j=1}^m M^{g_j} \right)^k \right\}.$$

[The argument of γ^g is a sum of α_0^g and of summands that are denoted by terms in $\cup_{j=1}^m M^{g_j}$. Hence the terms in M^g correspond to the summands that one gets by multiplying the output of γ^g with the weight on the next edge and then rewriting this product as a sum of products by multiplying out.]

Finally, for the output gate g_{out} of $\mathcal{N}$, we place into V^g in addition the variable $v^{g_{\text{out}}}$. We define $W(v^{g_{\text{out}}})$ as the value of the outer threshold of $\mathcal{N}$.

DEFINITION 3.4. *Assume that $\mathcal{N}$ is a network architecture in normal form with n input variables and some fixed assignment of reals to its parameters. Let $\underline{x} \in \boldsymbol{R}^n$ be a fixed input for $\mathcal{N}$. We define for each gate g in $\mathcal{N}$ by simultaneous induction on the depth of g*

(1) *a set $L^g(\underline{x})$ of inequalities (that are linear in the variables from V^G),*

(2) *a set $S^g(\underline{x})$ of formal terms (whose sum represents the argument of γ^g for network input $\underline{x}$),*

(3) *a set $T^g(\underline{x}) \subseteq M^g$ (whose sum represents the output of g for network input $\underline{x}$ after multiplication with the weight on the next edge).*

Since $\underline{x}$ is now a fixed element of $\boldsymbol{R}^n$, one can assign a specific value $W(P, \underline{x}) \in \boldsymbol{R}$ to each term P of the form $\pm \boldsymbol{x}_1^{j_1} \cdot \ldots \cdot \boldsymbol{x}_n^{j_n}$ that occurs in a formal term of the preceding

definition. Hence one can assign to any formal term $t = v \cdot P$ *(that belongs to some* M^g*) a specific value* $W(t, \underline{x}) := W(v) \cdot W(P, \underline{x})$*. For a set* S *of formal terms we define* $W(S, \underline{x}) := \sum_{t \in S} W(t, \underline{x})$*. For the case* $S = \phi$ *we set* $W(\phi, \underline{x}) := 0$*.*

The value $W(t, \underline{x})$ *of a formal term* t *reflects the value of this term for network input* $\underline{x}$ *under the fixed parameter assignment in* $\mathcal{N}$*. These values* $W(t, \underline{x})$ *are needed for the definition of the systems* $L^g(\underline{x})$ *and* $L(\mathcal{N}, \underline{x})$ *of linear inequalities that describe the computation of* $\mathcal{N}$*.*

If g *has depth* 1*, then we define*

$$S^g(\underline{x}) := \{v_0^g\} \cup \{v_i^g \cdot \boldsymbol{x}_i \mid i = 1, \ldots, n\}.$$

Assume that g *is a gate on level* $l + 1$ *with edges from gates* $g_1, \ldots, g_m$ *on level* l *leading into* g*. Then we set*

$$S^g(\underline{x}) := \{v_0^g\} \cup \bigcup_{j=1}^{m} T^{g_j}(\underline{x}).$$

We define $L^g(\underline{x})$ *and* $T^g(\underline{x})$ *as follows for any gate* g *in* $\mathcal{N}$*. If* $W(S^g(\underline{x}), \underline{x}) < t_1^g$*, then* $L^g(\underline{x})$ *contains the inequality* $[\sum S^g(\underline{x}) + 1]_{\underline{\boldsymbol{x}}}[\underline{x}] \leq v_{\mathrm{I}}^g$*. If* $W(S^g(\underline{x}), \underline{x}) \geq t_2^g$*, then* $L^g(\underline{x})$ *contains the inequality* $[\sum S^g(\underline{x})]_{\underline{\boldsymbol{x}}}[\underline{x}] \geq v_{\mathrm{II}}^g$*. In either case we set* $T^g(\underline{x}) := \phi$*.*

[We use here and in the following the notation $[H]_{\underline{\boldsymbol{x}}}[\underline{x}]$ for any sum H of formal terms to indicate that each variable $\boldsymbol{x}_i$ in H is replaced by the value of the ith coordinate x_i of the concrete input $\underline{x} \in \mathbf{R}^n$. Note that the only variables that are left in $[H]_{\underline{\boldsymbol{x}}}[\underline{x}]$ are the variables of the form v_{const}^g, v_i^g, or $v_{w_1,\ldots,w_k}^g$. This substitution is necessary to make sure that the only variables that occur in the resulting system $L(\mathcal{N}, S)$ of linear inequalities are of this type or are variables of the form $v_{\mathrm{I}}^g, v_{\mathrm{II}}^g$.]

If $t_1^g \leq W(S^g(\underline{x}), \underline{x}) < t_2^g$*, then* $L^g(\underline{x})$ *contains the inequalities* $v_{\mathrm{I}}^g \leq [\sum S^g(\underline{x})]_{\underline{\boldsymbol{x}}}[x]$ *and* $[\sum S^g(\underline{x}) + 1]_{\underline{\boldsymbol{x}}}[\underline{x}] \leq v_{\mathrm{II}}^g$*. If* γ^g *gives on its nontrivial piece a constant* a^g *as output, we set in this case* $T^g(\underline{x}) := \{v_{\mathrm{const}}^g\}$*. If* γ^g *computes on its nontrivial piece a power* $y \mapsto y^k$*, we set*

$$T^g(\underline{x}) := \left\{ v_{w_1,\ldots,w_k}^g \cdot \prod_{i=1}^{k} P_i \mid \langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in (\{v_0^g\} \cup \{v_1^g \cdot \boldsymbol{x}_1, \ldots, v_n^g \cdot \boldsymbol{x}_n\})^k \right\}$$

if g *has depth* 1*, and in the general case*

$$T^g(\underline{x}) := \left\{ v_{w_1,\ldots,w_k}^g \cdot \prod_{i=1}^{k} P_i \mid \langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in \left(\{v_0^g\} \cup \bigcup_{j=1}^{m} T^{g_j}(\underline{x}) \right)^k \right\}.$$

Finally, if g *is the output gate* g_{out} *of* $\mathcal{N}$ *and* $W(T^{g_{\mathrm{out}}}(\underline{x}), \underline{x}) < W(v^{g_{\mathrm{out}}})$*, we add to* $L^{g_{\mathrm{out}}}(\underline{x})$ *also the inequality* $[\sum T^{g_{\mathrm{out}}}(\underline{x}) + 1]_{\underline{\boldsymbol{x}}}[\underline{x}] \leq v^{g_{\mathrm{out}}}$*. If* $W(T^{g_{\mathrm{out}}}(\underline{x}), \underline{x}) \geq W(v^{g_{\mathrm{out}}})$*, we add to* $L^{g_{\mathrm{out}}}(\underline{x})$ *also the inequality* $[\sum T^{g_{\mathrm{out}}}(\underline{x})]_{\underline{\boldsymbol{x}}}[\underline{x}] \geq v^{g_{\mathrm{out}}}$*.*

We define

$$L(\mathcal{N}, \underline{x}) := \bigcup \{L^g(\underline{x}) \mid g \text{ is a gate of } \mathcal{N}\},$$

and for $S \subseteq \boldsymbol{R}^n$

$$L(\mathcal{N}, S) := \bigcup \{L(\mathcal{N}, \underline{x}) \mid \underline{x} \in S\}.$$

The following lemma verifies that for any $\underline{x} \in \boldsymbol{R}^n$ the system $L(\mathcal{N}, \underline{x})$ of inequalities provides a truthful description of the computation of $\mathcal{N}$ for input $\underline{x}$.

LEMMA 3.5. *Assume that $\mathcal{N}$ is a network architecture in normal form with n input variables and some arbitrary assignment to its parameters and that $\underline{x} \in \boldsymbol{R}^n$ is an arbitrary concrete input.*

Then we have for any gate g in $\mathcal{N}$ that $W(S^g(\underline{x}), \underline{x})$ is the input and $W(T^g(\underline{x}), \underline{x})$ is the output of gate g (multiplied with the weight on the next edge) in the computation of $\mathcal{N}$ for input $\underline{x}$. Furthermore,

$$W(S^g(\underline{x}), \underline{x}) < t_1^g \Leftrightarrow \text{“}[S^g(\underline{x}) + 1]_{\underline{\boldsymbol{x}}}[\underline{x}] \leq v_{\mathrm{I}}^g\text{”} \in L(\mathcal{N}, \underline{x})$$
$$W(S^g(\underline{x}), \underline{x}) < t_2^g \Leftrightarrow \text{“}[S^g(\underline{x}) + 1]_{\underline{\boldsymbol{x}}}[\underline{x}] \leq v_{\mathrm{II}}^g\text{”} \in L(\mathcal{N}, \underline{x})$$
$$W(S^g(\underline{x}), \underline{x}) \geq t_1^g \Leftrightarrow \text{“}[S^g(\underline{x})]_{\underline{\boldsymbol{x}}}[\underline{x}] \quad\; \geq v_{\mathrm{I}}^g\text{”} \in L(\mathcal{N}, \underline{x})$$
$$W(S^g(\underline{x}), \underline{x}) \geq t_2^g \Leftrightarrow \text{“}[S^g(\underline{x})]_{\underline{\boldsymbol{x}}}[\underline{x}] \quad\; \geq v_{\mathrm{II}}^g\text{”} \in L(\mathcal{N}, \underline{x}).$$

Proof. The claim about $L(\mathcal{N}, \underline{x})$ follows immediately from the definition of $L(\mathcal{N}, \underline{x})$ in Definition 3.4.

One shows by induction on g that for any network input $\underline{x}$ the input of g in $\mathcal{N}$ is equal to $W(S^g(\underline{x}), \underline{x})$, and the output of g in $\mathcal{N}$ (after multiplication with the weight on the next edge) is equal to $W(T^g(\underline{x}), \underline{x})$.

If g is of depth 1 then we have by the definition of $S^g(\underline{x})$ in Definition 3.4 and by the definition of the values $W(t, \underline{x})$ for terms t in Definition 3.3 that $W(S^g(\underline{x}), \underline{x}) = \alpha_0^g + \sum_{i=1}^n \alpha_i^g \cdot x_i$, where $\alpha_1, \ldots, \alpha_n^g$ are the weights and α_0^g is the bias of gate g in $\mathcal{N}$ under the given parameter assignment in $\mathcal{N}$. Hence $W(S^g(\underline{x}), \underline{x})$ is equal to the input of g in $\mathcal{N}$ for network input $\underline{x}$. Furthermore if $\alpha_0^g + \sum_{i=1}^n \alpha_i^g \cdot x_i < t_1^g$ or $\alpha_0^g + \sum_{i=1}^n \alpha_i^g \cdot x_i \geq t_2^g$ then $T^g(\underline{x}) = \phi$; hence $W(T^g(\underline{x}), \underline{x}) = 0$. If $t_1^g \leq \alpha_0^g + \sum_{i=1}^n \alpha_i^g \cdot x_i < t_2^g$ then $W(T^g(\underline{x}), \underline{x}) = \alpha^g$ if γ^g outputs the constant 1 on its nontrivial piece (where α^g is the weight on the edge out of g). If γ^g computes $y \mapsto y^k$ on its nontrivial piece, then $W(T^g(\underline{x}), \underline{x}) = \alpha^g \cdot \sum(\{\alpha_0^g\} \cup \{\alpha_i^g \cdot x_i \mid i = 1, \ldots, n\})^k$. In either case $W(T^g(\underline{x}), \underline{x})$ is equal to the output of g in $\mathcal{N}$ (multiplied with α^g) for network input $\underline{x}$.

If g is of depth $l+1$ with immediate predecessors $g_1, \ldots, g_m$ then $W(S^g(\underline{x}), \underline{x}) = \alpha_0^g + \sum_{j=1}^m W(T^{g_j}(\underline{x}), \underline{x})$. By induction hypothesis this value is equal to the input of gate g in $\mathcal{N}$ for network input $\underline{x}$. In the most interesting case, where gate g applies the polynomial piece $y \mapsto y^k$ to this input, its output (multiplied with α^g) is equal to

$$\alpha^g \cdot \left(\alpha_0^g + \sum_{j=1}^m W(T^{g_j}(\underline{x}), \underline{x}) \right)^k$$

$$= \alpha^g \cdot \sum \left\{ \sum_{i=1}^k W(w_i \cdot P_i, \underline{x}) | \langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in \left(\{v_0^g\} \cup \bigcup_{j=1}^m T^{g_j}(\underline{x}) \right)^k \right\}$$

$$= \sum \left\{ W(v^g_{w_1, \ldots, w_k}) \cdot \prod_{i=1}^k W(P_i, \underline{x}) | \langle w_1 \cdot P_1, \ldots, w_k \cdot P_k \rangle \in \left(\{v_0^g\} \cup \bigcup_{j=1}^m T^{g_j}(\underline{x}) \right)^k \right\}$$

$$= W(T^g(\underline{x}), \underline{x}). \qquad \square$$

DEFINITION 3.6. *Assume that $\mathcal{N}$ is a neural act in normal form with n inputs. Furthermore, assume that $S \subseteq \boldsymbol{R}^n$ and $\mathcal{A} : V^{\mathcal{N}} \to \boldsymbol{R}$ is an arbitrary solution of the system $L(\mathcal{N}, S)$ of inequalities with the variable set $V^{\mathcal{N}} := \bigcup\{V^g \mid g \text{ is a gate in } \mathcal{N}\}$.*

We define by induction on the depth of gate g in $\mathcal{N}$ for each term $t \in M^g$ a first-order network architecture $\mathcal{M}^{\mathcal{A}}_{g,t}$. Together the network architectures $(\mathcal{M}^{\mathcal{A}}_{g,t})_{t\in M^g}$ mimic the initial segment of $\mathcal{N}$ between the input and gate g. The first-order network architecture $\mathcal{M}^{\mathcal{A}}_{g,t}$ consists of gates with activation functions from the class $\{$Heaviside, $y \mapsto y$, $y \mapsto y^2\}$. For any circuit input $\underline{x} \in S$ the output of the first-order network architecture $\mathcal{M}^{\mathcal{A}}_{g,t}$ will be 1 if $t \in T^g(\underline{x})$; otherwise it will be 0.

One associates with each network architecture $\mathcal{M}^{\mathcal{A}}_{g,t}$ for $t \in M^g$ of the form $t \equiv v \cdot P$ another network architecture $\tilde{\mathcal{M}}^{\mathcal{A}}_{g,t}$ that outputs for any network input $\underline{x} \in S$ the real number

$$\mathcal{A}(t, \underline{x}) := \begin{cases} \mathcal{A}(v) \cdot W(P, \underline{x}) & \text{if } \mathcal{M}^{\mathcal{A}}_{g,t}(\underline{x}) = 1, \\ 0 & \text{if } \mathcal{M}^{\mathcal{A}}_{g,t}(\underline{x}) = 0. \end{cases}$$

The extension from $\mathcal{M}^{\mathcal{A}}_{g,t}$ to $\tilde{\mathcal{M}}^{\mathcal{A}}_{g,t}$ is done in a canonical manner with the help of subcircuits that simulate product gates via the equality $y \cdot z = \frac{1}{2}((y+z)^2 - y^2 - z^2)$. Obviously, $\tilde{\mathcal{M}}^{\mathcal{A}}_{g,t}$ just has to compute the product of $\mathcal{A}(v)$, $W(P, \underline{x})$ and of the output of $\mathcal{M}^{\mathcal{A}}_{g,t}$ for network input $\underline{x}$.

The definition of a value $\mathcal{A}(t, \underline{x})$ for each term t and each $\underline{x} \in S$ is extended in a canonical way to arbitrary sets M of terms

$$\mathcal{A}(M, \underline{x}) := \sum_{t\in M} \mathcal{A}(t, \underline{x}), \qquad \mathcal{A}(\phi, \underline{x}) := 0.$$

We consider first the case where g has depth 1. Let H^g_1 be a linear threshold gate that checks whether $\mathcal{A}(v^g_{\mathrm{I}}) \le \mathcal{A}(S^g(\underline{x}), \underline{x})$, and let H^g_2 be a linear threshold gate that checks whether $\mathcal{A}(S^g(\underline{x}), \underline{x}) + 1 \le \mathcal{A}(v^g_{\mathrm{II}})$. For each term $t \in M^g$ we define $\mathcal{M}^{\mathcal{A}}_{g,t}$ to be the AND of H^g_1 and H^g_2.

Assume then that g is a gate on level $l+1$ with edges from the gates $g_1, \ldots, g_m$ on level l leading into g. According to Definition 3.4 we have in this case $S^g(\underline{x}) = \{v^g_0\} \cup \bigcup_{j=1}^m T^{g_j}(\underline{x})$ for every $\underline{x} \in S$. By induction hypothesis we have already defined network architectures $\mathcal{M}^{\mathcal{A}}_{g_j,t}$, and hence also network architectures $\tilde{\mathcal{M}}^{\mathcal{A}}_{g_j,t}$ for all $t \in M^{g_j}$, $j = 1, \ldots, m$. For each term $t \in M^g$ the network architecture $\mathcal{M}^{\mathcal{A}}_{g,t}$ employs two linear threshold gates H^g_1 and H^g_2, which receive their inputs from the network architectures $\tilde{\mathcal{M}}^{\mathcal{A}}_{g_j,t}$ for $t \in M^{g_j}$, $j = 1, \ldots, m$. The linear threshold gate H^g_1 has the task to check for any $\underline{x} \in S$ whether $\mathcal{A}(v^g_{\mathrm{I}}) \le \mathcal{A}(S^g(\underline{x}), \underline{x})$. Obviously it can easily accomplish this task provided that for input $\underline{x}$ the network architectures $\tilde{\mathcal{M}}^{\mathcal{A}}_{g_j,t}$ for $t \in M^{g_j} (j = 1, \ldots, m)$ give as output the value $\mathcal{A}(t, \underline{x})$. Analogously the linear threshold gate H^g_2 has the task to check whether $\mathcal{A}(S^g(\underline{x}), \underline{x}) + 1 \le \mathcal{A}(v^g_{\mathrm{II}})$.

If γ^g outputs the constant 1 on its nontrivial piece, $\mathcal{M}^{\mathcal{A}}_{g,v^g_{\mathrm{const}}}$ is defined as the AND of H^g_1 and H^g_2.

If γ^g computes $y \mapsto y^k$ on its nontrivial piece, then each $t \in M^g$ is of the form $v^g_{w_1,\ldots,w_k} \cdot \prod_{i=1}^k P_i$ for some ktuple $\langle w_1 \cdot P_1, \ldots, w_k \cdot P_k\rangle \in (\{v^g_0\} \cup \bigcup_{j=1}^m M^{g_j})^k$. In this case $\mathcal{M}^{\mathcal{A}}_{g,t}$ is defined as the AND of H^g_1, H^g_2 and of the outputs of the network architectures $\mathcal{M}^{\mathcal{A}}_{g_j,w_i\cdot P_i}$ for all $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, m\}$ with $w_i \cdot P_i \in M^{g_j}$.

[A word of caution: Although the variable $v^g_{w_1,\ldots,w_k}$ is supposed to play the role of the product of $w_1,\ldots,w_k$ and α^g (where α^g is the weight on the edge out of g), the assignment $\mathcal{A}$ will in general *not* satisfy $\mathcal{A}(v^g_{w_1,\ldots,w_k}) = \mathcal{A}(\alpha^g)\cdot\prod_{i=1}^k \mathcal{A}(w_i)$.]

Finally we define the network architecture $\mathcal{M}^{\mathcal{A}}$ by using as components the network architectures $\mathcal{M}^{\mathcal{A}}_{g_{\text{out}},t}$ for all $t \in M^{g_{\text{out}}}$. The output of $\mathcal{M}^{\mathcal{A}}$ is given by a linear threshold gate H that checks whether $\sum_{t:\mathcal{M}^{\mathcal{A}}_{g_{\text{out}},t}(\underline{x})=1} \mathcal{A}(t,\underline{x}) \geq \mathcal{A}(v^{g_{\text{out}}})$.

LEMMA 3.7. *Assume that $S \subseteq \boldsymbol{R}^n$ and $\mathcal{A}$ is an arbitrary solution of $L(\mathcal{N},S)$. Then the following holds for any gate g in $\mathcal{N}$, for any term $t \in M^g$, and any input $\underline{x} \in S$:*

(a) *For network input $\underline{x}$ the gate H^g_1 in $\mathcal{M}^{\mathcal{A}}_{g,t}$ outputs 1 if and only if $t^g_1 \leq W(S^g(\underline{x}),\underline{x})$. Similarly the output of the gate H^g_2 in $\mathcal{M}^{\mathcal{A}}_{g,t}$ is 1 if and only if $W(S^g(\underline{x}),\underline{x}) + 1 \leq t^g_2$.*

(b) *$t \in T^g(\underline{x}) \Leftrightarrow$ ($\mathcal{M}^{\mathcal{A}}_{g,t}$ outputs 1 for network input $\underline{x}$).*

(c) *$\tilde{\mathcal{M}}^{\mathcal{A}}_{g,t}$ outputs $\mathcal{A}(t,\underline{x})$ for network input $\underline{x}$.*

Proof. The proof proceeds by induction on the depth of gate g. The claim is obvious from the definition if g is of depth 1. If g is of depth $l+1>1$ we exploit the induction hypothesis for the network architectures $\mathcal{M}^{\mathcal{A}}_{g_j,t}$ and $\tilde{\mathcal{M}}^{\mathcal{A}}_{g_j,t}$ with $t \in M^{g_j}$ (for the immediate predecessors g_j of gate g). Hence we may assume that gate H^g_1 in $\mathcal{M}^{\mathcal{A}}_{g,t}$ outputs 1 if and only if $\mathcal{A}(v^g_{\text{I}}) \leq \mathcal{A}(S^g(\underline{x}),\underline{x})$. Since $\mathcal{A}$ is a solution of $L(\mathcal{N},S)$, the latter inequality holds if and only if $L(\mathcal{N},S)$ contains the inequality $v^g_{\text{I}} \leq [S^g(\underline{x})]_{\underline{x}}[\underline{x}]$. By Lemma 3.5 this holds if and only if $t^g_1 \leq W(S^g(\underline{x}),\underline{x})$. The claim for H^g_2 is verified analogously.

The least trivial case for part (b) of the claim is the case where γ^g computes $y \mapsto y^k$ on its nontrivial piece. Then each $t \in M^g$ is of the form $v_{w_1,\ldots,w_k}\cdot\prod_{i=1}^k P_i$ for some $k$tuple $\langle w_1\cdot P_1,\ldots,w_k\cdot P_k\rangle \in (\{v^g_0\}\cup\bigcup_{j=1}^m M^{g_j})^k$. By definition of $T^g(\underline{x})$ we have $t \in T^g(\underline{x})$ if and only if $t^g_1 \leq W(S^g(\underline{x}),\underline{x}) < t^g_2$ and $w_i\cdot P_i \in T^{g_j}(t)$ for all $i \in \{1,\ldots,k\}$ and $j \in \{1,\ldots,m\}$ with $w_i\cdot P_i \in M^{g_j}$. By construction of $\mathcal{M}^{\mathcal{A}}_{g,t}$ and by the induction hypothesis we have that $\mathcal{M}^{\mathcal{A}}_{g,t}$ outputs 1 for network input $\underline{x}$ if and only if all of the preceding conditions are satisfied.

Part (c) of the claim for gate g follows immediately from part (b) and the definition of $\tilde{\mathcal{M}}^{\mathcal{A}}_{g,t}$. □

LEMMA 3.8. *Assume that $\mathcal{N}$ is a network architecture in normal form with n input variables, $S \subseteq \boldsymbol{R}^n$ is an arbitrary set of inputs, and $\mathcal{A}$ is an arbitrary solution of $L(\mathcal{N},S)$. Then $\mathcal{N}$ and $\mathcal{M}^{\mathcal{A}}$ compute the same function from S into $\{0,1\}$.*

Proof. This is an immediate consequence of Lemmas 3.5 and 3.7. By the definition of $\mathcal{M}^{\mathcal{A}}$ the output of $\mathcal{M}^{\mathcal{A}}$ for any network input $\underline{x} \in S$ is 1 if and only if $\sum_{t:\mathcal{M}^{\mathcal{A}}_{g_{\text{out}},t}(\underline{x})=1} \mathcal{A}(t,\underline{x}) \geq \mathcal{A}(v^{g_{\text{out}}})$. By Lemma 3.7 we have that $\mathcal{M}^{\mathcal{A}}_{g_{\text{out}},t}(\underline{x}) = 1 \Leftrightarrow t \in T^{g_{\text{out}}}(\underline{x})$. Hence, since $\mathcal{A}$ is a solution of $L(\mathcal{N},S)$, the preceding inequality holds if and only if $L(\mathcal{N},S)$ contains the inequality $[\sum T^{g_{\text{out}}}(\underline{x})]_{\underline{\boldsymbol{x}}}[\underline{x}] \geq v^{g_{\text{out}}}$. By definition of $L(\mathcal{N},S)$ the latter holds if and only if $W(T^{g_{\text{out}}}(\underline{x}),\underline{x}) \geq W(v^{g_{\text{out}}})$. By Lemma 3.5 the value $W(T^{g_{\text{out}}}(\underline{x}),\underline{x})$ is the output of g_{out} in $\mathcal{N}$ for network input $\underline{x}$. Hence $W(T^{g_{\text{out}}}(\underline{x}),\underline{x}) \geq W(v^{g_{\text{out}}})$ holds if and only if $\mathcal{N}$ outputs 1 for network input $\underline{x}$. □

We are now in a position where we can complete the proof of Theorem 3.1. Assume that a given array $(\mathcal{N}_n)_{n\in\mathbf{N}}$ of neural nets satisfies the assumption of Theorem 3.1 and that $(\underline{\alpha}_n)_{n\in\mathbf{N}}$ is an arbitrary array of real-valued assignments $\underline{\alpha}_n$ to the variable parameters in $\mathcal{N}_n$. One can transform the given neural nets $(\mathcal{N}_n^{\underline{\alpha}_n})_{n\in\mathbf{N}}$ into an array $(\hat{\mathcal{N}}_n^{\underline{\beta}_n})_{n\in\mathbf{N}}$ of neural nets in normal form (with properties as specified above) such that

$\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ computes the same Boolean function as $\mathcal{N}_n^{\underline{\alpha}_n}$. We then apply the machinery from the definition and Lemmas 3.5 to 3.8 to each neural net $\mathcal{N} := \hat{\mathcal{N}}_n^{\underline{\beta}_n}$ with $S := \{0,1\}^n$. By construction of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ the resulting system $L(\mathcal{N}, \{0,1\}^n)$ of inequalities has some solution over $\mathbf{R}$. We exploit here in particular that $\underline{\beta}_n$ was chosen so that all relevant strict inequalities "$s_1 < s_2$" in computations of $\hat{\mathcal{N}}_n^{\underline{\beta}_n}$ on inputs $\underline{x} \in \{0,1\}^n$ were strengthened to "$s_1 + 1 \leq s_2$." Since $|\bigcup\{M^g \mid g \text{ gate in } \hat{\mathcal{N}}_n^{\underline{\beta}_n}\}| = O(n^{O(1)})$, it follows that the number of gates in $\mathcal{M}^{\mathcal{A}}$ is bounded by $O(n^{O(1)})$.

The number of variables in $L(\mathcal{N}, \{0,1\}^n)$ is polynomial in n, and it only contains small constants. Hence by Lemma 2.4 there is a solution $\mathcal{A}$ of $L(\mathcal{N}, \{0,1\}^n)$ that consists of rationals of the form $\frac{s}{t}$ (with a common integer t) such that s and t are integers of size $2^{O(n^{O(1)})}$. By Lemma 3.8 the constructed network architecture $\mathcal{M}^{\mathcal{A}}$ computes the same Boolean function as $\mathcal{N}$. Furthermore all constants and parameters in $\mathcal{M}^{\mathcal{A}}$ are quotients of integers with polynomially in n many bits. Thus (see [SBKH], [SR]) one can carry out all arithmetical operations in $\mathcal{M}^{\mathcal{A}}$ for inputs from $\{0,1\}^n$ by polynomial-sized digital subcircuits of constant depth with linear threshold gates (or, equivalently, with MAJORITY-gates, see [CSV]). In the resulting circuit all parameters from $\mathcal{A}$ are replaced by corresponding sequences of bits. Hence one gets in this way neural nets $\tilde{\mathcal{N}}_n$ which satisfy the claim of Theorem 3.1. □

Remark 3.9. Sontag [S3] suggested using the quasilinearization that is achieved in the proof of Theorem 3.1 in order to also get upper bounds for the VC dimension over $\mathbf{R}^n$ by counting the number of components into which the weight space is partitioned by the hyperplanes that are defined by some arbitrary finite set $S \subseteq \mathbf{R}^n$ of inputs.

By letting $\underline{\alpha}_n$ vary and keeping the neural net $\mathcal{N}_n$ and the input $\underline{x} \in S$ fixed one gets up to $2^{O(n^{O(1)})}$ different systems $L(\hat{\mathcal{N}}_n^{\underline{\beta}_n}, \underline{x})$ in the proof of Theorem 3.1. Hence the total number l_n of linear inequalities that arise in this way for different $\underline{x} \in S$ and different parameters $\underline{\alpha}_n$ is bounded by $|S| \cdot 2^{O(n^{O(1)})}$. Furthermore, the total number w_n of variables that occur in these l_n inequalities is bounded by $O(n^{O(1)})$. Therefore the hyperplanes that are associated with these l_n inequalities partition the range $\mathbf{R}^{w_n}$ of the variables into at most $\sum_{k=0}^{w_n} \sum_{i=0}^{k} \binom{w_n - i}{k-i} \binom{l_n}{w_n - i} = |S|^{O(n^{O(1)})}$ connected components (Theorem 1.3 in [E]). Each $\mathcal{A} \in \mathbf{R}^{w_n}$ gives rise to at most $2^{O(n^{O(1)})}$ different network architectures $\mathcal{M}^{\mathcal{A}}$ when $\mathcal{N}_n$ and S are kept fixed, but the parameters $\underline{\alpha}_n$ vary. Thus each $\mathcal{A} \in \mathbf{R}^{w_n}$ can be used to compute at most $2^{O(n^{O(1)})}$ different functions $S \to \{0,1\}$ on the resulting circuits. Furthermore, if $\mathcal{A}$ and $\tilde{\mathcal{A}}$ belong to the *same* connected component of the partition of $\mathbf{R}^{w_n}$ then for all $\underline{\alpha}_n$ the network architectures $\mathcal{M}^{\mathcal{A}}$ and $\mathcal{M}^{\tilde{\mathcal{A}}}$ compute the same function $S \to \{0,1\}$. Hence if S is shattered by $\mathcal{N}_n$ (i.e., *any* function $S \to \{0,1\}$ can be computed by $\mathcal{N}_n^{\underline{\alpha}_n}$ for suitable parameters $\underline{\alpha}_n$) then $2^{|S|} \leq |S|^{O(n^{O(1)})} \cdot 2^{O(n^{O(1)})}$; hence $|S| = O(n^{O(1)})$. This implies that VC dimension $(\mathcal{N}_n, \mathbf{R}^n) = O(n^{O(1)})$.

One can apply in a similar fashion the linearization that is achieved in the proof of Theorem 2.1. Consider a neural net $\mathcal{N}$ over a graph $\langle V, E\rangle$ as in Theorem 2.1, but allow that each activation function γ^g consists of $\leq p$ linear pieces with arbitrary fixed *real* parameters. Then one can show that VC dimension $(\mathcal{N}, \mathbf{R}^n) = O(w^2 \log p)$, where $w := |V| + |E| + 1$ is the number of variable parameters in $\mathcal{N}$. It is sufficient to observe that for different $\underline{x} \in S$ and different initial assignments $\underline{\alpha}$ altogether at most $|S| \cdot 2^{O(w \log p)}$ linear inequalities arise in the description of the computations of the associated nets $\hat{\mathcal{N}}[\underline{c}]^{\underline{\beta}}$ for input $\underline{x}$. The associated hyperplanes partition the "weight space" $\mathbf{R}^w$ for the variable parameters $\underline{\beta}$, $\underline{c}$ into $\leq |S|^{O(w)} \cdot 2^{O(w^2 \log p)}$ connected

components. The vectors from each connected component can be used to compute at most $2^{O(w)}$ different functions $S \to \{0,1\}$ (note that in general more than one function $S \to \{0,1\}$ can be computed because of different weights from $\{-1,0,1\}$ between computation nodes). Hence $2^{|S|} \leq |S|^{O(w)} \cdot 2^{O(w^2 \log p)} \cdot 2^{O(w)}$ if S is shattered by $\mathcal{N}$; thus $|S| = O(w^2 \log p)$.

Subsequent to this observation from [M92] and [M93a], our polynomial upper bound for the VC dimension of analog neural nets of constant depth has been extended to neural nets of unbounded depth via an application of a well-known theorem of Milnor [GJ]. In [M93c] this result has been further generalized to yield a polynomial upper bound for the *pseudodimension* (see [H]) of analog neural nets of arbitrary depth which takes over the role of the VC dimension in the case of learning on analog neural nets with real-valued outputs.

4. PAC learning on analog neural nets. We now turn to the analysis of learning on analog neural nets in Valiant's model [V] for probably approximately correct learning (PAC learning). More precisely, we consider the common extension of this model to real-valued domains due to [BEHW]. Unfortunately, most results about PAC learning on neural nets are negative (see [BR], [KV]). This could mean either that learning on neural nets is impossible or that the common theoretical analysis of learning on neural nets is not quite adequate.

We want to point to one somewhat problematic aspect of the traditional asymptotic analysis of PAC learning on neural nets. In analogy to the standard asymptotic analysis of the run time of algorithms in terms of the number n of input bits one usually formalizes PAC learning on neural nets in exactly the same fashion. However in contrast to the common situation for computer algorithms (which typically receive their input in digital form as a long sequence of n bits) for many important applications of neural nets the input is given in analog form as a vector of a *small* number n of analog real-valued parameters. These relatively few input parameters may consist for example of sensory data, or they may be the relevant components of a longer feature vector (which were extracted by some other mechanism). If one analyzes PAC learning on neural nets in this fashion, the relevant asymptotic problem becomes a different one: can a given analog neural net with a fixed number n of analog inputs approximate the target concept arbitrarily close after it has been shown sufficiently many training examples?

We show that for those types of neural nets which were considered in the preceding sections the previously discussed PAC learning problem has in fact a positive solution.

THEOREM 4.1. *Let $\mathcal{N}$ be an arbitrary network architecture of order 1 as in Theorem 2.1, where the fixed parameters of the piecewise-linear activation functions may now be arbitrary reals. Let $\mathcal{C}_{\mathcal{N}} := \{C \subseteq \boldsymbol{R}^n | \exists \underline{\alpha} \in \boldsymbol{R}^w \, \forall \underline{x} \in \boldsymbol{R}^n (\chi_C(\underline{x}) = \mathcal{N}^{\underline{\alpha}}(\underline{x}))\}$ be the associated concept class, where χ_C is the characteristic function of a concept C. Then $\mathcal{C}_{\mathcal{N}}$ is properly PAC learnable.*

This means that one can design for the given network architecture $\mathcal{N}$ a learning algorithm $LA^{\mathcal{N}}$ such that for any distribution Q over $\boldsymbol{R}^n$, any target concept $C_T \in \mathcal{C}_{\mathcal{N}}$, and any given ε, $\delta \in \boldsymbol{R}^+$ the learning algorithm $LA^{\mathcal{N}}$ with inputs ε and δ carries out in $O((\frac{1}{\varepsilon})^{O(1)}, (\frac{1}{\delta})^{O(1)})$ computations steps (with respect to the uniform cost criterion on a RAM) the following task: it computes a suitable number m and draws some sequence S of m examples for C_T according to distribution Q. Then it computes from S an assignment $\underline{\alpha}_S \in \boldsymbol{R}^w$ for the programmable parameters of $\mathcal{N}$ such that $Q[\{\underline{x} \in \boldsymbol{R}^n | \chi_{C_T}(\underline{x}) \neq \mathcal{N}^{\underline{\alpha}_S}(\underline{x})\}] \leq \varepsilon$ with probability $\geq 1 - \delta$.

Proof. We have VC dimension $(\mathcal{C}_{\mathcal{N}}) < \infty$ by Remark 3.9. Hence according

to [BEHW], it suffices to show that for any given set S of m examples for C_T one can compute from S within a number of computation steps that is polynomial in $m, \frac{1}{\varepsilon}, \frac{1}{\delta}$ an assignment $\underline{\alpha}_S \in \mathbf{R}^w$ to the programmable parameters of $\mathcal{N}$ such that $\forall \underline{x} \in S(\chi_{C_T}(\underline{x}) = \mathcal{N}^{\underline{\alpha}_S}(\underline{x}))$. The construction in the proof of Theorem 2.1 implies that it is sufficient if one computes instead with polynomially in m, $\frac{1}{\varepsilon}$, $\frac{1}{\delta}$ computation steps an assignment $\underline{\beta}_S$, $\underline{c}_S$ of parameters for the associated neural net $\hat{\mathcal{N}}$ such that $\forall \underline{x} \in S(\chi_{C_T}(\underline{x}) = \hat{\mathcal{N}}[\underline{c}_S]^{\underline{\beta}_S}(\underline{x}))$. The latter task is easier because the role of the parameters $\underline{\beta}$, $\underline{c}$ in a computation of $\hat{\mathcal{N}}$ for a specific input $\underline{x}$ can be described by *linear* inequalities (provided one knows which linear piece is used at each gate).

Nevertheless, the following technical problem remains. Although we know which *output* $\hat{\mathcal{N}}[\underline{c}_S]^{\underline{\beta}_S}$ should give for an input $\underline{x} \in S$, we do not know *in which way* this output should be produced by $\hat{\mathcal{N}}[\underline{c}_S]^{\underline{\beta}_S}$. More specifically, we don't know which particular piece of each piecewise-linear activation function γ^g of $\hat{\mathcal{N}}$ will be used for this computation. However, this detailed information would be needed for each $\underline{x} \in S$ and for all gates g of $\hat{\mathcal{N}}$ in order to describe the resulting constraints on the parameters $\underline{\beta}$, $\underline{c}$ by a system of linear inequalities.

However, one can generate a set of polynomially in m many systems of linear inequalities such that at least one of these systems provides for all $\underline{x} \in S$ satisfiable and sufficient constraints for $\underline{\beta}, \underline{c}$. By definition $\mathcal{C}_{\mathcal{N}}$ we know that there are parameters $\underline{\beta}_T$, $\underline{c}_T$ such that $\hat{\mathcal{N}}[\underline{c}_T]^{\underline{\beta}_T}$ computes χ_{C_T}. Consider any inequality $I(\underline{\beta}, \underline{c}, \underline{x}) \leq 0$ (with $I(\underline{\beta}, \underline{c}, \underline{x})$ linear in $\underline{\beta}, \underline{c}$ for fixed $\underline{x}$, *and* linear in $\underline{x}$ for fixed $\underline{\beta}, \underline{c}$) as they were introduced in the proof of Theorem 2.1 in order to describe the comparison with a threshold at some gate g of $\hat{\mathcal{N}}$. The hyperplane $\{\underline{x} \in \mathbf{R}^n \mid I(\underline{\beta}_T, \underline{c}_T, \underline{x}) = 0\}$ defines a partition of S into $\{\underline{x} \in S \mid I(\underline{\beta}_T, \underline{c}, \underline{x}) \leq 0\}$ and $\{\underline{x} \in S \mid I(\underline{\beta}_T, \underline{c}, \underline{x}) > 0\}$. Hence it suffices to produce (e.g., with the algorithm of [EOS]) in polynomially in m many computation steps all partitions of S that can be generated by as many hyperplanes as there are linear inequalities $I(\underline{\beta}, \underline{c}, \underline{x}) \leq 0$ in the proof of Theorem 2.1. One of these partitions will agree with the partition of S that is defined by the hyperplanes $\{\underline{x} \in \mathbf{R}^n \mid I(\underline{\beta}_T, \underline{c}_T, \underline{x}) = 0\}$ for the "correct values" $\underline{\beta}_T, \underline{c}_T$ of the parameters. Each of these partitions corresponds to a guess which linear pieces of the activation functions γ^g of $\hat{\mathcal{N}}$ are used for the different inputs $\underline{x} \in S$, and hence it defines a unique system of linear inequalities in $\underline{\beta}, \underline{c}$ (with the inputs $\underline{x} \in S$ as fixed coefficients). Furthermore, it is guaranteed that one of these guesses is correct for $\underline{\beta}_T, \underline{c}_T$.

For each of the resulting polynomially in m many systems of inequalities we apply the method of the proof of Lemma 2.4 (i.e., we reduce the solution of each system of inequalities to the solution of polynomially in m many systems of linear equalities), or we apply Megiddo's polynomial time algorithm for linear programming in a fixed dimension [Me] in order to find values $\underline{\beta}_s, \underline{c}_s$ for which $\hat{\mathcal{N}}[\underline{c}_s]^{\underline{\beta}_s}$ gives the desired outputs for all $\underline{x} \in S$. By construction, this algorithm will succeed for at least one of the selected system of inequalities. □

Remark 4.2. Assume $\mathcal{N}$ is some arbitrary network architecture of order 1 according to Definition 1.1 with arbitrary piecewise-linear activation functions, and $\mathcal{N}$ does not satisfy the condition that all computation nodes of $\mathcal{N}$ have fan-out ≤ 1. Then Theorem 4.1 does not show that $C_{\mathcal{N}}$ is *properly* PAC learnable. However, it implies that $C_{\mathcal{N}}$ is PAC learnable, with $\mathcal{C}_{\mathcal{N}'}$ for a somewhat larger network architecture $\mathcal{N}'$ of the same depth used as hypothesis class (see Remark 2.2 for the definition of $\mathcal{N}'$).

Note that this result may lead toward a theoretical explanation of an effect that has been observed in many experiments: one often achieves better learning results on artificial neural nets if one uses a neural net with somewhat more units than necessary

(i.e., necessary in order to compute the target concept on the neural net).

THEOREM 4.3. *Let $\mathcal{N}$ be an arbitrary network architecture with arbitrary piecewise-polynomial activation functions and arbitrary polynomial gate inputs $Q^g(y_1, \ldots, y_m)$. Then the associated concept class $\mathcal{C}_{\mathcal{N}}$ is PAC learnable with a hypothesis class of the form $\mathcal{C}_{\tilde{\mathcal{N}}}$ for a somewhat larger network architecture $\tilde{\mathcal{N}}$.*

Proof. One can reduce this problem to the case of network architectures with linear gate inputs as indicated at the beginning of the proof of Theorem 3.1. One uses as hypotheses, sets which are defined by a network architecture $\tilde{\mathcal{N}}$ of the same structure as the network architecture $\mathcal{M}^{\mathcal{A}}$ in the proof of Theorem 3.1. For this network architecture $\tilde{\mathcal{N}}$ one can express the constraints on the assignment $\mathcal{A}$ by *linear* inequalities. Remark 3.9 implies that VC dimension $(\tilde{\mathcal{N}}, \mathbf{R}^n) < \infty$.

One applies the method from the proof of Lemma 2.4 in a manner analogous to the proof of Theorem 4.1 or linear programming in a fixed dimension [Me] to polynomially in m many systems of linear inequalities. There is one small obstacle in generating the associated partitions of S since the corresponding inequalities are not linear in the circuit inputs $\underline{x}$. One overcomes this difficulty by going to an input space of higher dimension (where the variables represent monomials of bounded degree in the original variables). □

Remark 4.4. It is shown in [M93c] that the positive learning results of this section can be extended to analog neural nets with real-valued outputs. Furthermore it is shown in that paper that these learning results can be extended to Haussler's refinement [H] of Valiant's model [V], where no a priori assumptions about the target function are required and where arbitrary noise in the training examples is permitted.

Acknowledgments. We would like to thank Eduardo D. Sontag for drawing our attention to the problem of finding upper bounds for neural nets with π-gates and for his insightful comments. We thank Peter Auer, Franz Aurenhammer, Philip M. Long, and Gerhard Wöginger for various helpful discussions on this research.

REFERENCES

[A] Y. S. ABU-MOSTAFA, *The Vapnik-Chervonenkis dimension: Information versus complexity in learning*, Neural Comput., 1 (1989), pp. 312–317.

[B] P. L. BARTLETT, *Lower bounds on the Vapnik-Chervonenkis dimension of multi-layer threshold networks*, in Proc. 5th Annual ACM Conference on Computational Learning Theory, ACM, New York, 1993, pp. 144–150.

[BH] E. B. BAUM AND D. HAUSSLER, *What size net gives valid generalization?*, Neural Comput., 1 (1989), pp. 151–160.

[BR] A. BLUM AND R. L. RIVEST, *Training a 3-node neural network is NP-complete*, in Proc. 1988 Workshop on Computational Learning Theory, Morgan Kaufmann, San Mateo, CA, 1988, pp. 9–18.

[BEHW] A. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, AND M. K. WARMUTH, *Learnability and the Vapnik-Chervonenkis dimension*, J. Assoc. Comput. Mach., 36 (1989), pp. 929–965.

[CSV] A. K. CHANDRA, L. STOCKMEYER, AND U. VISHKIN, *Constant depth reducibility*, SIAM J. Comput., 13 (1984), pp. 423–439.

[DS] B. DASGUPTA AND G. SCHNITGER, *The power of approximating: A comparison of activation functions*, in Advances in Neural Information Processing Systems, Vol. 5, Morgan Kaufmann, San Mateo, 1993, pp. 615–622.

[DR] R. DURBIN AND D. E. RUMELHART, *Product units: A computationally powerful and biologically plausible extension to backpropagation networks*, Neural Computation, 1 (1989), pp. 133–142.

[E] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, 1987.

[EOS] H. Edelsbrunner, J. O'Rourke, and R. Seidel, *Constructing arrangements of lines and hyperplanes with applications*, SIAM J. Comput., 15 (1986), pp. 341–363.

[GJ] P. Goldberg and M. Jerrum, *Bounding the Vapnik-Chervonenkis dimension of concept classes parameterized by real numbers*, in Proc. 5th Annual ACM Conference on Computational Learning Theory, ACM, New York, 1993, pp. 361–369.

[GHR] M. Goldmann, J. Håstad, and A. Razborov, *Majority gates vs. general weighted threshold gates*, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 2–13.

[HMPST] A. Hajnal, W. Maass, P. Pudlák, M. Szegedy, and G. Turan, *Threshold circuits of bounded depth*, J. Comput. System Sci., 46 (1993), pp. 129–154.

[Has] J. Håstad, *On the size of weights for threshold gates*, SIAM J. Discrete Math., 7 (1994), pp. 484–492.

[H] D. Haussler, *Decision theoretic generalizations of the PAC model for neural nets and other learning applications*, Inform. and Comput., 100 (1992), pp. 78–150.

[Ho] J. J. Hopfield, *Neurons with graded response have collective computational properties like those of two-state neurons*, Proc. Nat. Acad. Sci. U.S.A., (1984), pp. 3088–3092.

[J] D. S. Johnson, *A catalog of complexity classes*, in Handbook of Theoretical Computer Science, A, J. van Leeuwen, ed., MIT Press, Cambridge, MA, 1990, pp. 67–161.

[KV] M. Kearns and L. Valiant, *Cryptographic limitations on learning boolean formulae and finite automata*, in Proc. 21st ACM Symposium on Theory of Computing, ACM, New York, 1989, pp. 433–444.

[K93] P. Koiran, *A weak version of the Blum, Shub, Smale model*, in Proc. 34th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 486–495.

[K94] P. Koiran, *Efficient learning of continuous neural networks*, Proc. 7th Annual ACM Conference on Computational Learning Theory, 1994, ACM, New York, 1994, pp. 348–355.

[L] R. P. Lippmann, *An introduction to computing with neural nets*, IEEE ASSP Magazine, 1987, pp. 4–22.

[M92] W. Maass, *Bounds for the computational power and learning complexity of analog neural nets*, IIG Report 349, Technische Universität Graz, Graz, Austria, 1992.

[M93a] W. Maass, *Bounds for the computational power and learning complexity of analog neural nets* (extended abstract), in Proc. 25th ACM Symposium on the Theory of Computing, 1993, pp. 335–344.

[M93b] W. Maass, *Neural nets with superlinear VC-dimension*, in Proc. International Conference on Artificial Neural Networks, Springer-Verlag, Berlin, 1994, pp. 581–584. Neural Comput., 6 (1994), pp. 877–884.

[M93c] W. Maass, *Agnostic PAC-learning of functions on analog neural nets*, Neural Comput., 7 (1995), pp. 902–926.

[MSS] W. Maass, G. Schnitger, and E. D. Sontag, *On the computational power of sigmoid versus boolean threshold circuits*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 767–776.

[MT] W. Maass and G. Turan, *How fast can a threshold gate learn?*, in Computational Learning Theory and Natural Learning Systems: Constraints and Prospects, G. Drastal, S. J. Hanson, and R. Rivest, eds., MIT Press, Cambridge, MA, 1994, pp. 381–414.

[MR] J. L. McClelland and D. E. Rumelhart, *Parallel Distributed Processing*, Vol. 2, MIT Press, Cambridge, MA, 1986.

[Me] N. Megiddo, *Linear programming in linear time when the dimension is fixed*, J. Assoc. Comput. Mach., 31 (1984), pp. 114–127.

[MP] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, expanded Ed., MIT Press, Cambridge, MA, 1988.

[MD] J. Moody and C. J. Darken, *Fast learning in networks of locally tuned processing units*, Neural Computation, 1 (1989), pp. 281–294.

[Mu] S. Muroga, *Threshold Logic and Its Applications*, John Wiley, New York, 1971.

[Ni] N. J. Nilsson, *Learning Machines*, McGraw–Hill, New York, 1971.

[PS] I. Parberry and G. Schnitger, *Parallel computation with threshold functions*, in Lecture Notes in Computer Science, Vol. 223, Springer-Verlag, Berlin, 1986, pp. 272–290.

[PG] T. Poggio and F. Girosi, *Networks for approximation and learning*, Proc. IEEE, 78 (1990), pp. 1481–1497.

[R] F. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1988.

[RM] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing*, Vol. 1, MIT Press, Cambridge, MA, 1986.

[Sch] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.

[SS] H. T. Spiegelmann and E. D. Sontag, *Neural Networks with Real Weights: Analog Computational Complexity*, Report SYCON-92-05, Rutgers Center for Systems and Control, Rutgers University, New Brunswick, NJ, 1992.

[SBKH] K. Y. Siu, J. Bruck, T. Kailath, and T. Hofmeister, *Depth efficient neural networks for division and related problems*, IEEE Trans. Inform. Theory, 39 (1993), pp. 946–956.

[SR] K. Y. Siu and V. Roychowdhury, *On Optimal Depth Threshold Circuits for Multiplication and Related Problems*, Technical Report ECE-92-05, University of California at Irvine, Irvine, CA, 1992.

[S1] E. D. Sontag, *Remarks on interpolation and recognition using neural nets*, in Advances in Neural Information Processing Systems 3, R. P. Lippmann, J. Moody, D. S. Touretzky, eds., Morgan Kaufmann, San Mateo, CA, 1991, pp. 939–945.

[S2] E. D. Sontag, *Feedforward nets for interpolation and classification*, J. Comput. System Sci., 45 (1992), pp. 20–48.

[S3] E. D. Sontag, private communication, July, 1992.

[V] L. G. Valiant, *A theory of the learnable*, Comm. Assoc. Comput. Mach., 27 (1984), pp. 1134–1142.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 733–750, June 1997

008

ON THE AMOUNT OF NONDETERMINISM AND THE POWER OF VERIFYING*

LIMING CAI† AND JIANER CHEN‡

Abstract. The relationship between nondeterminism and other computational resources is investigated based on the "guess-then-check" model *GC*. Systematic techniques are developed to construct natural complete languages for the classes defined by this model. This improves a number of previous results in the study of limited nondeterminism. Connections of the model *GC* to computational optimization problems are exhibited.

Key words. computational complexity, nondeterminism, complete languages, computational optimization

AMS subject classifications. 68Q05, 68Q10, 68Q15, 68Q25

PII. S0097539793258295

1. Introduction.

The study of the power of nondeterminism is central to complexity theory. The relationship between nondeterminism and other computational resources still remains unclear. Two fundamental questions are those of how much computational resource we should pay in order to eliminate nondeterminism and how much computational resource we can save if we are granted nondeterminism. A computation with nondeterminism can basically be decomposed into the phase of guessing (nondeterministically) and the phase of verifying (using other computational resources). In general, the phase of guessing and the phase of verifying work interactively.

The notion of classifying problems according to the amount of nondeterminism and the power of verifying in a computation has appeared in recent research. Díaz and Torán [14] studied the classes β_k, for $k \geq 1$, by allowing a deterministic polynomial-time computation to make an $O(\log^k n)$ amount of nondeterminism.[1] The class β_f for a more general function $f(n)$ was studied by Farr [16]. Buss and Goldsmith [5] considered the classes $N^k P_h$, for $k, h \geq 1$, in which languages can be recognized by an $O(n^h \log^{O(1)} n)$-time multitape Turing machine making at most $k \log n$ binary nondeterministic choices. Wolf [25] studied models that are *NC* circuits with nondeterministic gates. Papadimitriou and Yannakakis [21] considered a set of optimization problems that can be solved by computations with an $O(\log^2 n)$ amount of nondeterminism.

We generalize the above ideas by introducing the computation model *GC* (guess-then-check). Let $s(n)$ be a function and let $\mathcal{C}$ be a complexity class; then $GC(s(n), \mathcal{C})$ is the class of languages that can be recognized by first nondeterministically guessing

* Received by the editors November 10, 1993; accepted for publication (in revised form) July 12, 1995. A preliminary version of this paper appeared in *Proc.* 18*th International Symposium on Mathematical Foundations of Computer Science* (*MFCS* '93), Lecture Notes in Comput. Sci. 711, Springer-Verlag, Berlin, 1993, pp. 311–320.

http://www.siam.org/journals/sicomp/26-3/25829.html

† Department of Mathematics, School of Electrical Engineering and Computer Science, Ohio University, Athens, OH 45701 (cai@cs.ohiou.edu). The research of this author was supported in part by an Engineering Excellence Award from Texas A&M University.

‡ Department of Computer Science, Texas A&M University, College Station, TX 77843-3112 (chen@cs.tamu.edu). The research of this author was supported in part by National Science Foundation grant CCR-9110824.

[1] These classes were originally introduced in Kintala and Fisher [19].

$O(s(n))$ binary bits then using the power of $\mathcal{C}$ to verify. The reader should realize that the GC model is a restricted version of the interactive-proof systems that have received considerable attention recently (for a survey, see [18]).

We develop systematic and powerful techniques to show that for a large class of functions $s(n)$ and for many complexity classes $\mathcal{C}$, the class $GC(s(n), \mathcal{C})$ has natural complete languages. The techniques involve characterizing the computation of a verifier by a circuit and encoding a nondeterministic string of length $s(n)$ as an input of length $(n \cdot s(n))/\log n$ to the circuit. Our techniques improve a number of previous results in the study of completeness for complexity classes with limited nondeterminism. In particular, we show that the weight-k circuit-satisfiability problem is complete under quasi-linear-time reduction for the class N^kP_1 proposed by Buss and Goldsmith [5]. This gives a complete language for the class N^kP_1 which is more natural than the previously known complete languages for the class [5] in the sense that no explicit mention of "$k \log n$" appears in its statement. Moreover, we prove that one can obtain complete languages for the class β_k by restricting the amount of nondeterminism in NP-complete languages. This result is opposite to a conjecture made by Díaz and Torán [14] that by restricting the amount of nondeterminism in NP-complete languages, one would not get complete languages for the class β_k. We also derive complete languages for the classes $NNC^k(\log^i n)$ studied by Wolf [25], for which it was unknown whether there exist complete languages.

Of special interest is the class $GC(s(n), \Pi_k^B)$, where $s(n)$ is a function larger than $\Theta(\log n)$ and Π_k^B is the class of languages accepted by log-time alternating Turing machines of k alternations. The model $GC(s(n), \Pi_k^B)$ has guessing ability presumably stronger than and verifying ability provably weaker than deterministic polynomial-time Turing machines [27]. More careful analysis is given to show that for many functions $s(n)$ and for all integers $k > 1$, the class $GC(s(n), \Pi_k^B)$ has natural complete languages.

The importance of the class $GC(s(n), \Pi_k^B)$ is its close connection to computational optimization problems. We show that the optimization classes $LOGSNP$ and $LOGNP$ introduced by Papadimitriou and Yannakakis [21] can be precisely characterized by $GC(\log^2 n, \Pi_2^B)$ and $GC(\log^2 n, \Pi_3^B)$, respectively. An inequality $GC(\log^2 n, \Pi_2^B) \neq GC(\log^2 n, \Pi_3^B)$ is established to show the difference between $LOGSNP$ and $LOGNP$. We explain based on our characterization of the class $LOGSNP$ why the problems LOG^{2}SAT, LOG CLIQUE, and LOG CHORDLESS PATH do not seem to be complete for the class $LOGSNP$. This partially answers a question posed by Papadimitriou and Yannakakis [21]. Our characterizations also give a restricted version of the satisfiability problem that is polynomial-time equivalent to the problem TOURNAMENT DOMINATING SET, improving a result of Meggido and Vishkin [20]. The GC model also has nice applications in the study of the fixed-parameter tractability of optimization problems [15].

The paper is organized as follows. Section 2 introduces the necessary preliminaries. The model GC is defined in section 3, in which complete languages are constructed for several GC classes. Section 4 studies the GC models with very weak verifiers. Connections of the GC models to computational optimization problems are given in section 5.

2. Preliminaries. We assume reader's familiarity with the basic definitions in circuit complexity theory and alternating Turing machines. For more detailed descriptions of these topics, the reader is referred to [3, 23].

Let $b > 0$ be an integer. A string x of length n can be partitioned into $\lceil n/b \rceil$

segments of b consecutive characters (the last segment may contain less than b characters). The segments will be called *b-blocks* of x.

To simplify expressions, we will denote $2^{\lceil \log \log n \rceil}$ by $\ell(n)$. Note that $\ell(n) = \Theta(\log n)$. Moreover, a deterministic Turing machine can multiply or divide a number of $O(\log n)$ bits by $\ell(n)$ in $O(\log n)$ time. The $(\ell(n))$-blocks of a string x of length n will be simply called *ℓ-blocks* of x.

An *$O(\log n)$-time alternating Turing machine* (log-*time ATM*) is equipped with a *random-access input tape* and a read-write *input address tape* such that the Turing machine has access to the bit of the input tape denoted by the contents of the input address tape. Several input read-modes have been proposed. The one that we adopt here is the standard read-mode introduced in [11] with the restriction that in the last phase of each computation path, a log-time ATM only reads inputs from a constant number of ℓ-blocks of the input. This input read-mode is a Turing machine implementation of the *Block Transfer* mode of RAMs proposed in [1] (see [7] for discussions).

A *Π_k^B-ATM* is a log-time ATM that makes at most k alternations and must begin with $\wedge$ states. Define Π_k^B to be the class of languages accepted by Π_k^B-ATMs.

An (unbounded fan-in) *Boolean circuit* α_n with n inputs $x_1, \ldots, x_n$ is a directed acyclic graph. The nodes of fan-in 0 are called *input nodes* and are labeled from the set $\{0, 1, x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n\}$. The nodes of fan-in greater than 0 are called *gates* and are labeled either AND or OR. A set of the nodes is designated the *output* nodes. The *size* is the number of gates, and the *depth* is the maximum distance from an input to an output. Each node in a circuit of size s has a unique node number of length $O(\log s)$. We assume that circuits are of a special form where all AND and OR gates are organized into alternating levels with edges only between adjacent levels. Any circuit may be converted to one of this form without increasing the depth and by at most squaring the size [12]. In this special form, the gates connected to input nodes are called *level-1 gates*. We also assume that circuits are topologically ordered in the sense that the node number of a gate is always larger than the node numbers of its inputs.

A *family* of circuits is a sequence $F = \{\alpha_n \mid n \geq 1\}$ of circuits, where circuit α_n has n inputs and one output. A family of circuits may be used to accept a language in $\{0,1\}^*$. The circuit family F is log-*space uniform* if there is an $O(\log n)$-space deterministic Turing machine that on input 1^n prints the encoding of the circuit α_n.

Let $x = x_1 x_2 \ldots x_n$ be a string of n Boolean variables and let $b > 0$ be an integer. A *specimen* of a b-block $x_{ib+1} \ldots x_{(i+1)b}$ of x is a string $s(x_{ib+1}) \ldots s(x_{(i+1)b})$, where $s(x_j)$ is either x_j or $\overline{x}_j$, for $ib + 1 \leq j \leq (i+1)b$.

DEFINITION 2.1. *A family $F = \{\alpha_n \mid n \geq 1\}$ of circuits is called a Π_k^{poly, B_c}-*family *if there is a polynomial p such that each α_n is a circuit with an* AND *output gate and of size at most $p(n)$ and depth at most $k+1$ in which the input of each level-1 gate consists of exactly c specimens of ℓ-blocks of $x_1 x_2 \ldots x_n$.*

Cai and Chen [7] have shown the equivalence of Π_k^B-ATMs and Π_k^{poly, B_c}-families of circuits under a proper uniformity. For the present paper, the following theorem will be sufficient (see Lemma 5.2 and Remark 5.1 in [7]).

THEOREM 2.2 (see [7]). *If a language $L \in \{0,1\}^*$ is accepted by a Π_k^B-ATM M, then L is accepted by a* log-*space uniform Π_k^{poly, B_c}-family $F = \{\alpha_n \mid n \geq 1\}$ of circuits, where $c \geq 1$ is an integer. Moreover, if the input of a level-1 gate of α_n consists of specimens of ℓ-blocks $B_1, \ldots, B_c$ of the input, then there is a computation path of M that reads only from these ℓ-blocks.*

In order to let a Π_k^B-ATM simulate the computation of a circuit efficiently, we introduce a regular way to encode a circuit. An input node labeled "x_i" (resp. "$\overline{x}_i$") is called a *positive input* (resp. *negative input*).

DEFINITION 2.3. *The* normal encoding *of a circuit α with inputs $x_1 \dots x_n$ is a sequence $s = \langle n, g_1, \dots, g_m \rangle$, where the g_i's are nodes of α, $|g_i| = 2^{c\ell(n)}$ for a fixed constant c. For a gate g_i with inputs $h_1, \dots, h_j$, g_i is encoded as $\langle id, op, a(h_1), \dots, a(h_j) \rangle$, where id is the gate number, op is the gate type, and $a(h_p)$ is the address of the node h_p in the sequence s. Moreover, if g_i is a level-1 gate, then the negative inputs of g_i should be listed first.*

3. The *GC* model and complete languages. Informally, $GC(s(n), \mathcal{C})$ is a restricted interactive-proof system in which a prover passes a proof of length $O(s(n))$ to a verifier that has the power of $\mathcal{C}$. A more formal definition is given as follows.

DEFINITION 3.1. *Let $s(n)$ be a function and let $\mathcal{C}$ be a complexity class. A language L is in the class $GC(s(n), \mathcal{C})$ if there is a language $A \in \mathcal{C}$ together with an integer $c > 0$ such that for all $x \in \{0,1\}^*$, $x \in L$ if and only if $\exists y \in \{0,1\}^*$, $|y| \le c \cdot s(|x|)$, and $\langle x, y \rangle \in A$.*

We point out that in the above definition, the condition $|y| \le c \cdot s(|x|)$ can be replaced by the equality $|y| = c's(|x|)$, where $c' = 2c$, if we encode 0 by 00, 1 by 01, and "useless symbol" by 10 or 11 and use the useless symbols to make up the guessed string y to be of length exactly $2c \cdot s(|x|)$.

If we require that the length $|y|$ of the guessed string y be strictly bounded by $s(|x|)$ (i.e., the constant c be strictly 1), we call the model a "strict *GC*, " written as $sGC(s(n), \mathcal{C})$.

Many complexity classes can be characterized by the *GC* model. For example, the class *NP* can be characterized by $GC(n^{O(1)}, P)$. We will develop a systematic technique to show that many *GC* classes have natural complete languages. The technique is illustrated in detail by deriving complete languages for the classes $N^k P_1$ introduced in [5].

Let P_1 be the class of languages accepted by deterministic Turing machines of running time $O(n \log^{O(1)} n)$. A circuit family $F = \{\alpha_n \mid n \ge 1\}$ is P_1-*uniform* if there is a deterministic Turing machine M that, on input 1^n, generates the circuit α_n in time $O(n \log^{O(1)} n)$.

LEMMA 3.2. *A language L is in P_1 if and only if L is accepted by a P_1-uniform family of circuits.*

Proof. That $L \in P_1$ implies L accepted by a P_1-uniform family of circuits follows directly from the classical work of Fischer and Pippenger [17]. Conversely, if L is accepted by a P_1-uniform circuit family $F = \{\alpha_n \mid n \ge 1\}$, then L can be accepted by an $O(n \log^{O(1)} n)$-time deterministic Turing machine M as follows: on input x of length n, M first generates the circuit α_n in time $O(n \log^{O(1)} n)$, then simulates the circuit α_n on input x in time $O(|\alpha_n| \log^2 |\alpha_n|) = O(n \log^{O(1)} n)$ using an algorithm by Pippenger [22]. □

Let $N^k P_1$ denote the class of languages that is accepted by a deterministic $O(n \log^{O(1)} n)$-time Turing machine that can make at most $k \log n$ binary guesses [5]. It is easy to see that $N^k P_1$ is identical to the class $sGC(k \log n, P_1)$.

A language L is *complete* for the class $N^k P_1$ under *quasi-linear-time reduction* if (i) L is in $N^k P_1$, (ii) for every language L' in $N^k P_1$, there is a function f such that $x \in L'$ if and only if $f(x) \in L$, and (iii) the function $f(x)$ is computable in deterministic $O(n \log^{O(1)} n)$ time.

Define the *weight* of a binary string to be the number of 1's in the string. Consider the following language, where f is a function.

BWCS[f] (bounded-weight circuit satisfiability)
Instance: A circuit α of m inputs.
Question: Does α accept an input of weight $\leq f(m)$?

In particular, for any fixed constant k, we denote by BWCS[k] the language BWCS[c_k], where c_k is the constant function $c_k(n) \equiv k$.

Let d be an integer. A binary string y of length $d \log m$ is a *weight representation* of a binary string v_y of length m if the ith bit of v_y is 1 if and only if a $(\log m)$-block of y is a binary representation of the integer $i-1$.

LEMMA 3.3. *For each integer $k \geq 1$, the language* BWCS[k] *is in the class* $sGC(k \log n, P_1)$.

Proof. For each fixed integer $k \geq 1$, we construct a deterministic Turing machine M_k as follows: on an input of the form $\langle \alpha, y \rangle$, where α is a circuit with m inputs and y is a binary string of length $d \log m$, $d \leq k$, M_k writes down on a worktape a string v_y of length m whose weight representation is y. Then M_k simulates the circuit α on input v_y and accepts if and only if the circuit α accepts the input v_y.

The simulation of the circuit α on input v_y can be implemented by the algorithm of Pippenger [22], which runs in time $O(|\alpha| \log^2 |\alpha|)$. All other steps of the Turing machine M_k can easily be implemented in time $O(|\alpha| \log^{O(1)} |\alpha|)$. Thus, the language A_k accepted by M_k is in the class P_1. It is easy to see that a circuit α of m inputs is in BWCS[k] if and only if $\langle \alpha, y \rangle \in A_k$ for a binary string y of length $\leq k \log |\alpha|$. That is, the language BWCS[k] is in the class $sGC(k \log n, P_1)$. □

We introduce a special function ψ from binary numbers to binary numbers as follows.

$$\psi(x) = \begin{cases} i & \text{if } x = 0^{i-1}10^{n-i} \text{ and } i < 2^{\lfloor \log |x| \rfloor}, \\ 0^{\lfloor \log |x| \rfloor} & \text{if } x = 0^{i-1}10^{n-i} \text{ and } i \geq 2^{\lfloor \log |x| \rfloor}, \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

Note that for any binary number i of length $\lfloor \log n \rfloor$, there is a length-n binary number x of the form $0^{i-1}10^{n-i}$ such that $\psi(x) = i$. It is also easy to see that the function ψ can be computed by a P_1-uniform family of circuits.

Now we are ready to prove our first main theorem. For a circuit α, we denote by $I(\alpha)$ the input of α.

THEOREM 3.4. *The language* BWCS[k] *is complete for the class* $sGC(k \log n, P_1)$ $(= N^k P_1)$ *under quasi-linear-time reduction, for all $k \geq 1$.*

Proof. By Lemma 3.3, it suffices to show that BWCS[k] is hard for the class $sGC(k \log n, P_1)$ under quasi-linear-time reduction. Let L be a language in the class $sGC(k \log n, P_1)$. Then there is a language $A \in P_1$ such that for any x, $x \in L$ if and only if there is a $y \in \{0,1\}^*$, $|y| \leq k \log |x|$, such that $\langle x, y \rangle \in A$. We show how to reduce the language L to the language BWCS[k] in deterministic $O(n \log^{O(1)} n)$ time.

Since $A \in P_1$, by Lemma 3.2, there is a P_1-uniform family $F_A = \{\gamma_n \mid n \geq 1\}$ of circuits accepting A. Given an instance x to the language L, $|x| = n$, we consider the circuits γ_n, $\gamma_{n+1}, \ldots, \gamma_{n+k \log n}$ in F_A. Let $\tau_i(x)$ be the circuit with i inputs that is obtained from the circuit γ_{n+i} with the first n input bits being assigned by the value x, $0 \leq i \leq k \log n$. Thus $x \in L$ if and only if at least one of the circuits $\tau_i(x)$ is satisfiable.

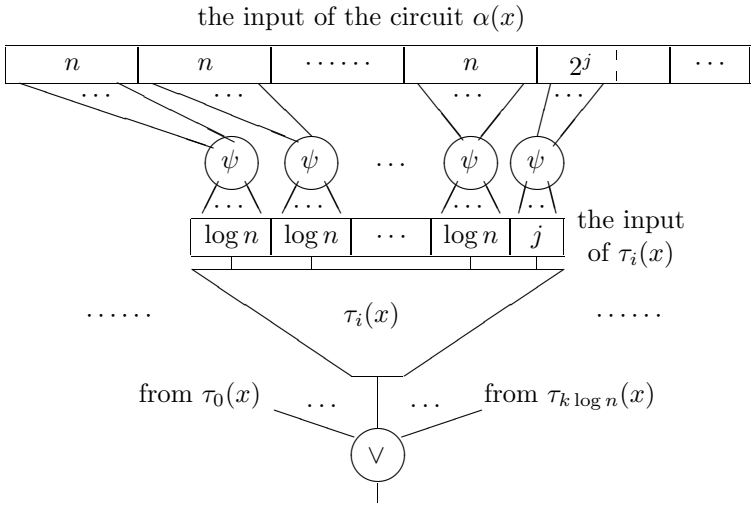


FIG. 1. *The structure of the circuit* $\alpha(x)$.

We construct a circuit $\alpha(x)$ based on these circuits $\tau_0(x), \ldots, \tau_{k\log n}(x)$. The circuit $\alpha(x)$ has kn input nodes and one output gate, plus the $k\log n + 1$ circuits $\tau_i(x)$, $0 \le i \le k\log n$. For each i, $0 \le i \le k\log n$, suppose $i = d\log n + j$, where $0 \le d \le k$ and $0 \le j < \log n$. For each q, $1 \le q \le d$, we construct a subcircuit $\sigma_{i,q}$ with n inputs and $\log n$ outputs that computes the function ψ on domain $\{0,1\}^n$ such that the n inputs of $\sigma_{i,q}$ are the qth n-block of the input $I(\alpha(x))$ of the circuit $\alpha(x)$ and the $\log n$ outputs of $\sigma_{i,q}$ replace the qth $(\log n)$-block of the input $I(\tau_i(x))$ of the circuit $\tau_i(x)$ in $\alpha(x)$. Similarly, we construct a subcircuit $\sigma_{i,d+1}$ with 2^j inputs and j outputs that computes the function ψ on domain $\{0,1\}^{2^j}$ such that the 2^j inputs of $\sigma_{i,d+1}$ are the first 2^j positions of the $(d+1)$st n-block of $I(\alpha(x))$ and the j outputs of $\sigma_{i,d+1}$ replace the $(d+1)$st $(\log n)$-block of $I(\tau_i(x))$. Finally, the output gate of the circuit $\alpha(x)$ is an OR gate that receives inputs from output gates of all the circuits $\tau_i(x)$ in $\alpha(x)$, $i = 0, 1, \ldots, k\log n$. See Figure 1.

By the definition of the function ψ, every assignment to the qth $(\log n)$-block of $I(\tau_i(x))$ can be realized in $\alpha(x)$ by a weight-1 assignment to the qth n-block of $I(\alpha(x))$. Furthermore, through the circuit $\sigma_{i,q}$, every assignment to the qth n-block of $I(\alpha(x))$ (not necessarily weight 1) realizes an assignment to the qth $(\log n)$-block of $I(\tau_i(x))$ of the circuit $\tau_i(x)$ in $\alpha(x)$. Therefore, for each $\tau_i(x)$ in $\alpha(x)$, $0 \le i \le k\log n$, every assignment to $I(\tau_i(x))$ can be realized by an assignment to $I(\alpha(x))$, in which each n-block has weight 1. Moreover, each assignment to $I(\alpha(x))$ (not necessarily with weight-1 n-blocks) realizes an assignment to $I(\tau_i(x))$ for each $\tau_i(x)$.

If the circuit $\alpha(x)$ accepts an input of weight at most k, then at least one of the circuits $\tau_i(x)$ in $\alpha(x)$ is satisfiable, which implies that $x \in L$. Conversely, suppose that $x \in L$. Then one of the circuits, say $\tau_i(x)$, is satisfiable by an assignment y to $I(\tau_i(x))$. We can construct an assignment z to $I(\alpha(x))$ in which every n-block has weight 1 such that z realizes the assignment y to $I(\tau_i(x))$. The weight of z is k. With this assignment z to $I(\alpha(x))$, the circuit $\alpha(x)$ has value 1.

By the above discussion, we conclude that $x \in L$ if and only if the circuit $\alpha(x)$ accepts an input of weight at most k, i.e., if and only if $\alpha(x)$ is in the language BWCS[k]. Moreover, since the circuit family $F_A = \{\gamma_n \mid n \ge 1\}$ is P_1-uniform,

there is a deterministic Turing machine M that constructs the circuit γ_{n+i} in time $O((n+i)\log^{O(1)}(n+i)) = O(n\log^{O(1)} n)$, for all $0 \le i \le k\log n$. Therefore, the circuit $\alpha(x)$ has size $O(n\log^{O(1)} n)$ and can be constructed in time $O(n\log^{O(1)} n)$.

Thus the language BWCS[k] is complete for the class $N^kP_1 = sGC(k\log n, P_1)$ under quasi-linear-time reduction for all $k \ge 1$. □

Theorem 3.4 presents a new complete language for the class N^kP_1 under quasi-linear-time reduction. The language BWCS[k] is more natural than the previously known complete languages for N^kP_1 (see [5]) in the sense that there is no explicit mention of "$k\log n$" or "$n\log^{O(1)} n$" in the statement of BWCS[k]. We point out that the above proofs can easily be extended to derive natural complete languages under proper reductions for the classes N^kP_h studied by Buss and Goldsmith [5], for $h > 1$.

Theorem 3.4 and its proof illustrate a systematic technique for constructing complete languages for the class $GC(s(n), \mathcal{C})$. Roughly speaking, what we need is a circuit characterization of the complexity class $\mathcal{C}$ together with an encoding of a nondeterministic string of length $s(n)$ by a length $(n \cdot s(n))/\log n$ string of weight $s(n)/\log n$. Below we list a few more examples and briefly describe the proofs.

Let f be a function. Define β_f to be the class of languages that are accepted by deterministic polynomial-time Turing machines that can make an $O(f(n))$ amount of nondeterminism. The class β_f was introduced by Kintala and Fisher [19] and studied in detail by Díaz and Torán [14]. By our GC model, the class β_f is identical to the class $GC(f(n), P)$.

THEOREM 3.5. *Let $f(n)$ be a function constructible in deterministic $O(\log n)$ space. Then the language* BWCS[f] *is complete for the class $GC(f(n)\log n, P) = \beta_{f\log n}$ under* log*-space reduction.*

Proof. The proof that BWCS[f] is in the class $GC(f(n)\log n, P)$ is similar to the proof of Lemma 3.3.

To show that BWCS[f] is hard for the class $GC(f(n)\log n, P)$ under log-space reduction, let L be a language in $GC(f(n)\log n, P)$. Then there is a language A in P with a constant c such that for all x, $x \in L$ if and only if there is a y, $|y| = cf(|x|)\log|x|$, and $\langle x, y\rangle \in A$. By the results of Borodin [4], there is a log-space uniform circuit family $F_A = \{\gamma_n \mid n \ge 1\}$ that accepts A. Given an x, $|x| = n$, let $\tau_{cf(n)\log n}(x)$ be the circuit with $cf(n)\log n$ inputs that is obtained from $\gamma_{n+cf(n)\log n}$ with the first n input bits assigned by the value x. Thus $x \in L$ if and only if the circuit $\tau_{cf(n)\log n}(x)$ is satisfiable.

Now the proof goes similarly to that of Theorem 3.4. We construct a circuit $\alpha(x)$ based on the circuit $\tau_{cf(n)\log n}(x)$. The circuit $\alpha(x)$ has $m = f(n)n^c$ inputs, and for each (n^c)-block of $I(\alpha(x))$, there is a subcircuit with n^c inputs that computes the function ψ whose n^c inputs are from the (n^c)-block of $I(\alpha(x))$ and whose $c\log n$ outputs are connected to the corresponding $(c\log n)$-block of the input of $\tau_{cf(n)\log n}(x)$. Now if the circuit $\alpha(x)$ accepts an input of weight at most $f(m)$, then this input of $\alpha(x)$ should produce a satisfiable input to the circuit $\tau_{cf(n)\log n}(x)$ via the ψ function, which implies that $x \in L$. Conversely, if $x \in L$, then a weight-$f(n)$ input of $\alpha(x)$, $f(n) \le f(m)$, can be constructed to produce via the ψ function a satisfiable input to the circuit $\tau_{cf(n)\log n}(x)$. Thus this weight-$f(n)$ input should satisfy the circuit $\alpha(x)$, and $\alpha(x)$ is in BWCS[f]. □

In particular, the language BWCS[$\log^{i-1} n$] is complete for the class $GC(\log^i n, P) = \beta_{\log^i n} \stackrel{\text{def}}{=} \beta_i$, for all integers $i \ge 2$. Note that the language BWCS[$\log^{i-1} n$] is a restricted version of the circuit-satisfiability problem that is complete for the class *NP*. This answers a question posed by Díaz and Torán [14], who were able to

construct complete languages for β_i from certain complete languages for the class P by adding nondeterminism, and who conjectured that one might not be able to construct complete languages for β_i from complete languages for the class NP by restricting nondeterminism. We point out that a different approach has been adopted by Szelepcsényi [24] and Farr [16] to study complete languages for the class β_f.

Let f be a function. Now we consider the class $GC(f(n), NC^d)$ for all $d \geq 1$. In particular, the classes $GC(\log^i n, NC^d)$ have been investigated by Wolf [25]—in his notation, $NNC^d(\log^i n)$.

Consider the following language.

BWCS$_d[f]$
Instance: A circuit α with m inputs and depth $\log^d m$.
Question: Does α accept an input of weight $\leq f(m)$?

THEOREM 3.6. *Let f be a function constructible in deterministic $O(\log n)$ space and let $d \geq 1$ be an integer. Then the language* BWCS$_d[f]$ *is complete for the class $GC(f(n)\log n, NC^d)$ under* log*-space reduction.*

Proof. Using the *universal-depth NC^d* circuit family developed by Cook and Hoover [13], we can show that the language BWCS$_d[f]$ is in $GC(f(n)\log n, NC^d)$.

The proof that the language BWCS$_d[f]$ is hard for the class $GC(f(n)\log n, NC^d)$ under log-space reduction is completely similar to that of Theorem 3.5. □

In particular, the language BWCS$_d[\log^{i-1} n]$ is complete for the class $NNC^d(\log^i n)$ under log-space reduction, for all $i \geq 2$. This is the first language that is known to be complete for the class $NNC^d(\log^i n)$.

4. The *GC* classes with very weak verifiers. In this section, we derive complete languages for GC classes with verifiers strictly weaker than deterministic polynomial-time Turing machines.

The basic idea here is the same as that in section 3: we characterize the verifier by a circuit family and encode the nondeterministic string of length $s(n)$ by an input of length $(n \cdot s(n))/\log n$ to the circuit. However, implementation of these methods on models such as Π_k^B-ATMs becomes more subtle. For example, it would not be proper to encode the string of length $s(n)$ using the function ψ defined in section 3 because that would increase the depth of the circuit. Similar difficulty also occurs when we test the bounded-weight satisfiability of circuits of depth k using a Π_k^B-ATM. We will present a number of new techniques to overcome these difficulties.

Recall that in the last phase of each computation path, a Π_k^B-ATM M can read input bits from at most a constant number of ℓ-blocks of the input. We further require that if the input to a Π_k^B-ATM M is of the form $\langle x, y\rangle$, then the last phase of each computation path of M can read input bits from at most *one* ℓ-block from the second string y. It is easy to see that this additional requirement has no influence on the class accepted by Π_k^B-ATMs. Furthermore, we also assume that the pairing function $\langle x, y\rangle$ is simple and can be decoded into x and y in deterministic $O(\log n)$ time [2].

We first consider the GC classes whose verifier is Π_{2k}^B, for $k \geq 1$. A circuit is a Π-*circuit* if it has a single output gate, which is of type AND. A circuit α is *semimonotone* if at most two inputs of each level-1 gate of α are negative input nodes.

DEFINITION 4.1. BWCS$(s(n), k)$ *is the set of pairs $x = \langle \alpha, w\rangle$, where α is a semimonotone* Π*-circuit of depth k in the normal encoding such that α accepts an input of weight w, $w \leq s(|x|)$.*

LEMMA 4.2. *For any function $s(n)$ and any $k \geq 1$, the language* BWCS $(s(n), 2k)$ *is in the class $GC(s(n)\ell(n), \Pi_{2k}^B)$.*

Algorithm BWCS-Simulator

Input: $\langle x, y\rangle$, where $x = \langle \alpha, w\rangle$ and α is a circuit with m inputs.

1. Reject if $w > m$, or if y does not consist of exactly w different input-node numbers of α, each of length $\ell(|x|)$, or if α is not in the normal encoding, or if α is not a semimonotone Π-circuit of depth $2k$;
2. Let g be the output gate of α;
3. Repeat the following loop $(2k-1)$ times:
   Suppose $g = \langle id, op, a(h_1), \ldots, a(h_j)\rangle$:
   If $op =$ AND, then universally choose an $a(h_i)$. Let $g = h_i$;
   If $op =$ OR, then existentially guess an $a(h_i)$. Let $g = h_i$;
4. {At this point, $g = \langle id, op, a(h_1), \ldots, a(h_j)\rangle$ is a level-1 gate.}
   (a) Deterministically find the negative input nodes in h_1, ..., h_j. For each such negative input node h_i, universally check whether h_i is contained in y.
   (b) If any of these negative input nodes is not contained in y, then accept; otherwise, existentially guess a positive input node $a(h_i)$ of g and accept if and only if the node number of h_i is contained in y.

FIG. 2. *The algorithm* BWCS-*Simulator.*

Proof. Consider the algorithm BWCS-Simulator in Figure 2. We will prove that this algorithm can be implemented by a Π^B_{2k}-ATM such that for all $x = \langle \alpha, w\rangle$, $x \in \mathrm{BWCS}(s(n), 2k)$ if and only if there is a string $y \in \{0,1\}^*$, $|y| \le s(|x|)\ell(|x|)$ such that the algorithm accepts $\langle x, y\rangle$.

Let $z = \langle x, y\rangle$ be an input to the algorithm BWCS-Simulator, where $x = \langle \alpha, w\rangle$. In deterministic $O(\log |z|)$ time, we can compute the lengths $|x|$ and $|y|$ (see [2]) and check the relation $w > m$. Note that $w \le m$ implies $|w| \le \log |z|$, so multiplying and dividing w by $\ell(|x|)$ can be done in deterministic $O(\log |z|)$ time.

The Boolean string y of length $w\ell(|x|)$ is used as the weight representation of an input v_y of weight w to the circuit α. To verify that y contains exactly w different input-node numbers of α, each of length $\ell(|x|)$, we universally check that (i) $|y| = w\ell(|x|)$, (ii) each $\ell(|x|)$-block of y is a binary number $\le m$, and (iii) no two $\ell(|x|)$-blocks of y are identical. To check whether α is in the normal encoding and whether α is a semimonotone Π-circuit of depth $2k$, we universally check the gates of α level by level, starting from the output gate of α.

Since the output gate of α is an AND gate, the first execution of the loop in step 3 is a universal branch. Therefore, steps 1 and 2 combined with the first execution of the loop in step 3 form the first phase, which is a universal phase of the algorithm.

The loop execution of step 3 simply simulates the computation of the circuit α. After $2k-1$ executions of the loop in step 3, the algorithm is in its $(2k-1)$st phase, which is a universal phase, and the current gate g is a level-1 gate of the circuit α, which is an OR gate. Therefore, the universal checking in step 4(a) can be combined into the $(2k-1)$st phase. Since α is semimonotone, only h_1 and h_2 can be negative input nodes to the gate g, which can be checked in deterministic $O(\log |z|)$ time.

If any of the negative input nodes of g is not in y, then all universal branches in step 4(a) accept according to the "then" part of step 4(b). This is correct because this forces the gate g to have value 1 on input v_y. On the other hand, if all negative input nodes of g are in y, then we must check the positive input nodes of g. Thus the algorithm starts its $(2k)$th phase in the "else" part of step 4(b). Each path in this phase checks whether a positive input node h_i of g is contained in y. For this, the algorithm existentially checks an $\ell(|x|)$-block in y. Note that in this phase, the algorithm reads one $\ell(|x|)$-block from the first string x (the address $a(h_i)$ of the gate h_i) and one $\ell(|x|)$-block from the second string y in the input $\langle x, y\rangle$.

This concludes that the algorithm BWCS-Simulator is a Π^B_{2k}-ATM which accepts $\langle x, y\rangle$, where $x = \langle \alpha, w\rangle$, if and only if y contains exactly w different input-node numbers of α and the circuit α accepts the weight-w input v_y whose weight representation is y.

Now it is easy to see that $x = \langle \alpha, w\rangle \in \mathrm{BWCS}(s(n), 2k)$ if and only if the Π^B_{2k}-ATM BWCS-Simulator accepts $\langle x, y\rangle$ for a binary string of length bounded by $s(|x|)\ell(|x|)$. Thus the language $\mathrm{BWCS}\,(s(n), 2k)$ is in the class $GC(s(n)\ell(n), \Pi^B_{2k})$. □

Now we show the completeness of $\mathrm{BWCS}(s(n), 2k)$ in the class $GC(s(n)\ell(n), \Pi^B_{2k})$. A proof for the following lemma can be found in [6].

LEMMA 4.3. *Let $b > 0$ and $c > 0$ be two integers, and let τ be a Π-circuit of depth k in which the input of each level-1 gate is a specimen of a b-block of $x_1 \dots x_n$. Then there is a Π-circuit γ of depth k and size $\le 2^{cb}\mathit{size}(\tau)$ computing the same function in which the input of each level-1 gate is a specimen of a (cb)-block of $x_1 \dots x_n$.*

THEOREM 4.4. *Let $s(n) \le n$ be a nondecreasing function computable in deterministic $O(\log n)$ space. Then the language $\mathrm{BWCS}(s(n), 2k)$ is complete for the class $GC(s(n)\ell(n), \Pi^B_{2k})$ under $O(\log n)$-space reduction, for $k \ge 1$.*

Proof. By Lemma 4.2, we only need to prove hardness.

Let L be a language in $GC(s(n)\ell(n), \Pi^B_{2k})$. By definition, there is a Π^B_{2k}-ATM M with an integer $c > 0$ such that $x \in L$ if and only if there is a $y \in \{0,1\}^*$, $|y| = c\,s(|x|)\ell(|x|)$, and M accepts $\langle x, y\rangle$. Without loss of generality, assume that c is an even number. Moreover, we can assume that no phase except the last phase of each computation path of M has access to the input [7]. By Theorem 2.2, there is a log-space uniform Π_{2k}^{poly, B_h}-family $\{\tau_m \mid m \ge 1\}$ of circuits that accepts $L(M)$, where h is a constant.

Given an instance x of the language L, we show how to reduce x to an instance $z = \langle \alpha(x), w(x)\rangle$ for the language $\mathrm{BWCS}\,(s(n), 2k)$ such that $x \in L$ if and only if z is in $\mathrm{BWCS}\,(s(n), 2k)$. Let $|x| = n$, and $|\langle x, y\rangle| = m$, where y is a binary string of length $c\,s(n)\ell(n)$. Let $\tau_m(x)$ be the circuit τ_m with the first part of the input assigned by the value of x. $\tau_m(x)$ is a circuit with $c\,s(n)\ell(n)$ inputs, and $x \in L$ if and only if the circuit $\tau_m(x)$ is satisfiable.

Since $|y| = c\,s(n)\ell(n) = O(n \log n)$, $|x| = n \le m = |\langle x, y\rangle| \le n^2$. Therefore, we have either $\ell(m) = \ell(n)$ or $\ell(m) = 2\ell(n)$. Thus the number of inputs of the circuit $\tau_m(x)$ can be written as $as(n)\ell(m)$, where a is an integer such that $a = c$ if $\ell(m) = \ell(n)$ or $a = c/2$ if $\ell(m) = 2\ell(n)$.

Recall that the circuit τ_m is a Π-circuit of depth $2k + 1$ in which the input of each level-1 gate consists of exactly h specimens of ℓ-blocks of $\langle x, y\rangle$. Moreover, the last phase of each computation path of M reads inputs from at most one ℓ-block from the string y. By Theorem 2.2, at most one specimen in the input of each level-1 gate of τ_m is from an ℓ-block of the string y. Therefore, the circuit $\tau_m(x)$ is a Π-circuit with $as(n)\ell(m)$ inputs and of depth $2k + 1$ in which the input of each level-1 gate is a specimen of an $\ell(m)$-block of its input. By Lemma 4.3, there is a Π-circuit $\gamma_m(x)$ of depth $2k + 1$ and size bounded by a polynomial of m that computes the same function as $\tau_m(x)$ such that the input of each level-1 gate of $\gamma_m(x)$ is a specimen of an $(a\ell(m))$-block of its input.

Now we are ready to describe the circuit $\alpha(x)$.

The input $I(\alpha(x)) = v_1 \dots v_{s(n)2^{a\ell(m)}}$ of the circuit $\alpha(x)$ is of length $s(n)2^{a\ell(m)}$ and partitioned into $s(n)$ $2^{a\ell(m)}$-blocks. Similarly, the input $I(\gamma_m(x)) = u_1 \dots u_{as(n)\ell(m)}$ of the circuit $\gamma_m(x)$ is partitioned into $s(n)$ $(a\ell(m))$-blocks. The circuit $\alpha(x)$ will be

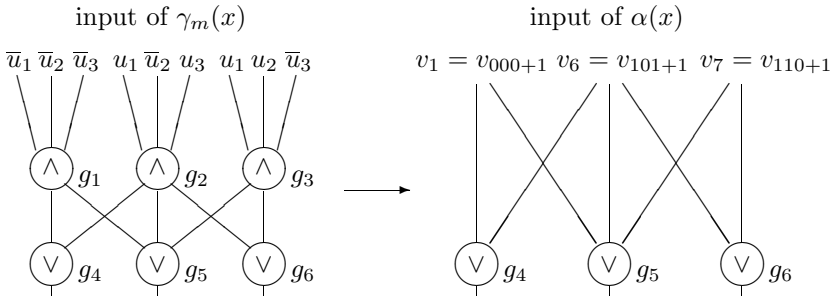


FIG. 3. *From circuit $\gamma_m(x)$ to circuit $\alpha(x)$.*

constructed from the circuit $\gamma_m(x)$ by replacing each level-1 gate in $\gamma_m(x)$ by an input node of $\alpha(x)$. As we have done in section 3, for each q, we will use a position in the qth $2^{a\ell(m)}$-block of $I(\alpha(x))$ to represent an assignment to the qth $(a\ell(m))$-block of $I(\gamma_m(x))$. To simplify discussion, we describe the construction of the first $2^{a\ell(m)}$-block of $I(\alpha(x))$ based on the first $(a\ell(m))$-block of $I(\gamma_m(x))$. The construction of the qth $2^{a\ell(m)}$-block of $I(\alpha(x))$ for general q can be done similarly.

Let g be a level-1 gate of the circuit $\gamma_m(x)$ whose input is a specimen of the first $(a\ell(m))$-block of $I(\gamma_m(x))$. Note that g is an AND gate; thus there is a unique Boolean assignment b_g to the first $(a\ell(m))$-block of $I(\gamma_m(x))$ that makes the gate g have value 1. Regarding b_g as a binary number between 0 and $2^{a\ell(m)} - 1$, we replace the gate g in $\gamma_m(x)$ by the positive input node v_{b_g+1} in $I(\alpha(x))$, which is in the first $2^{a\ell(m)}$-block of $I(\alpha(x))$. In the same way, we perform this replacement on each level-1 gate in the circuit $\gamma_m(x)$. See Figure 3. The resulting circuit $\alpha(x)$ has depth $2k$. Also, note that so far the circuit $\alpha(x)$ contains no negative input nodes.

By construction, each assignment of the qth $(a\ell(m))$-block of $I(\gamma_m(x))$ in $\gamma_m(x)$ can be implemented by a weight-1 assignment of the qth $2^{a\ell(m)}$-block of the input $I(\alpha(x))$ in the circuit $\alpha(x)$. Therefore, each assignment of $I(\gamma_m(x))$ in $\gamma_m(x)$ can be implemented by a weight-$s(n)$ assignment of $I(\alpha(x))$ in $\alpha(x)$, in which the assignment to each $2^{a\ell(m)}$-block has weight 1. We conclude that if the circuit $\gamma_m(x)$ is satisfiable, then the circuit $\alpha(x)$ accepts an input of weight $s(n)$.

The construction has not yet been completed. Note that if in an assignment of $I(\alpha(x))$, a $2^{a\ell(m)}$-block has weight different from 1, then the assignment does not implement any assignment of $I(\gamma_m(x))$. To ensure that each $2^{a\ell(m)}$-block of $I(\alpha(x))$ is assigned exact one 1, we let

$$\phi(v_1, \ldots, v_t) = (v_1 \vee \ldots \vee v_t) \bigwedge \left(\bigwedge_{i,j} (\overline{v}_i \vee \overline{v}_j) \right).$$

It is easy to see that $\phi(v_1, \ldots, v_t) = 1$ if and only if exactly one v_i is 1. The function $\phi(v_1, \ldots, v_t)$ can be implemented by a Π-circuit of depth 2 in which at most two negative input nodes appear in the input of each level-1 gate. Now for each $2^{a\ell(m)}$-block of $I(\alpha(x))$, we add a Π-subcircuit of depth 2 that implements the function ϕ with $2^{a\ell(m)}$ variables, and we connect the output of this subcircuit to the output gate of the circuit $\alpha(x)$ (which is the output gate of the circuit $\gamma_m(x)$). Note that this does not increase the depth of the circuit $\alpha(x)$ since both the output gate of such a subcircuit and the output gate of the circuit $\alpha(x)$ are AND gates, and the depth of the circuit $\alpha(x)$ is at least 2. This completes the construction of the circuit $\alpha(x)$.

Algorithm antiBWCS-Simulator

Input: $\langle x, y\rangle$, where $x = \langle \alpha, w\rangle$ and α is a circuit with m inputs.

1. Reject if $w > m$, or if y does not consist of exactly w different input node numbers, each of length $\ell(|x|)$, or if α is not in the normal encoding, or if α is not a semiantimonotone Π-circuit of depth $2k+1$;
2. Let g be the output gate of α;
3. Repeat the following loop $2k$ times:
   Suppose $g = \langle id, op, a(h_1), \ldots, a(h_j)\rangle$:
   If $op =$ AND, then universally choose an $a(h_i)$. Let $g = h_i$;
   If $op =$ OR, then existentially guess an $a(h_i)$. Let $g = h_i$;
4. {At this point, $g = \langle id, op, a(h_1), \ldots, a(h_j)\rangle$ is a level-1 gate.}
   (a) Deterministically find the positive input nodes in $h_1, \ldots, h_j$. For each such positive input node h_i, existentially check whether h_i is contained in y.
   (b) If any of these positive input nodes is not contained in y, then reject; otherwise, universally guess a negative input node $a(h_i)$ of g and reject if and only if the node number of h_i is contained in y.

FIG. 4. *The algorithm anti*BWCS-*Simulator.*

The circuit $\alpha(x)$ is semimonotone because only the subcircuits that implement the function ϕ contain negative input nodes.

Now if the circuit $\alpha(x)$ accepts an assignment of weight $s(n)$, then each $2^{a\ell(m)}$-block of the assignment must have weight 1. Thus the assignment implements a satisfying assignment to the circuit $\gamma_m(x)$.

Since the circuit $\gamma_m(x)$ is satisfiable if and only if $x \in L$, we conclude that the circuit $\alpha(x)$ accepts a weight-$s(n)$ input if and only if $x \in L$. Consequently, the mapping from x to $\langle \alpha(x), s(n)\rangle$ is a many–one reduction from the language L to the language BWCS$(s(n), 2k)$.

By assumption, the function $s(n)$ can be constructed in deterministic $O(\log n)$ space and the circuit τ_m can be constructed in deterministic $O(\log m) = O(\log n)$ space. It is also easy to see that the circuit constructions from τ_m to $\tau_m(x)$, from $\tau_m(x)$ to $\gamma_m(x)$, and from $\gamma_m(x)$ to $\alpha(x)$ (in the normal form) can all be done in deterministic $O(\log n)$ space. Therefore, the reduction can be implemented in deterministic $O(\log n)$ space. This completes the proof of the theorem. □

Unfortunately, the above methods do not seem to work for $GC(s(n)\ell(n), \Pi_{2k+1})$. In fact, we are even unable to prove that the language BWCS$(s(n), 2k+1)$ is in the class $GC(s(n)\ell(n), \Pi_{2k+1})$. For this, we need to introduce another special type of circuits that are dual to the semimonotone circuits. A circuit α is *semiantimonotone* if at most two inputs of each level-1 gate of α are positive input nodes.

DEFINITION 4.5. *anti* BWCS$(s(n), k)$ *is the set of pairs* $x = \langle \alpha, w\rangle$*, where* α *is a semiantimonotone* Π*-circuit of depth* k *in the normal encoding such that* α *accepts an input of weight* w*,* $w \le s(|x|)$*.*

LEMMA 4.6. *For any function* $s(n)$ *and any* $k \ge 1$*,* antiBWCS$(s(n), 2k+1)$ *is in the class* $GC(s(n)\ell(n), \Pi^B_{2k+1})$*.*

Proof. We first design an algorithm antiBWCS-Simulator as shown in Figure 4. As in the proof of Lemma 4.2, we can prove that the algorithm antiBWCS-Simulator can be implemented by a Π^B_{2k+1}-ATM M. In particular, since the circuit α is in the normal encoding, the positive input nodes should always appear at the end of the input list of a level-1 gate. Thus they can be found in deterministic $O(\log n)$ time since α is semiantimonotone. It can also be proved that M accepts $\langle x, y\rangle$, where $x = \langle \alpha, w\rangle$, if and only if y contains exactly w different input-node numbers of the semiantimonotone circuit α, and the circuit α accepts the input v_y of weight w whose weight

representation is y. Consequently, for any $x = \langle \alpha, w \rangle$, $x \in \text{antiBWCS}(s(n), 2k+1)$ if and only if there is a string $y \in \{0,1\}^*$ of length $s(|x|)\ell(|x|)$ such that the algorithm antiBWCS-Simulator accepts $\langle x, y \rangle$. We leave the detailed proof to the interested reader. □

THEOREM 4.7. *Let $s(n) \leq n$ be a nondecreasing function computable in deterministic $O(\log n)$ space. Then* antiBWCS$(s(n), 2k+1)$ *is complete for the class $GC(s(n)\ell(n), \Pi^B_{2k+1})$ under $O(\log n)$-space reduction, for $k \geq 1$.*

Proof. Let L be a language in $GC(s(n)\ell(n), \Pi^B_{2k+1})$. For an instance x of L, let $n = |x|$, $m = |\langle x, y \rangle|$, where y is a binary string of length $as(n)\ell(m)$ for an integer $a > 0$. As in the proof of Theorem 4.4, we can construct a Π-circuit $\gamma_m(x)$ with $as(n)\ell(m)$ inputs and of depth $2k+2$ in which the input of each level-1 gate is a specimen of an $(a\ell(m))$-block of its input such that $x \in L$ if and only if the circuit $\gamma_m(x)$ is satisfiable.

The construction of the circuit $\alpha(x)$ from the circuit $\gamma_m(x)$ is in some sense dual to the one in the proof of Theorem 4.4.

Let g be a level-1 gate of the circuit $\gamma_m(x)$ whose input is a specimen of the first $(a\ell(m))$-block of $I(\gamma_m(x))$. Note that g is an OR gate; thus there is a unique Boolean assignment b_g to the first $(a\ell(m))$-block of $I(\gamma_m(x))$ that makes the gate g have value 0. Regarding b_g as a binary number between 0 and $2^{a\ell(m)} - 1$, we replace the gate g in $\gamma_m(x)$ by the negative input node $\overline{v}_{b_g+1}$ in $I(\alpha(x))$. We perform this replacement on each level-1 gate in the circuit $\gamma_m(x)$. The resulting circuit $\alpha(x)$ has depth $2k+1$. Also, note that so far the circuit $\alpha(x)$ contains no positive input nodes.

To ensure that each $2^{a\ell(m)}$-block of $I(\alpha(x))$ is assigned exactly one 1, we again use the function ϕ in the proof of Theorem 4.4. However, this time we implement the function ϕ by a Π-circuit C_ϕ of depth 3 so that all level-1 gates of the circuit C_ϕ have fan-in 1. Thus the circuit C_ϕ is semiantimonotone. Now including the subcircuits C_ϕ into the circuit $\alpha(x)$ ensures that the circuit $\alpha(x)$ accepts an input of weight $s(n)$ if and only if the circuit $\gamma_m(x)$ is satisfiable. Moreover, adding the subcircuits C_ϕ to $\alpha(x)$ does not increase the depth of the circuit $\alpha(x)$ because the depth of $\alpha(x)$ is at least 3.

All other parts of the proof are exactly the same as that in the proof of Theorem 4.4. □

In many cases, the function $\ell(n)$ in $GC(s(n)\ell(n), \Pi^B_k)$ can be replaced by $\log n$, as stated in the following theorem. A proof of this theorem can be found in [6].

THEOREM 4.8. *If the function $s(n)$ is computable in deterministic $O(\log n)$ time and $|s(n)| = O(\log n / \log\log n)$ for all n, then for all $k \geq 1$,*

$$GC(s(n)\ell(n), \Pi^B_k) = GC(s(n)\log n, \Pi^B_k).$$

5. *GC* classes and optimization problems. In this section, we present a number of interesting connections of the GC classes to computational optimization problems.

Following Papadimitriou and Yannakakis [21], define $LOGNP_0$ to be the class of all problems described as follows:

$$\{I : \exists S \in [n]^{\log n},\ \forall x \in [n]^p,\ \exists y \in [n]^q,\ \forall j \in [\log n],\ \phi(I, s_j, x, y, j)\}, \tag{1}$$

where $I \subseteq [n]^m$ is the input relation, x and y are tuples of first-order variables ranging over $[n] = \{1, 2, \ldots, n\}$, j is a first-order variable ranging over $[\log n]$, S is an ordered subset $S = (s_1, \ldots, s_{\log n})$ of $[n]$, and ϕ is a quantifier-free first-order

expression involving the relation symbol I and the variables in x and y as well as the variables j and s_j.

A weaker class $LOGSNP_0$ contains all problems definable by one less alternation of quantifiers:

$$\{I : \exists S \in [n]^{\log n},\ \forall x \in [n]^p,\ \exists j \in [\log n],\ \phi(I, s_j, x, j)\}. \tag{2}$$

The class $LOGNP$ is defined to be the class of languages that can be polynomial-time reduced to a problem in $LOGNP_0$, and the class $LOGSNP$ is defined to be the class of languages that can be polynomial-time reduced to a problem in $LOGSNP_0$.

The complexity of a number of interesting optimization problems can be nicely characterized by the classes $LOGNP$ and $LOGSNP$. For instance, it has been shown [21] that the problems LOG DOMINATING SET, TOURNAMENT DOMINATING SET, RICH HYPERGRAPH COVER, and LOG ADJUSTMENT are complete under polynomial-time reduction for the class $LOGSNP$ and that the problem V-C DIMENSION is complete under polynomial-time reduction for the class $LOGNP$.

The following two theorems characterize the classes $LOGNP$ and $LOGSNP$ by the GC models.

THEOREM 5.1. *A language L is in the class $LOGSNP$ if and only if L is polynomial-time reducible to a language in $GC(\log^2 n, \Pi_2^B)$.*

Proof. We first show that the language BWCS$(\log n, 2)$ is in the class $LOGSNP_0$ when circuits are encoded properly. A circuit is in the *edge-relation encoding* if it is represented by a collection of five kinds of tuples: $e(g, g')$ if node g is an input of node g', $p(i, g)$ if the ith input of the level-1 gate g is a positive input node, $n(i, g)$ if the ith input of the level-1 gate g is a negative input node, $pos(k, g)$ if the positive input node x_k is an input of the level-1 gate g, and $neg(k, g, i)$ if the ith input of the level-1 gate g is the negative input node $\overline{x}_k$. Note that for a semimonotone circuit, $neg(k, g, i)$ is false for all $i \geq 3$. It is easy to see that the edge-relation encoding of circuits and the normal encoding of circuits can be converted to each other in polynomial time.

With the edge-relation encoding, it is easy to see the language BWCS$(\log n, 2)$ is defined by the following logic expression:

$$\{\langle \alpha, w\rangle : \exists S \in [n]^{\log n},\ \forall g \in [n],\ \forall a, b \in [\log n],\ \exists j \in [\log n],\ ([A] \wedge [B] \wedge [C] \wedge [D])\},$$

where

$$\begin{aligned}
[A] &= e(g, g_0) \wedge p(1, g) \rightarrow pos(s_j, g),\\
[B] &= e(g, g_0) \wedge n(1, g) \wedge p(2, g) \wedge (a \leq w)\\
&\qquad \rightarrow pos(s_j, x) \vee [(a = j) \wedge \overline{neg(s_j, g, 1)}],\\
[C] &= e(g, g_0) \wedge n(1, g) \wedge n(2, g) \wedge (a, b \leq w)\\
&\qquad \rightarrow pos(s_j, x) \vee [(a = j) \wedge \overline{neg(s_j, g, 1)}] \vee [(b = j) \wedge \overline{neg(s_j, g, 2)}],\\
[D] &= (j \leq w),
\end{aligned}$$

where g_0 is the output gate of the circuit α.

Thus the language BWCS$(\log n, 2)$ is in the class $LOGSNP_0$ when circuits are in the edge-relation encoding. By Theorems 4.4 and 4.8, the language BWCS$(\log n, 2)$ is complete under $O(\log n)$-space reduction for class $GC(\log^2 n, \Pi_2^B)$ when circuits are in the normal encoding. Since the normal encoding of a circuit can easily be converted into the edge-relation encoding of the circuit, we conclude that all languages in the class $GC(\log^2 n, \Pi_2^B)$ are in $LOGSNP$. Consequently, all languages that are polynomial-time reducible to a language in the class $GC(\log^2 n, \Pi_2^B)$ are contained in the class $LOGSNP$.

To prove the inverse, we consider the following problem. A *tournament* is a directed graph in which for any two vertices exactly one of the two directed edges is presented. It is not difficult to see that a tournament of n vertices has a dominating set of size $\log n$ [21]. Thus the following problem has a trivial solution when $k \geq \log n$.

TOURNAMENT DOMINATING SET: "Given a tournament with n vertices and integer k, does it have a dominating set of size k?"

Papadimitriou and Yannakakis [21] have shown that TOURNAMENT DOMINATING SET is complete for *LOGSNP* under polynomial-time reduction. Therefore, to show that all languages in *LOGSNP* are polynomial-time reducible to a language in the class $GC(\log^2 n, \Pi_2^B)$, we only have to show that TOURNAMENT DOMINATING SET is in $GC(\log^2 n, \Pi_2^B)$. It can be easily done as follows: given an input $\langle G, k, y\rangle$, where y is of length $k \log n$ and encodes k vertices of the tournament G, the Π_2^B-ATM M first universally picks a vertex v of G, then existentially guesses a vertex w in y, and finally verifies if $v = w$ or $[w, v]$ is an edge of G.

This completes the proof. □

The proof of Theorem 5.1 also shows that TOURNAMENT DOMINATING SET is complete for the class $GC(\log^2 n, \Pi_2^B)$ under polynomial-time reduction.

THEOREM 5.2. *A language is in the class LOGNP if and only if it is polynomial-time reducible to a language in* $GC(\log^2 n, \Pi_3^B)$.

Proof. Since the proof is similar to that of Theorem 5.1, we only describe the differences here.

The language antiBWCS$(\log n, 3)$ can be given by the following logic expression:

$$\{\langle \alpha, w\rangle : \exists S \in [n]^{\log n},\ \forall g' \in [n],\ \exists g \in [n],\ \exists a, b \in [\log n],\ \forall j \in [\log n],\ ([A] \wedge [B] \wedge [C] \wedge [D])\},$$

where

$$[A] = e(g', g_0) \wedge e(g, g') \wedge n(1, g) \wedge (j \leq w) \rightarrow \overline{neg(s_j, g)},$$
$$[B] = e(g', g_0) \wedge e(g, g') \wedge p(1, g) \wedge n(2, g) \wedge (j \leq w) \rightarrow \overline{neg(s_j, x)} \wedge [(a \neq j) \vee pos(s_j, g, 1)],$$
$$[C] = e(g', g_0) \wedge e(g, g') \wedge p(1, g) \wedge p(2, g) \wedge (j \leq w) \rightarrow \overline{neg(s_j, x)} \wedge [(a \neq j) \vee pos(s_j, g, 1)] \wedge [(b \neq j) \vee pos(s_j, g, 2)],$$
$$[D] = (a \leq w) \wedge (b \leq w).$$

To show that every language in the class *LOGNP* is polynomial-time reducible to a language in $GC(\log^2 n, \Pi_3^B)$, we will show that the problem V-C DIMENSION, which is known to be complete for the class *LOGNP* [21], is contained in the class $GC(\log^2 n, \Pi_3^B)$.

Let $\mathcal{C}$ be a family of subsets of a universe U. The *V-C dimension* of $\mathcal{C}$ is the largest cardinality of a subset S of U such that the following holds: For all subsets T of S there is a set $C[T] \in \mathcal{C}$ such that $S \cap C[T] = T$. Closer inspection reveals that the V-C dimension of a family $\mathcal{C}$ is at most $\log |\mathcal{C}|$.

V-C DIMENSION: "Given a finite family $\mathcal{C}$ of finite sets and an integer k, is the V-C dimension of $\mathcal{C}$ at least k?"

To show that the problem V-C DIMENSION is in the class $GC(\log^2 n, \Pi_3^B)$, we construct a Π_3^B-ATM M as follows. Let $n = |\mathcal{C}|$. On input $\langle \mathcal{C}, k, y\rangle$, where y is a binary string of length $k \log n$ that encodes a subset S of U of k elements, the Π_3^B-ATM M first universally picks a subset T of S (this can be done by picking a binary string of length k to indicate which elements of S are included in T), then existentially guesses a subset $C[T]$ in $\mathcal{C}$, and finally universally checks for each element v in S that v is in T if and only if v is in $C[T]$. It is not hard to see that with a proper encoding of the

family $\mathcal{C}$, the machine M can be implemented by a Π_3^B-ATM. □

The proof of Theorem 5.2 also shows that V-C DIMENSION is complete for the class $GC(\log^2 n, \Pi_3^B)$ under polynomial-time reduction.

One may not expect an easy proof that the classes *LOGSNP* and *LOGNP* are different because that would imply $P \neq NP$. However, based on the *GC* model characterizations of these classes, we can show strong evidence that these two classes are distinct.

THEOREM 5.3. *For all integer $k \geq 1$, the class $GC(\log^2 n, \Pi_k^B)$ is a proper subclass of the class $GC(\log^2 n, \Pi_{k+1}^B)$.*

Proof. By Boppana and Sipser [3, Thm. 3.13], for any integer $k \geq 1$, there is a language L_{k+1} that is accepted by a Π_{k+1}^B-ATM but not by any family of Σ-circuits of size $O(n^{O(\log n)})$ and depth $k+2$ in which the fan-in of the level-1 gates is bounded by $O(\log n)$. Thus the language L_{k+1} is in the class $GC(\log^2 n, \Pi_{k+1}^B)$.

Suppose that the language L_{k+1} is also in the class $GC(\log^2 n, \Pi_k^B)$. Then there is a Π_k^B-ATM M such that for every instance x of L_{k+1}, $x \in L_{k+1}$ if and only if there is a binary string y of length $c\log^2 n$ such that $\langle x, y\rangle$ is accepted by M, where c is a fixed constant. By Theorem 2.2, there is a Π_k^{poly,B_d}-family $F = \{\alpha_m \mid m \geq 1\}$ of circuits that accepts the same language as M. Now we can construct a circuit family $F' = \{\gamma_n \mid n \geq 1\}$ to accept L_{k+1} as follows. Given an instance x of L_{k+1}, $|x| = n$, let $m = n + c\log^2 n$. Note that by definition, $x \in L_{k+1}$ if and only if $\langle x, y\rangle$ is accepted by the circuit α_m, for a binary string y of length $c\log^2 n$. For each binary string y of length $c\log^2 n$, we construct a circuit $\alpha_m(y)$ that is the circuit α_m with the last $c\log^2 n$ input variables assigned by the value of y. Now let γ_n be the OR of all these $2^{c\log^2 n} = n^{c\log n}$ circuits $\alpha_m(y)$, $y \in \{0,1\}^{c\log^2 n}$. Then the Σ-circuit γ_n has size $O(n^{O(\log n)})$ and depth $k+2$ in which the fan-in of the level-1 gates is bounded by $O(\log n)$ (recall that the circuit family $F = \{\alpha_m | m \geq 1\}$ is a Π_k^{poly,B_d}-family of circuits), and γ_n accepts x if and only if $x \in L_{k+1}$. However, this contradicts the definition of the language L_{k+1}.

This completes the proof that the language L_{k+1} is in the class $GC(\log^2 n, \Pi_{k+1}^B)$ but not in the class $GC(\log^2 n, \Pi_k^B)$. □

Papadimitriou and Yannakakis [21] have shown that the following three problems are in *LOGSNP*.

> LOG^{2}SAT: "Given a CNF Boolean formula with n clauses and $\log^2 n$ variables, does it have a satisfying truth assignment?"
>
> LOG CLIQUE: "Given a graph with n vertices, does it have a clique of size $\log n$?"
>
> LOG CHORDLESS PATH: "Given a graph with n vertices, does it have a chordless path of length $\log n$?"

It was asked in [21] whether these three problems are complete for the class *LOGSNP*. Based on our characterization, we show that it is unlikely that these problems are complete for the class *LOGSNP*.

The problem LOG CLIQUE is actually in the class $GC(\log^2 n, \Pi_1^B)$ if on input $\langle x, y\rangle$, we allow each computation path of a Π_1^B-ATM to read two ℓ-blocks from the second parameter y. In fact, let G be a graph and let y encode $\log n$ vertices of G; then on input $\langle G, y\rangle$, a Π_1^B-ATM can universally pick a pair (v, w) of vertices from the string y and check with G if there is an edge connecting them. Similarly, the problem LOG CHORDLESS PATH can be shown to be in the class $GC(\log^2 n, \Pi_1^B)$. Note that in the proof of Theorem 5.3, the fact that the last phase of each computation path of a Π_k^B-ATM reads at most one ℓ-block from y was actually not used. Thus even though

we allow each computation path of a Π_1^B-ATM to read two ℓ-blocks from y, the class $GC(\log^2 n, \Pi_1^B)$ is still a proper subclass of the class $GC(\log^2 n, \Pi_2^B)$. Therefore, the problems LOG CLIQUE and LOG CHORDLESS PATH seem not hard enough to be complete for the class *LOGSNP*. Similarly, the problem LOG^{2}SAT can be shown to belong to the class $GC(\log^2 n, \Pi_2^{B*})$, where Π_2^{B*} is the class of languages accepted by restricted Π_2^B-ATMs in which the last phase of each computation path runs at most $O(\log \log n)$ steps and reads at most constant number of input bits. A proof similar to that of Theorem 5.3 can be derived to show that $GC(\log^2 n, \Pi_2^{B*})$ is a proper subclass of $GC(\log^2 n, \Pi_2^B)$. Therefore, the problem LOG^{2}SAT also seems not hard enough to be complete for the class $GC(\log^2 n, \Pi_2^B)$ (for detailed discussions on the problem LOG^{2}SAT, see [6]).

Meggido and Vishkin [20] studied the problem TOURNAMENT DOMINATING SET and proved that the problem LOG^{2}SAT is polynomial-time reduced to TOURNAMENT DOMINATING SET and that TOURNAMENT DOMINATING SET is polynomial-time reduced to a generalization of the LOG^{2}SAT problem. They asked whether there is a version of the satisfiability problem that precisely characterizes the problem TOURNAMENT DOMINATING SET. Our Theorem 5.1 concludes that the problem TOURNAMENT DOMINATING SET is polynomial-time equivalent to the problem BWCS$(\log n, 2)$, which is the standard satisfiability problem with a weight restriction on the truth assignment and the restriction that in each clause of the CNF formula there are at most two negative literals.

We point out that one can also derive from a result in [21] that the problem SPARSE SAT ("Given a CNF Boolean formula of n variables, does it have a satisfying truth assignment of weight at most $\log n$?") is also polynomial-time equivalent to TOURNAMENT DOMINATING SET [26]. However, the language BWCS$(\log n, 2)$ characterizes the problem TOURNAMENT DOMINATING SET more precisely in the sense that both BWCS$(\log n, 2)$ and TOURNAMENT DOMINATING SET belong to the class LOGSNP$_0 \cap GC(\log^2 n, \Pi_2^B)$ under standard encodings, while SPARSE SAT does not seem to belong to either LOGSNP$_0$ or $GC(\log^2 n, \Pi_2^B)$ under standard encodings.

Finally, we point out that the *GC* model also has nice applications in the study of fixed-parameter tractability of optimization problems [15]. We refer interested readers to our related work [9, 10].

Acknowledgments. The authors thank Rod Downey, Mike Fellows, Don Friesen, Judy Goldsmith, Christos Papadimitriou, Robert Szelepcsényi, Marty Wolf, Mihalis Yannakakis, and Chee Yap for their comments and constructive discussions. The authors also thank the referees for their careful reading and many useful comments, which have greatly improved the presentation.

REFERENCES

[1] A. Aggarwal, A. K. Chandra, and M. Snir, *Hierarchical memory with block transfer*, in Proc. 28th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 204–216.

[2] D. A. Barrington, N. Immerman, and H. Straubing, *On uniformity within NC^1*, J. Comput. System Sci., 41 (1990), pp. 274–306.

[3] R. B. Boppana and M. Sipser, *The complexity of finite functions*, in Handbook of Theoretical Computer Science, Vol. A, J. van Leeuwen, ed., MIT Press, Cambridge, MA, 1990, pp. 757–804.

[4] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

[5] J. F. Buss and J. Goldsmith, *Nondeterminism within P*, SIAM J. Comput., 22 (1993), pp. 560–572.
[6] L. Cai, *Nondeterminism and optimization*, Ph.D. thesis, Department of Computer Science, Texas A&M University, College Station, TX, 1994.
[7] L. Cai and J. Chen, *On input read-modes of alternating Turing machines*, Theoret. Comput. Sci., 148 (1995), pp. 33–55.
[8] L. Cai and J. Chen, *On the amount of nondeterminism and the power of verifying*, in Proc. 18th International Symposium on Mathematical Foundations of Computer Science (MFCS '93), Lecture Notes in Comput. Sci. 711, Springer-Verlag, Berlin, 1993, pp. 311–320.
[9] L. Cai and J. Chen, *Fixed parameter tractability and approximability of NP-hard optimization problems*, in Proc. 2nd Israel Symposium on Theory and Computing Systems, Springer-Verlag, Berlin, 1993, pp. 118–126.
[10] L. Cai and J. Chen, R. Downey, and M. Fellows, *On the structure of parameterized problems in NP*, Inform. and Comput., 123 (1995), pp. 38–49.
[11] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.
[12] J. Chen, *Characterizing parallel hierarchies by reducibilities*, Inform. Process. Lett., 39 (1991), pp. 303–307.
[13] S. A. Cook and H. J. Hoover, *A depth-universal circuit*, SIAM J. Comput., 14 (1985), pp. 833–839.
[14] J. Díaz, and J. Torán, *Classes of bounded nondeterminism*, Math. System Theory, 23 (1990), pp. 21–32.
[15] R. G. Downey and M. R. Fellows, *Fixed-parameter intractability*, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 36–49.
[16] G. Farr, *On problems with short certificates*, Acta Inform., 31 (1994), pp. 479–502.
[17] M. Fischer and N. J. Pippenger, *Relations among complexity measures*, J. Assoc. Comput. Mach, 26 (1979), pp. 361–381.
[18] D. S. Johnson, *The NP-completeness column: An ongoing guide*, J. Algorithms, 13 (1992), pp. 502–524.
[19] C. Kintala and P. Fisher, *Refining nondeterminism in relativized complexity classes*, SIAM J. Comput., 13 (1984), pp. 329–337.
[20] N. Meggido and U. Vishkin, *On finding a minimum dominating set in a tournament*, Theoret. Comput. Sci. 61 (1988), pp. 307–316.
[21] C. H. Papadimitriou and M. Yannakakis, *On limited nondeterminism and the complexity of the V-C dimension*, in Proc. 8th Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 12–18.
[22] N. J. Pippenger, *Fast simulation of combinational logic networks by machines without random-access storage*, in Proc. 15th Allerton Conference on Communication, Control, and Computing, 1977, pp. 25–33.
[23] W. L. Ruzzo, *On uniform circuit complexity*, J. Comput. System Sci., 21 (1981), pp. 365–383.
[24] R. Szelepcsényi, *β_k-complete problems and greediness*, Technical Report 455, Computer Science Department, University of Rochester, Rochester, NY, 1993.
[25] M. J. Wolf, *Nondeterministic circuits, space complexity, and quasigroups*, Theoret. Comput. Sci., 125 (1994), pp. 295–313.
[26] M. Yannakakis, private communications, 1993.
[27] A. Yao, *Separating the polynomial-time hierarchy by oracles*, in Proc. 26th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 1–10.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 751–763, June 1997

009

BOUNDS ON THE NUMBER OF EXAMPLES NEEDED FOR LEARNING FUNCTIONS*

HANS ULRICH SIMON†

Abstract. We prove general lower bounds on the number of examples needed for learning function classes within different natural learning models which are related to pac-learning (and coincide with the pac-learning model of Valiant in the case of $\{0,1\}$-valued functions). The lower bounds are obtained by showing that all nontrivial function classes contain a "hard binary-valued subproblem." Although (at first glance) it seems to be likely that real-valued function classes are much harder to learn than their hardest binary-valued subproblem, we show that these general lower bounds cannot be improved by more than a logarithmic factor. This is done by discussing some natural function classes like nondecreasing functions or piecewise-smooth functions (the function classes that were discussed in [M. J. Kearns and R. E. Schapire, *Proc. 31st Annual Symposium on the Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 382–392, full version, J. Comput. System Sci., 48 (1994), pp. 464–497], [D. Kimber and P. M. Long, *Proc. 5th Annual Workshop on Computational Learning Theory*, ACM, New York, 1992, pp. 153–160]) with certain restrictions concerning their slope.

Key words. function learning, sample complexity

AMS subject classification. 68T05

PII. S0097539793259185

1. Introduction.

The question of how many examples are necessary and sufficient for learning has found a very satisfactory answer within the distribution-free learning model for deterministic concept classes C (the pac-learning model of Valiant [12]). There is a lower bound of $\Omega((d(C)+\ln(1/\delta))/\epsilon)$ (see [3] and [4]) and an upper bound of $O((d(C)\ln(1/\epsilon)+\ln(1/\delta))/\epsilon)$ (see [3]), where $d(C)$ denotes the Vapnik–Chervonenkis dimension of C. Since these bounds are within a logarithmic factor of each other, it is fairly well known how the sample complexity scales with the parameters ϵ and δ (which specify the demanded accuracy and confidence of learning).

The distribution-free learning model has been transferred to probabilistic concept classes (also called p-concept classes) by Kearns and Schapire (see [6]) and to function classes by Haussler (see [5]). In this paper, we restrict ourselves to functions with real values in the range $[0,1]$ (hereafter simply called functions).[1] The main difference between p-concept learning and function learning lies in the kind of feedback. A labeled example for a function f has the form $(x,f(x))$, i.e., the learning algorithm obtains the correct value of f at point x. For p-concept learning, the feedback is weaker. Here $f(x)$ is interpreted as a probability (that x fits into the concept). A labeled example has the form $(x,l(x))$, where $l(x)\in\{0,1\}$ is the outcome of a coin which produces 1 with probability $f(x)$ and 0 with probability $1-f(x)$. Note that learning f as a function is easier than learning f as a p-concept since the feedback is much stronger.

* Received by the editors November 16, 1993; accepted for publication (in revised form) July 12, 1995. This research was supported by Bundesministerium für Forschung und Technologie grant 01IN102C/2.

http://www.siam.org/journals/sicomp/26-3/25918.html

† Fachbereich Informatik, Universität Dortmund, D-44221 Dortmund, Germany (simon@ls2.informatik.uni-dortmund.de).

[1] It will become evident, however, that all results are easily extended to functions with bounded range.

This paper will demonstrate that the sample complexity of function learning can be characterized very well in terms of a quantity $d_F(\gamma)$, which is related to the Natarajan dimension of F (see [9]), but depends on a given width γ of shattering.[2] We first derive some general lower bounds. They are obtained by showing that all nontrivial function classes contain a "hard binary-valued subproblem." Although (at first glance) it seems to be likely that real-valued function classes are much harder to learn than their hardest binary-valued subproblem, we show that these general lower bounds cannot be improved by more than a logarithmic factor. This is done by discussing some natural function classes like nondecreasing functions or piecewise-smooth functions (the function classes that were discussed in [6] and [7]) with certain restrictions concerning their slope.[3]

Although our general lower bounds cannot be substantially improved (because there are "almost matching" upper bounds for some specific function classes), we are not aware of general upper bounds which match the lower ones modulo a logarithmic factor. Recently, Anthony and Shawe-Taylor derived general upper bounds for function learning in terms of several other notions of dimension associated with F and γ (see [2]). Their results are incomparable to ours because the variation within the different notions of dimension is too large. In particular, their general upper bounds cannot be directly applied to the specific function classes considered in this paper (the corresponding dimensions are not finite). Thus closing the gap between the general upper and lower bounds is a major object of future research.

This paper is structured as follows. Section 2 presents the definition of the learning models. Section 3 presents the general lower bounds. Section 4 presents "almost matching" upper bounds for some natural function classes. Section 5 presents generalizations to higher-dimensional domains. Section 6 discusses some open problems.

2. The learning models. Let F be a class of functions from a domain X into $[0,1]$. The environment for learning consists of a hidden target function $f \in F$ and an unknown distribution D on X.[4] A *random sample I of size m* w.r.t. D, f consists of m sample points $x_1, \ldots, x_m$ drawn randomly according to D^m and the corresponding values $f(x_1), \ldots, f(x_m)$ of the target function. A function $h : X \to [0,1]$ is called an *(ϵ, γ)-good model for f* if

$$D(\{x|\ |f(x) - h(x)| \geq \gamma\}) < \epsilon.$$

The *expected absolute difference* and the *expected quadratic difference* of h (w.r.t. D, f) are given by

$$e_1(h) = E_D[|f(x) - h(x)|] \quad \text{and} \quad e_2(h) = E_D[(f(x) - h(x))^2],$$

respectively.

A *learning algorithm* A for F with sample size $m = m(\epsilon, \gamma, \delta)$ gets as input the parameters ϵ, γ, and δ and a random sample of size m w.r.t. D and f. It outputs a hypothesis $h_I : X \to [0,1]$. Note that the output h_I depends on the randomly chosen

[2] In a similar way, Kearns and Schapire defined a variant of the combinatorial dimension which depends on a width γ of shattering (see [6]).

[3] The reader interested in related work for p-concept learning is referred to [11].

[4] Formally, D is a probability measure for $(X, \mathcal{A})$, where $\mathcal{A}$ denotes a σ-algebra of X. We assume that all functions from F and all hypotheses h used are measurable functions from $(X, \mathcal{A})$ into $([0,1], \mathcal{B})$, where $\mathcal{B}$ denotes the standard algebra of Borel sets in $[0,1]$. This assumption will guarantee that throughout the paper all probabilities or expectations are well defined.

sample I. It is therefore meaningful to speak about the probability (w.r.t. D^m) that h_I has a particular property. For some of the learning models (to be defined below), parameter γ is missing.

We say that *A learns F with an (ϵ,γ)-good model* if for all $f \in F$, all domain distributions D, all $0 < \epsilon, \gamma, \delta < 1$, and $m = m(\epsilon,\gamma,\delta)$, there is a probability of at least $1-\delta$ (w.r.t. D^m) that h_I is an (ϵ,γ)-good model for f.

Similarly, we say that *A learns F with ϵ-bounded absolute (or ϵ-bounded quadratic) difference* if for all $f \in F$, all domain distributions D, all $0 < \epsilon, \delta < 1$, and $m = m(\epsilon,\delta)$, there is a probability of at least $1-\delta$ (w.r.t. D^m) that h_I satisfies $e_1(h) < \epsilon$ (or $e_2(h) < \epsilon$, respectively).

3. General lower bounds. We begin this section with the definition of the function $d_F(\gamma)$ associated with a function class F and a given width γ of shattering. A sequence

$$S = ((x_1, r_1, s_1), \ldots, (x_d, r_d, s_d)) \in (X \times [0,1] \times [0,1])^d$$

is called γ-shattered by F if the following hold:

1. $(\forall b \in \{0,1\}^d) \quad (\exists f = f_b \in F) \quad (\forall i = 1, \ldots, d) \quad f(x_i) = \begin{cases} r_i & \text{if } b_i = 0, \\ s_i & \text{if } b_i = 1. \end{cases}$
2. $(\forall i = 1, \ldots, d)\ s_i \geq 2\gamma + r_i$.

In other words, for each point x_i there exists a low function value r_i and a high function value s_i. Function class F is sufficiently rich to allow all combinations of low and high values, and the high value exceeds the low one by at least 2γ. We define $d_F(\gamma)$ as the maximal number d (possibly ∞) such that there exists a sequence of length d which is γ-shattered by F.

We would like to make some informal comments on the motivation behind this definition. Without demanding that $s_i \geq 2\gamma + r_i$, we would obtain the definition of the so-called Natarajan dimension $d_N(F)$ (defined in [9], where it is called the generalized dimension). However, the existence of low and high function values (and free combinations there of) is not really "dangerous" for learning algorithms A. If r_i and s_i are very close to each other, A may make a trivial guess (such as $(r_i + s_i)/2$, for instance) and still approximate the true function value quite accurately. For this reason, in [6], Kearns and Schapire proposed to incorporate the width γ of shattering into the definition. Their results show that $d_F(\gamma)$ is a lower bound for p-concept learning.[5] In this section, we strengthen this result in two directions. First, we derive a higher lower bound. Second, we show that the lower bound is even valid w.r.t. function learning. (Recall that the feedback for p-concept learning is weaker; lower bounds for function learning are therefore always lower bounds for p-concept learning, but not vice versa.)[6]

Consider, for example, the class *ND* of nondecreasing functions from the real domain $\Re$ into $[0,1]$ (discussed as a p-concept class in [6]). It is not hard to show that

$$((0,0,2\gamma), (2\gamma, 2\gamma, 4\gamma), (4\gamma, 4\gamma, 6\gamma), \ldots)$$

is a γ-shattered sequence of maximal length. Thus

$$d_{ND}(\gamma) = \lfloor 1/(2\gamma) \rfloor.$$

[5] They used a slightly different definition of $d_F(\gamma)$. Both definitions coincide for the function classes considered in this paper.

[6] The reader interested in almost optimal lower bounds for p-concept learning is referred to [11].

As a second example, consider the class $\mathcal{F}_1$ of continuous functions from $[0,1]$ into $[0,1]$ which are piecewise continuously differentiable and satisfy

$$\int_0^1 |f'(x)|dx \leq 1$$

(i.e., the average slope of f is bounded by 1). A more restricted class is $\mathcal{F}_\infty$, which consists of all continuous functions from $[0,1]$ into $[0,1]$ which are piecewise continuously differentiable and whose slope is bounded by 1 at each $x \in [0,1]$. (For points at which two pieces are stuck together, there are "two slopes" depending on whether we differentiate from the left or from the right; for functions $f \in \mathcal{F}_\infty$, "both slopes" must then be bounded by 1.) It is obvious that the sequence

$$((0,0,2\gamma),(2\gamma,0,2\gamma),(4\gamma,0,2\gamma),\ldots)$$

is γ-shattered by $\mathcal{F}_1$ and $\mathcal{F}_\infty$. Thus

$$d_{\mathcal{F}_1}(\gamma) \geq d_{\mathcal{F}_\infty}(\gamma) \geq 1 + \lfloor 1/(2\gamma) \rfloor.$$

In section 4, we show that these inequalities can be turned into equalities. In [7], Kimber and Long discuss a hierarchy of function classes which starts in $\mathcal{F}_\infty$ and ends in $\mathcal{F}_1$. They were interested in (different variants of) on-line learning of function classes. Their results are incomparable to ours. (For instance, $\mathcal{F}_1$ is not learnable in the on-line model, but it is learnable in our model, as we will show in section 4.) However, it is worth mentioning that we will obtain (see section 4) an upper bound on the sample complexity of $\mathcal{F}_1$ (the most general class of the hierarchy) which is tight to within a logarithmic factor to a lower bound on the sample complexity of $\mathcal{F}_\infty$ (the least general class of the hierarchy). Our bounds are therefore "almost tight" for the whole hierarchy considered by Kimber and Long.

We are now ready to derive our general lower bounds. We say that two functions f_1 and f_2 on domain X are *disjoint* if $f_1(x) \neq f_2(x)$ for all $x \in X$. A function class F is called *trivial* if its functions are pairwise disjoint. Note that a single example is sufficient for learning a trivial function class. (Each labeled example reveals the identity of the function.) Our attention will therefore be focused on nontrivial classes. The quantity $\Delta(F)$ is defined as follows:

$$\Delta(F) = \sup\{|g(x) - f(x)|, \text{ where } x \in X \text{ and } f \text{ and } g \text{ are nondisjoint functions from } F\}.$$

It measures how much two nondisjoint functions may differ on some point of domain X. For instance, $\Delta(F) = 1$ for $F = ND, \mathcal{F}_\infty, \mathcal{F}_1$. This is witnessed by the identity and the constant-zero function on $[0,1]$ in the obvious way.

THEOREM 3.1. *Let A be an algorithm which learns function class F with an (ϵ,γ)-good model.*

1. *If F is nontrivial, $\epsilon < 1/2$, and $\gamma < \Delta(F)/2$, then A needs $\Omega(\ln(1/\delta)/\epsilon)$ examples.* 2. *If $0 < \epsilon \leq 1/8$, $0 < \delta \leq 1/100$, then A needs $\Omega((d_F(\gamma) - 1)/\epsilon)$ examples.*

Proof. 1. Since F is nontrivial, there exist two nondisjoint functions $f, g \in F$ and two points $a, b \in X$ such that

$$f(a) = g(a) \quad \text{and} \quad f(b) + 2\gamma \leq g(b).$$

W.l.o.g., $X = \{a, b\}$ and $F = \{f, g\}$. Let $\bar{f}$ and $\bar{g}$ be given by

$$\bar{f}(a) = \bar{g}(a) = 0, \qquad \bar{f}(b) = 0, \quad \text{and} \quad \bar{g}(b) = 1,$$

and $\bar{F} = \{\bar{f}, \bar{g}\}$. $\bar{F}$ is a nontrivial deterministic concept class. It is well known that $\Omega(\ln(1/\delta)/\epsilon)$ examples are needed for pac-learning $\bar{F}$ under domain distribution $D(a) = 1 - \epsilon,\ D(b) = \epsilon$ (see [3]). The assertion of our theorem is now easily obtained because algorithm A can be converted into a pac-learning algorithm $\bar{A}$ for $\bar{F}$ as follows.

Let $\bar{t} \in \bar{F}$ denote the target concept. A sample for $\bar{t}$ can be converted into a sample for t by substituting $(a, f(a))$ for $(a, 0)$, $(b, f(b))$ for $(b, 0)$, and $(b, g(b))$ for $(b, 1)$. Algorithm A then runs on the converted sample and produces (with high confidence) a hypothesis h which is an (ϵ, γ)-good model for $t \in \{f, g\}$. W.l.o.g., $h(a) = f(a)$ because both possible target functions attain the same value at a. Since f and g differ by at least 2γ at point b, $h(b)$ differs from $f(b)$ or $g(b)$ by at least γ. It follows w.l.o.g. that $h(b) \in \{f(b), g(b)\}$. Thus $h \in F$. Finally, observe that $\bar{h}$ is an ϵ-good hypothesis for $\bar{t}$ w.r.t. pac-learning if h is an (ϵ, γ)-good model for t.

2. Let $d = d_F(\gamma)$ and $S = ((x_1, r_1, s_1), \ldots, (x_d, r_d, s_d))$, the γ-shattered sequence. W.l.o.g., $X = \{x_1, \ldots, x_d\}$ and F consists of the 2^d shattering concepts f_b for S ($b \in \{0, 1\}^d$). We associate with each f_b the deterministic concept $\bar{f}_b$ given by $\bar{f}_b(x_i) = b_i$. $\bar{F}$ denotes the corresponding deterministic concept class. Note that S is shattered by $\bar{F}$ (in the traditional sense) and d coincides with the Vapnik–Chervonenkis dimension of $\bar{F}$. It is well known that $\Omega((d-1)/\epsilon)$ examples are needed for pac-learning $\bar{F}$ under domain distribution $D(x_1) = 1 - 8\epsilon$, and $D(x_i) = 8\epsilon/(d-1)$ for $i = 2, \ldots, d$ (see [4]). The assertion of the theorem is now easily obtained because A can be converted into a pac-learning algorithm $\bar{A}'$ for $\bar{F}$. The design for $\bar{A}'$ is similar to the aforementioned design of $\bar{A}$. Now the labels r_i and s_i correspond to the labels 0 and 1, respectively. We omit the details. □

COROLLARY 3.2. *Any algorithm which learns one of the function classes ND, $\mathcal{F}_\infty$, $\mathcal{F}_1$ with an (ϵ, γ)-good model needs at least $\Omega(1/(\epsilon\gamma))$ examples.*

We shall see in section 4 that this bound is tight to within a logarithmic factor. To keep the assertion of the following corollary simple, we assume that $d_F(\gamma)$ is polynomially bounded in $1/\gamma$ and that ϵ and δ are sufficiently small.

COROLLARY 3.3. 1. *Let F be a nontrivial function class. Any algorithm which learns F with ϵ-bounded absolute (or quadratic) difference needs at least $\Omega(\ln(1/\delta)\Delta(F)/\epsilon)$ (or $\Omega(\ln(1/\delta)\Delta^2(F)/\epsilon)$) examples.*

2. *Any algorithm which learns F with ϵ-bounded absolute (or quadratic) difference needs at least $\Omega(d_F(\epsilon))$ (or $\Omega(d_F(\sqrt{\epsilon}))$) examples.*

Proof. The assertions follow immediately from Theorem 3.1 and the following facts from [6]:

Let $0 < c \leq 1$. If $e_1(h) < \epsilon$, then h is an $(\epsilon/c, c)$-good model. If $e_2(h) < \epsilon$, then h is an $(\epsilon/c^2, c)$-good model.

Assertion 1 is obtained by setting $c < \Delta(F)/2$ (for instance, $c = \Delta(F)/3$). Assertion 2 is obtained by setting $c = 8\epsilon$ or $\sqrt{8\epsilon}$, respectively. □

COROLLARY 3.4. *Any algorithm which learns one of the function classes ND, $\mathcal{F}_\infty$, $\mathcal{F}_1$ with an ϵ-bounded absolute (or quadratic) difference needs at least $\Omega(\ln(1/\delta)/\epsilon)$ examples.*

We shall see in section 4 that this bound is tight to within a logarithmic factor.

4. Upper bounds. In this section, we define the class BV of functions with bounded variation, which contains ND and $\mathcal{F}_1$ as subclasses. We describe learning algorithms for BV whose consumption of examples meets the lower bounds of section 3

within a logarithmic factor. (Note that the upper bounds hold for the most general class BV and the "almost matching" lower bounds hold already for subclasses.)

We say that a function f on domain $X \subseteq \Re$ has *bounded variation* if

$$\sum_{i=1}^{r-1} |f(x_{i+1}) - f(x_i)| \leq 1$$

for all $r \geq 1$ and all sequences $x_1 < \cdots < x_r$ of numbers from X. BV denotes the class of functions with bounded variation.[7]

LEMMA 4.1. *$ND, \mathcal{F}_1 \subseteq BV$.*

Proof. Nondecreasing functions from $\Re$ to $[0,1]$ satisfy

$$\sum_{i=1}^{r-1} |f(x_{i+1}) - f(x_i)| = \sum_{i=1}^{r-1} f(x_{i+1}) - f(x_i) = f(x_r) - f(x_1) \leq 1 - 0 = 1.$$

Thus $ND \subseteq BV$. Now let $f \in \mathcal{F}_1$ and define

$$t_i = \int_{x_i}^{x_{i+1}} |f'(x)| dx.$$

It follows that $\sum_{i=1}^{r-1} t_i \leq 1$. If f is monotonic in $[x_i, x_{i+1}]$, then

$$t_i = \int_{x_i}^{x_{i+1}} |f'(x)| dx = \left| \int_{x_i}^{x_{i+1}} f'(x) dx \right| = |f(x_{i+1)} - f(x_i)|.$$

A simple shortcut construction (illustrated in Figure 1) shows that a function f which is not monotonic in some interval $[a,b]$ can be transformed into a function which is monotonic, takes the same values at points a and b, and has an average slope which is not higher than the original one. This shows that, in general, $t_i \geq |f(x_{i+1}) - f(x_i)|$. Thus f has bounded variation. Therefore, $\mathcal{F}_1 \subseteq BV$. □

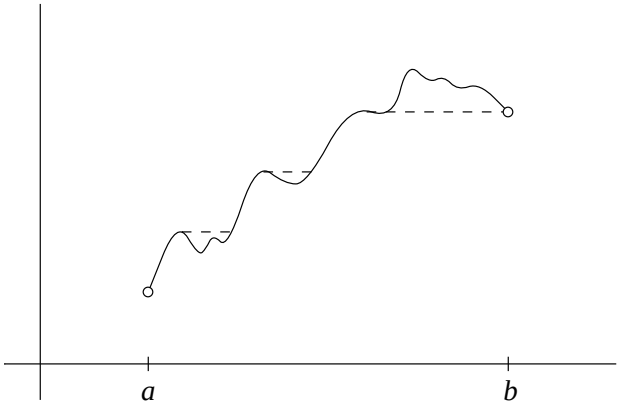


FIG. 1. *Making nonmonotonic functions monotonic by shortcuts (drawn dotted).*

The following lemma is useful for the analysis of functions with bounded variation.

[7] All the following results are easily extended to the class $BV[B]$, where we allow an arbitrary bound B on the variation. This class contains functions whose average slope is bounded by B, or which consist of B monotonic segments.

LEMMA 4.2. *Let $d \geq 2$. Let $0 \leq r_i < s_i \leq 1$ and $\lambda_i = s_i - r_i$ for $i = 1, \ldots, d$. Then it is possible to choose $t_i \in \{r_i, s_i\}$ such that*

$$\sum_{i=1}^{d-1} |t_{i+1} - t_i| \geq \frac{\lambda_1 + \lambda_d}{2} + \sum_{i=2}^{d-1} \lambda_i.$$

Proof. Choose the sequences $T = (t_i)$ and $T' = (t'_i)$ as

$$T = r_1, s_2, r_3, s_4, \ldots \quad \text{and} \quad T' = s_1, r_2, s_3, r_4, \ldots.$$

As illustrated in Figure 2, the following condition holds:

$$\sum_{i=1}^{d-1} (|t_{i+1} - t_i| + |t'_{i+1} - t'_i|) \geq \sum_{i=1}^{d-1} (\lambda_i + \lambda_{i+1}).$$

Thus T or T' satisfies the assertion of the lemma. □

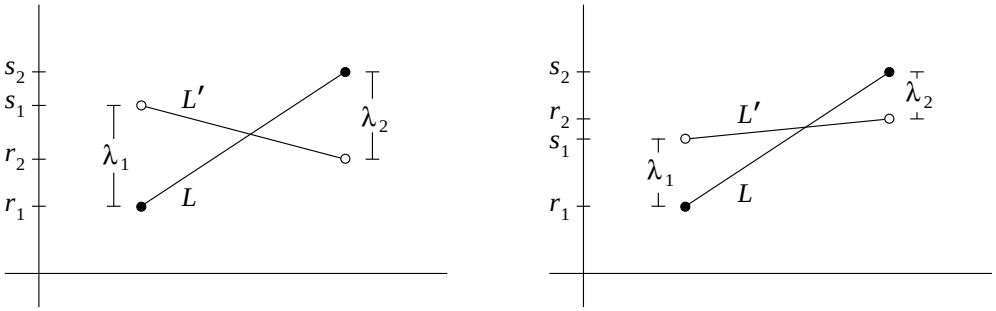


FIG. 2. *Two dual line segments L and L'. Their combined height difference $|s_2 - r_1| + |s_1 - r_2|$ is at least $\lambda_1 + \lambda_2$.*

COROLLARY 4.3. $d_{BV}(\gamma) = d_{\mathcal{F}_1}(\gamma) = d_{\mathcal{F}_\infty}(\gamma) = 1 + \lfloor 1/(2\gamma) \rfloor$.

Proof. Since $\mathcal{F}_\infty \subseteq \mathcal{F}_1 \subseteq BV$ and $d_{\mathcal{F}_\infty}(\gamma) \geq 1 + \lfloor 1/(2\gamma) \rfloor$, it suffices to show that $d = d_{BV}(\gamma) \leq 1 + \lfloor 1/(2\gamma) \rfloor$. Assume that $S = (x_i, r_i, s_i)_{1 \leq i \leq d}$ is γ-shattered by BV. Remember that $s_i \geq r_i + 2\gamma$. Choose the sequence $t_i \in \{r_i, s_i\}$ according to Lemma 4.2. Choose $b \in \{0,1\}^d$ such that the shattering function $f = f_b$ attains value t_i at point x_i. It follows from Lemma 4.2 and the definition of BV that the following holds:

$$(d-1)2\gamma \leq \sum_{i=1}^{d-1} |f(x_{i+1}) - f(x_i)| \leq 1.$$

Thus $d \leq 1 + 1/(2\gamma)$. Since d is an integer, the assertion of the corollary follows. □

The following two theorems are the main results of this section.

THEOREM 4.4. $O((1/\epsilon) \cdot ((1/\gamma)\ln(1/\epsilon) + \ln(1/\delta)))$ *examples are sufficient for learning BV with an (ϵ, γ)-good model.*

Proof. We first describe a learning function which transforms a labeled sample I of target function f into a hypothesis h. Let (after sorting of the sample points)

$$I = (x_1, \alpha_1), \ldots, (x_m, \alpha_m), \quad x_1 \leq x_2 \leq \cdots \leq x_m.$$

It will be technically convenient to define $x_0 = \inf X, x_{m+1} = \sup X$. For all $x \in X$, let $i(x) = j$ if and only if x_j is the sample point closest to x (breaking ties in favor of smaller indices). We come up with the hypothesis $h(x) = \alpha_{i(x)}$. Note that h is consistent with the sample. The critical question is how well h models f within the open intervals (x_j, x_{j+1}).

We now turn to the statistical analysis of this learning function. Let D denote the unknown domain distribution. The *variation* in interval $I_j = [x_j, x_{j+1}] \cap X$ is defined as

$$v_j = \sup_{a,b \in I_j} |f(b) - f(a)|.$$

Since $f \in BV$, the sum of all variations is bounded by 1. The open interval $I'_j = (x_j, x_{j+1})$ is said to be *wild* if $v_j \geq \gamma$. Otherwise, it is said to be *tame*. Within tame intervals, our hypothesis is certainly γ-close to f. Thus all that remains to do is to bound (with high confidence) the total probability of hitting a wild interval by ϵ. Since the variations sum up to at most one, there are at most $1/\gamma$ wild intervals. The "dangerous" region U of the real line is therefore always a union of at most $1/\gamma$ open intervals. The class $\mathcal{U}$ of all regions of this form has Vapnik–Chervonenkis dimension $2\lfloor 1/\gamma \rfloor$. Note that region U does not contain any point from random sample I. Applying a uniform convergence theorem from [3] (with improved constants from [1] or [10]), it follows that the following holds with a confidence of at least $1 - \delta$ for all sample sizes

$$m \geq \frac{4\lfloor 1/\gamma \rfloor \ln(6/\epsilon) + \ln(2/\delta)}{\epsilon(1 - \sqrt{\epsilon})}$$

and all $U \in \mathcal{U}$: if U does not contain any point from I, then $D(U) < \epsilon$. From this the theorem follows. □

THEOREM 4.5. *$O\left(\ln(1/(\epsilon\delta))/\epsilon\right)$ examples are sufficient for learning ND with an ϵ-bounded absolute or quadratic difference.*

Proof. Since $e_2(h) \leq e_1(h)$, it suffices to show the assertion w.r.t. ϵ-bounded absolute difference. The hypothesis h is constructed from the sample I in the same way as in the proof of Theorem 4.4. Also, the notations from this proof are reused. It follows easily that the expected absolute difference of h satisfies

$$e_1(h) \leq \sum_{j=0}^{m} v_j \cdot D(I'_j) \leq \max_{0 \leq j \leq m} D(I'_j)$$

because the variations sum up to at most 1. It suffices therefore to bound (with high confidence) the maximal probability of hitting an open interval I'_j by ϵ. Using the same uniform convergence theorem as in the proof of Theorem 4.4 (now applied to the system of open intervals, which has Vapnik–Chervonenkis dimension 2), we conclude that the following holds with a confidence of at least $1 - \delta$ for all sample sizes

$$m \geq \frac{4\ln(6/\epsilon) + \ln(2/\delta)}{\epsilon(1 - \sqrt{\epsilon})}$$

and all open intervals U: if U does not contain any point from I, then $D(U) < \epsilon$. From this the theorem follows. □

5. Higher-dimensional domains. In this section, we investigate the learnability of classes of functions with bounded variation which depend on several variables. We first discuss the two-dimensional case. Later, we briefly mention the generalization to more dimensions.

We call the sequence

$$(x_1, y_1), \ldots, (x_r, y_r) \in \Re \times \Re$$

an *ascending* (or *descending*) *chain* if $x_1 \leq \cdots \leq x_r$ and $y_1 \leq \cdots \leq y_r$ (or $y_1 \geq \cdots \geq y_r$, respectively). We say that a function f on a domain $X = X_1 \times X_2 \subseteq \Re^2$ has *bounded variation* if condition

$$\sum_{i=1}^{r-1} |f(x_{i+1}, y_{i+1}) - f(x_i, y_i)| \leq 1$$

is satisfied for all (ascending or descending) chains. We say that f has *semibounded variation* if this condition must hold only for ascending chains. The corresponding function classes are denoted by $BV(2)$ and $SBV(2)$, respectively. Note that both classes collapse to BV for functions which only depend on one of the two variables.

A *product distribution* on $\Re \times \Re$ is given by two independent distributions on $\Re$. Thus the x- and y-coordinates of a randomly drawn point $(x, y) \in \Re \times \Re$ are independent. We want to show that $SBV(2)$ (which contains $BV(2)$) is efficiently learnable under product distributions. The learning algorithm A will use a suitable partition of domain X into $r \times r$ orthogonal cells, given by $r-1$ horizontal and $r-1$ vertical lines, respectively. We associate with each cell C its indices $i(C), j(C) \in \{1, \ldots, r\}$ in the partitioning, where indexing goes from left to right and from bottom to top. We say that a sequence $C_1, \ldots, C_r$ of cells is an *ascending chain* if

$$i(C_1) < \cdots < i(C_r) \text{ and } j(C_1) < \cdots < j(C_r).$$

The basic idea in the design of A is to draw sufficiently many random examples such that with high confidence, each cell is hit at least once. If the first sample point which hits C has label $\alpha(C)$, then A's hypothesis is set to $\alpha(C)$ for all points in C. This definition is ambiguous at the boundary of the cells. We will, however, use partitionings which avoid this ambiguity. Thus if all cells are hit at least once, A comes up with a well-defined hypothesis h. An important notion is the *variation* of target function $f \in SBV$ within a cell, defined by

$$v(C) = \sup_{z', z'' \in C} |f(z'') - f(z')|.$$

Our analysis is based on the following lemma.

LEMMA 5.1. *If $C_1, \ldots, C_r$ is an ascending chain, then $\sum_{i=1}^{r} v(C_i) \leq 2$.*

Proof. Since the variation of f on ascending chains is bounded by 1, it suffices to show that there exists an ascending chain $z_1, \ldots, z_s$ which satisfies

$$\sum_{i=1}^{s-1} |f(z_{i+1}) - f(z_i)| \geq \frac{1}{2} \cdot \sum_{i=1}^{r} (v(C_i) - \Delta)$$

for arbitrarily small $\Delta > 0$. Because $C_1, \ldots, C_r$ is an ascending chain of cells, it suffices to find an ascending pair $z, \bar{z}$ within each cell $C \in \{C_1, \ldots, C_r\}$ which satisfies

$|f(z) - f(\bar{z})| \approx v(C)/2$. The sequence of these pairs then forms an appropriate ascending chain of length $2r$. Cell C contains two points z' and z'' which satisfy $|f(z'') - f(z')| \approx v(C)$. W.l.o.g., z' is left of z''. If z' is below z'', then z', z'' is an appropriate ascending pair. If z' is to the left of and above z'', then let z be the point which shares the x-coordinate with z' and the y-coordinate with z'' (see Figure 3). Then z, z' and z, z'' are both ascending pairs of points. At least one of these pairs witnesses a variation of approximately $v(C)/2$. This completes the proof of the lemma. □

FIG. 3. *A "descending pair" z', z'' and two "ascending pairs" z, z' and z, z''.*

COROLLARY 5.2. *The variations of all r^2 cells sum up to at most $4r - 2$.*

Proof. The r^2 cells partition into $2r - 1$ diagonal ascending chains. According to Lemma 5.1, each diagonal contributes at most 2 to the total sum of variations. □

Assume that there is a constant k such that the hitting probability $D(C)$ of each cell C satisfies

$$\frac{1}{kr^2} \leq D(C) \leq \frac{k}{r^2} \quad \text{(condition of } k\text{-uniformity)}.$$

For instance, if $X = [0,1] \times [0,1]$ and D is the uniform distribution on X, a regular partitioning into r^2 cells has this property with $k = 1$. The expected absolute difference of h can then be bounded as follows:

$$e_1(h) \leq \frac{k}{r^2} \cdot \sum_C v(C) < \frac{4k}{r}.$$

Thus $e_1(h) < \epsilon$ if $r \geq 4k/\epsilon$. Standard arguments show that $m = kr^2(2\ln(r) + \ln(1/\delta))$ random examples are sufficient to hit each cell at least once with a confidence of $1 - \delta$. This shows that $O(\ln(1/(\epsilon\delta))/\epsilon^2)$ examples are sufficient given the assumption of k-uniformity. We are now ready to prove the main result of this section.

THEOREM 5.3. *$O(\ln(1/(\epsilon\delta))/\epsilon^2)$ examples are sufficient to learn $SBV(2)$ with ϵ-bounded absolute (or quadratic) difference if the domain distribution is a product distribution.*

Proof. We may assume w.l.o.g. that two different sample points never share a coordinate.[8] It is then possible to cut X into r horizontal and r vertical slices such that each (open) slice contains m/r sample points. (W.l.o.g., assume that m is a multiple of r.) The system of slices has a Vapnik–Chervonenkis dimension of 2. We may therefore apply a uniform convergence theorem from [3] (Theorem A3.1) which (when applied separately to horizontal and vertical slices with confidence parameter $\delta/4$, respectively) states the following: there exists a constant c such that for all $m \geq$

[8] For product distributions given by continuous density functions, this is almost certainly the case. For general product distributions, one can use the trick of Kearns and Schapire in [6] which guarantees the technical assumption by formally changing the domain X into a new one, where different occurrences of the same coordinate are regarded as different. Geometrically, this means that, loosely speaking, horizontal or vertical lines get (artificially) a nonzero width. We omit the details of this construction.

$cr\ln(r/\delta)$, all slices S satisfy $1/(2r) \le D(S) \le 2/r$ with a confidence of $1-\delta/2$. Since D is a product distribution, each of the resulting r^2 cells C satisfies $1/(4r^2) \le D(C) \le 4/r^2$. In other words, we can achieve k-uniformity with $k=4$. Now $O(\ln(1/(\epsilon\delta))/\epsilon^2)$ additional examples are sufficient to guarantee with a confidence of $1-\delta/2$ that each cell is hit at least once. □

Using the simple fact (see [6]) that $e_1(h) < \epsilon\gamma$ implies that h is an (ϵ,γ)-good model, we obtain the following result.

COROLLARY 5.4. *$O(\ln(1/(\epsilon\gamma\delta))/\epsilon^2\gamma^2)$ examples are sufficient to learn $SBV(2)$ with an (ϵ,γ)-good model of probability if the domain distribution is a product distribution.*

There is no chance to generalize the results about $SBV(2)$ to arbitrary domain distributions because $d_{SBV(2)}(\gamma) = \infty$ for all $0 < \gamma \le 1/2$. Consider, for instance, the functions

$$f_b(x,y) = \begin{cases} b(x,y) & \text{if } x+y=0, \\ 1/2 & \text{otherwise,} \end{cases}$$

where b is an arbitrary function from $\Re^2$ to $\{0,1\}$. Each function f_b has semibounded variation. On the other hand, these functions γ-shatter the sequence

$$((x,-x),0,1)_{x\in\Re}$$

for all $\gamma \le 1/2$. According to the lower bounds derived in section 3, $SBV(2)$ is not learnable in the distribution-free model. We leave it as an open question whether $BV(2)$ is learnable under arbitrary domain distributions. All that we know is a lower bound on the number of examples required for learning. It is obtained from the general lower bounds in section 3 and the following result.

LEMMA 5.5.

$$\left\lfloor \frac{1}{2\gamma} \right\rfloor^2 \le d_{BV(2)}(\gamma) \le \left(1+\frac{1}{2\gamma}\right)^2.$$

Proof. The lower bound for $d_{BV(2)}(\gamma)$ follows from a close inspection of Figure 4 which shows a grid consisting of $r^2 \times r^2$ points ($r=4$ in the figure). r^2 of them are drawn black; the remaining ones are drawn white. Let the function class F consist of functions which attain value γ on white points and value 0 or 2γ on black points. Obviously, the r^2 black points are γ-shattered by F. We claim that all functions from F belong to $BV(2)$ if $r = \lfloor 1/(2\gamma) \rfloor$. This can be seen as follows. Each ascending or descending chain of grid points contains at most r black points. The induced variation is therefore bounded by $2\gamma r \le 1$.

We have still to verify the upper bound. Assume that there is a sequence S of length d which is γ-shattered by $BV(2)$. According to a classical result of Erdös (reported in [8]), S contains an ascending or a descending chain of length at least $\sqrt{d}$. At least one of the shattering functions must have a total variation of $2\gamma\cdot(\sqrt{d}-1)$ on this chain. (The proof for this is similar to the proof of Corollary 4.3.) Thus $2\gamma\cdot(\sqrt{d}-1) \le 1$, and the assertion of the lemma follows. □

We denote the obvious generalization of $BV(2)$ and $SBV(2)$ to n dimensions by $BV(n)$ and $SBV(n)$, respectively. Although the generalization of the above results to n dimensions is tedious, it uses basically the same ideas. We state without proof that the following holds.

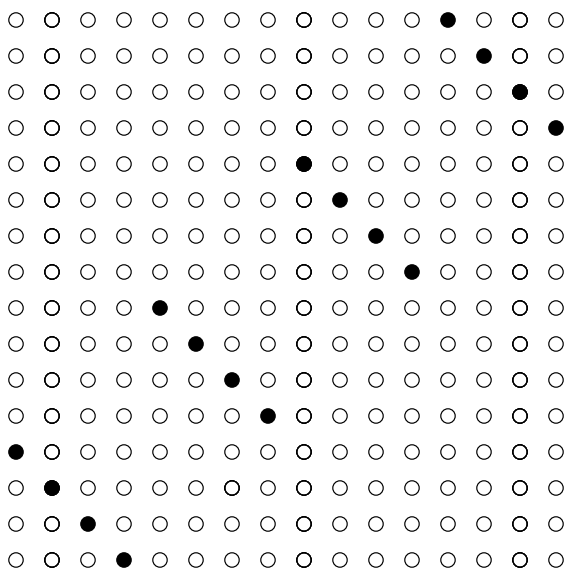


FIG. 4. *A* (16×16)*-grid with* 16 *black points.*

COROLLARY 5.6. $O\left((2en/\epsilon)^n \cdot (n\ln(n/\epsilon) + \ln(1/\delta))\right)$ *examples are sufficient to learn* $SBV(n)$ *with* ϵ*-bounded absolute difference if the domain distribution is a product distribution.*

As in the two-dimensional case, we obtain the corresponding bound for learning with an (ϵ, γ)-good model by substituting $\epsilon\gamma$ for ϵ in the bound of Corollary 5.6. Of course, $SBV(n)$ is not learnable in the distribution-free model for all $n \geq 2$. We do not know whether $BV(n)$ is learnable in this model. However, we know the following bounds:

$$\left\lfloor \frac{1}{2n\gamma} \right\rfloor^n \leq d_{BV(n)}(\gamma) \leq \left(1 + \frac{1}{2\gamma}\right)^{2^{n-1}}.$$

We made no serious attempt to close the large gap between the lower and the upper bound on $d_{BV(n)}(\gamma)$.

6. Conclusions and open problems. Some statisticians prefer learning models where expectation is also taken over all random samples. The confidence parameter δ then becomes superfluous. Each of the models considered in this paper has its "partner model" in this setting. We state without proof that all of our (lower and upper) bounds can be transferred to the partner models without changing the respective order of magnitude by more than a logarithmic factor.

It is an open problem whether there are general upper bounds which match our general lower bounds modulo a logarithmic factor. We do not know how the general lower and upper bounds change when we restrict the resources of time or space for the learning algorithms. It would be interesting to show a relation between nonlearnability of functions by polynomial-time learning algorithms and complexity theory (similar to the relation which is known for deterministic concept learning). It would also be interesting to know supplementary lower bounds (perhaps completely unrelated to the combinatorial or Natarajan dimension) which succeed in cases where our bound fails. We do not know whether $BV(n)$ is learnable in the distribution-free model and, if so,

whether our lower bound on the required number of examples in terms of $d_{BV(n)}(\gamma)$ is (almost) tight. We do not know whether the upper bound on the number of examples for learning $SBV(n)$ under product distributions is (almost) tight. It would be interesting to derive distribution-dependent lower bounds for function learning.

Acknowledgments. We thank Svetlana Anoulova and Martin Dietzfelbinger for valuable conversations.

REFERENCES

[1] M. Anthony, N. Biggs, and J. Shawe-Taylor, *The learnability of formal concepts*, in Proc. 3rd Annual Workshop on Computational Learning Theory, Morgan Kaufmann, San Mateo, CA, 1990, pp. 246–258.

[2] M. Anthony and J. Shawe-Taylor, *Valid generalization of functions from close approximation on a sample*, in Proc. European Conference on Computational Learning Theory, 1993, Oxford University Press, 1994, pp. 95–109.

[3] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, *Learnability and the Vapnik–Chervonenkis dimension*, J. Assoc. Comput. Mach., 36 (1989), pp. 929–965.

[4] A. Ehrenfeucht, D. Haussler, M. Kearns, and L. Valiant, *A general lower bound on the number of examples needed for learning*, Inform. and Comput., 82 (1989), pp. 247–261.

[5] D. Haussler, *Generalizing the pac model: Sample size bounds from metric–dimension based uniform convergence results*, in Proc. 30th Annual Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 40–46.

[6] M. J. Kearns and R. E. Schapire, *Efficient distribution-free learning of probabilistic concepts*, in Proc. 31st Annual Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 382–392; full version, J. Comput. System Sci., 48 (1994), pp. 464–497.

[7] D. Kimber and P. M. Long, *The learning complexity of smooth functions of a single variable*, in Proc. 5th Annual Workshop on Computational Learning Theory, ACM, New York, 1992, pp. 153–160.

[8] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Vol. 1, 2nd ed., Addison–Wesley, Reading, MA, 1973.

[9] B. K. Natarajan, *On learning sets and functions*, Mach. Learning, 4 (1989), pp. 67–97.

[10] J. Shawe-Taylor, M. Anthony, and N. Biggs, *Bounding sample size with the Vapnik–Chervonenkis dimension*, Discrete Appl. Math., 41 (1993), pp. 65–73.

[11] H. U. Simon, *General bounds on the number of examples needed for learning probabilistic concepts*, in Proc. 6th Annual Workshop on Computational Learning Theory, ACM, New York, 1993, pp. 402–412; J. Comput. System Sci., 52 (1996), pp. 239–255.

[12] L. G. Valiant, *A theory of the learnable*, Comm. Assoc. Comput. Mach., 27 (1984), pp. 1134–1142.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 764–771, June 1997

010

A PUMPING CONDITION FOR REGULAR SETS*

STEFANO VARRICCHIO†

Abstract. We prove that a language of a finitely generated free monoid is regular if and only if it satisfies the positive block pumping property. This gives a positive answer to a problem posed by Ehrenfeucht, Parikh, and Rozenberg [*SIAM J. Comput.*, 10 (1981), pp. 536–541].

Key words. automata theory, regular languages, pumping conditions

AMS subject classifications. 68Q45, 20M35

PII. S0097539790179944

1. Introduction. Pumping properties have always had great importance in formal languages theory and have frequently been used in order to prove that some sets of words do not belong to a certain family of languages. Several pumping conditions have been found for various families of languages. In particular, we recall the well-known "pumping lemma" for regular languages (cf. [9, pp. 55–56]) that gives a necessary condition for a language to be regular.

This condition alone does not enforce the regularity of a language, so one may be induced to consider stronger pumping properties in order to obtain conditions equivalent to regularity, and in fact many papers have been devoted to this problem [1], [3], [7], [8].

The block pumping properties introduced in [3] are quite interesting. In that paper it is proved that the regularity of a language is equivalent to a block pumping property with a nonnegative pump. Moreover, it is questioned whether a positive block pumping property is a regularity condition.

In this paper we will give a positive answer to that question, proving that a language of a finitely generated free monoid is regular if and only if it satisfies a block pumping property with positive pump. de Luca and Varricchio [1], [2] have given a partial answer to the problem when the pumping condition is assumed to be uniform. In this case the pumping property may be expressed as an iteration property of the syntactic monoid, and, for a language L, the problem is led back to the finiteness of $M(L)$.

In the following, A will denote a finite set, or *alphabet*, and A^* will denote the *free monoid* over A. The elements of A are called *letters* and those of A^* are called *words*. The identity element of A^* is denoted by Λ. For any word w, $|w|$ denotes its *length*. Let $u, v \in A^*$. We then say that u is a *factor* (resp., *prefix*) of v if there exist $\lambda, \mu \in A^*$ such that $v = \lambda u \mu$ (resp., $v = u\mu$). We set $u < v$ when u is a prefix of v. A^k (resp., $A^{[k]}$) denotes the set of the words having length k (resp., length $\leq k$). A *(right) infinite word* is a map b: $\mathbb{N} \to A$. The set of infinite words is denoted by A^ω. Moreover we set $A^\infty = A^* \cup A^\omega$. For any $b \in A^\infty$, $F(b)$ will denote the set of all finite words which are factors of b. The subsets of A^* are called *languages*. If L is a language, $F(L)$ is the set of the factors of words of L. Let L be a language of A^* and $\sigma \in A^*$, we set $L_\sigma = \{x \in A^* \mid \sigma x \in L\}$. Let $u, v \in A^*$ and $L \subseteq A^*$. We set $u \equiv_L v$ (resp., $u \ N_L \ v$) if the following condition holds: $\forall\, x, y \in A^*\ xuy \in L \Leftrightarrow xvy \in L$

*Received by the editors April 16, 1990; accepted for publication (in revised form) July 17, 1995. http://www.siam.org/journals/sicomp/26-3/17994.html

†Dipartimento di Matematica Pura e Applicata, Università di L'Aquila, Via Vetoio, 67010 L'Aquila, Italy (varricchio@univaq.it).

(resp., $\forall x \in A^*\ ux \in L \Leftrightarrow vx \in L$). N_L and $\equiv_L$ are called, respectively, the *right Nerode congruence* and *the syntactic congruence* of L. The *syntactic monoid* of L is the quotient monoid $M(L) = A^*/\equiv_L$. A monoid M is called *periodic* if for any $m \in M$, there exist two positive integers n, k, such that $m^n = m^{n+k}$.

The reader is referred to the books of Eilemberg [4] and Hopcroft and Ullman [9] for the basic notions concerning the theory of formal languages. Here we recall the fundamental theorem of Myhill–Nerode [10], [11] which says that a language L is regular if and only if the right congruence N_L has finite index.

2. Preliminaries. In this section we will recall some recent results of combinatorics on words that are fundamental tools in the study of finiteness condition for semigroups and in several combinatorial problems.

DEFINITION 2.1. *Let $b \in A^\omega$ be an infinite word. We say that b is* uniformly recurrent *if there exists a function c: $A^* \to \mathbb{N}$ such that if w, $v \in F(b)$ and $|v| \geq c(w)$, then w is a factor of v.*

We observe that, given a uniformly recurrent word b, it is possible to consider the function C: $\mathbb{N} \to \mathbb{N}$, defined by

$$C(n) = \max\{c(w) \mid w \in F(b) \cap A^n\}.$$

This function satisfies the following property. For any $n > 0$ and $v \in F(b)$ with $|v| > C(n)$, one has that

$$F(b) \cap A^{[n]} \subseteq F(v).$$

The importance of uniformly recurrent infinite words is due to the following.

THEOREM 2.1. *Let L be an infinite language of A^*. Then there exists an infinite word $b \in A^\omega$ such that*

(i) *b is uniformly recurrent,*

(ii) *$F(b) \subseteq F(L)$.*

The previous theorem is proved in [5] using techniques of symbolic dynamics; a combinatorial proof is given in [2]. Let us consider now some combinatorial properties of uniformly recurrent infinite words.

LEMMA 2.2. *Let $b \in A^\omega$ be a uniformly recurrent infinite word. For any $n > 0$ there exists a positive integer $K(n)$ such that for any $w \in A^*$, $a \in A$, with $wa \in F(b)$ and $|w| \geq K(n)$ one has that*

$$w = \lambda w_1 w_2 \cdots w_n,$$

with $\lambda \in A^*$, $w_i \in aA^*$ for $i \in \{1, \ldots, n\}$, and $w_i \in w_{i+1} \cdots w_n a A^*$ for $i \in \{1, \ldots, n-1\}$.

Proof. The proof is by induction on n. For $n = 1$ let $K(1) = C(1)$, where C is the function associated to the word b. Let $w \in A^*$, $|w| \geq K(1)$, and $wa \in F(b)$. Then w must contain the letter a, so $w = xay$ with x, $y \in A^*$. The statement follows if we set $w_1 = ay$.

Now let $n > 1$. By the induction hypothesis we may suppose that there exists an integer $K(n-1)$ that satisfies the statement for $n-1$. Then we set

$$K(n) = C(K(n-1) + 1) + K(n-1).$$

Let $w \in A^*$, $a \in A$ such that $|w| \geq K(n)$ and $wa \in F(b)$. We can write $w = xv$, with $|x| \geq C(K(n-1)+1)$ and $|v| = K(n-1)$. Since $va \in F(b)$, by the induction hypothesis one has

$$v = \lambda' w_2 \cdots w_n,$$

with $\lambda \in A^*$, $w_i \in aA^*$ for $i \in \{2, \ldots, n\}$ and $w_i \in w_{i+1} \cdots w_n a A^*$ for $i \in \{2, \ldots, n-1\}$. By the properties of the function C, one has that x contains va as a factor and hence also $w_2 \cdots w_n a$. Then one can write $x = \lambda w_2 \cdots w_n a \mu$, with λ, $\mu \in A^*$, so that

$$w = \lambda w_2 \cdots w_n a \mu \lambda' w_2 \cdots w_n.$$

Therefore, if we let $w_1 = w_2 \cdots w_n a \mu \lambda'$, one has $w_1 \in aA^*$, $w_1 \in w_2 \cdots w_n a A^*$, and the statement is true. □

3. Pumping conditions. In this section we recall the definitions of the pumping properties. Later on, after some technical lemmas, we will show that the positive block pumping property is a regularity condition. The proof is inspired by the techniques used in [3] to prove a similar result for the block cancellation property.

DEFINITION 3.1. *Let L be a language of A^*, $x \in A^*$, and $x = uvw$. We say that v is a* pump *for x relative to L if and only if for every $i \geq 0$*

$$u(v)^i w \in L \Leftrightarrow x \in L.$$

We say that v is a positive pump *if the latter condition is satisfied for every $i > 0$.*

DEFINITION 3.2. *Let L be a language of A^*. We say that L satisfies the property BP_m (resp., PP_m) if for any x, $w_1, w_2, \ldots, w_m$, $y \in A^*$, there exist $i, j, 1 \leq i < j \leq m+1$ such that $w_i \cdots w_{j-1}$ is a pump (resp., positive pump) for $x w_1 w_2 \cdots w_m y$ relative to L. We say that L satisfies the* block pumping property *(resp.,* positive block pumping property*) if there exists an integer $m > 0$ such that L satisfies BP_m (resp., PP_m).*

DEFINITION 3.3. *Let L be a language of A^*. We say that S satisfies BC_m if for any $x, w_1, w_2, \ldots, w_m, y \in A^*$, there exist $i, j, 1 \leq i < j \leq m+1$ such that*

$$x w_1 w_2 \cdots w_m y \in L \Leftrightarrow x w_1 w_2 \cdots w_{i-1} w_j \cdots w_m y \in L.$$

We say that L satisfies the block cancellation property *if there exists an integer $m > 0$ such that L satisfies BC_m.*

Remark. In the former definitions the integers i, j depend on the context (x, y) in which the block $w_1 w_2 \cdots w_m$ is considered. If they do not depend on the context, then the corresponding properties will be called, respectively, *uniform block pumping property* and *uniform block cancellation property.* For instance, S satisfies the uniform block cancellation property if there exists an integer $m > 0$ such that for any $w_1, w_2, \ldots, w_m \in A^*$ there exist i, j (depending only on $w_1, w_2, \ldots, w_m$), $1 \leq i < j \leq m+1$ such that for all x, $y \in A^*$

$$x w_1 w_2 \cdots w_m y \in L \Leftrightarrow x w_1 w_2 \cdots w_{i-1} w_j \cdots w_m y \in L.$$

We observe that BP_m implies BC_m and PP_m. Moreover, as shown in [3], BC_m implies BP_m. The contribution of this paper is to show that all are the same and equivalent to regularity. We recall that in [3] it has been proved that if a language L satisfies the block cancellation property, then L is regular. In [1] a similar result is proved for the uniform positive block pumping property. In what follows we will prove that any language satisfying the positive block pumping property is regular (Theorem 3.7). First we prove that the languages satisfying PP_m are closed under the map

$L \to L_\sigma$ (Lemma 3.4) and are finitely many in number (Lemma 3.6). The regularity of the languages satisfying PP_m will hence follow from a Nerode-type argument (Lemma 3.3).

Now we recall a version of the Ramsey theorem (cf. [6]). Let X be a set and r a positive integer; we denote by $P(X)$ the family of all the subsets of X, and we set $P_r(X) = \{Y \in P(X) \mid \mathrm{card}(Y) = r\}$.

THEOREM 3.1 (Ramsey). *Let r, k be positive integers with $1 \leq r \leq k$. Then there exists an integer $R(r,k)$ such that for any set X, with $\mathrm{card}(X) = R(r,k)$, and for any bipartition Y_1, Y_2 of $P_r(X)$ there exists $Y \in P_k(X)$ such that either $P_r(Y) \subseteq (Y_1)$ or $P_r(Y) \subseteq (Y_2)$.*

LEMMA 3.2. *Let m be a positive integer and $F = \{L \subseteq A^* \mid L \text{ satisfies } PP_m\}$. Then for any $k > 0$ there exists an integer M (depending on k) such that for any $(x_1, y_1, L_1), (x_2, y_2, L_2), \ldots, (x_k, y_k, L_k) \in A^* \times A^* \times F$ and for any $w_1, w_2, \ldots, w_M \in A^*$ there exist i, j, $1 \leq i < j \leq M+1$ such that $w_i \cdots w_{j-1}$ is a simultaneous positive pump for $x_s w_1 w_2 \cdots w_M y_s$ relative to L_s for $s \in \{1, 2, \ldots, k\}$.*

Proof. The proof is by induction on k. If $k = 1$, then the statement is true for $M = m$, since L_1 satisfies PP_m. Let $k > 1$. By the induction hypothesis there exists an integer M' that satisfies the statement for $k-1$.

Then let us set $M = R(2, M'+1)$, where R is the function of the Ramsey theorem. Let $w_1, w_2, \ldots, w_M \in A^*$ and $X = \{1, 2, \ldots, M\}$. Let us consider the following bipartition of $P_2(X)$:

$$Y_1 = \{\{i,j\} \mid w_i \cdots w_{j-1} \text{ is a positive pump for } x_k w_1 \cdots w_M y_k \text{ relative to } L_k\},$$

$$Y_2 = P_2(X) - Y_1.$$

For the Ramsey theorem there exists $Y \in P_{M'+1}(X)$ such that either $P_2(Y) \subseteq Y_1$ or $P_2(Y) \subseteq Y_2$. Let $Y = \{j_1, j_2, \ldots, j_{M'+1}\}$, $v_1 = w_{j_1} \cdots w_{j_2-1}, \ldots, v_{M'} = w_{j_{M'}} \cdots w_{j_{M'+1}-1}$. Since $M' \geq m$, there exist s, t with $1 \leq s < t \leq M'+1$ such that $v_s \cdots v_{t-1}$ is a positive pump for $x_k w_1 \cdots w_M y_k$ relative to L_k. Since $v_s \cdots v_{t-1} = w_{j_s} \cdots w_{j_t-1}$, one has that

$$\{j_s, j_t\} \in Y_1 \cap P_2(Y) \neq \emptyset;$$

hence $P_2(Y) \subseteq Y_1$. This means that for any i, j, with $1 \leq i < j \leq M'+1$, $v_i \cdots v_{j-1}$ is a positive pump for $x_k w_1 \cdots w_M y_k$ relative to L_k. By the induction hypothesis, we can choose i, j, with $1 \leq i < j \leq M'+1$ such that, for any $s \in \{1, \ldots, k-1\}$, $v_i \cdots v_{j-1}$ is a positive pump for $x_s w_1 \cdots w_M y_s$ relative to L_s. □

The following lemma has already been proved in [3]; however, we will report the proof for the sake of completeness.

LEMMA 3.3. *Let P be a property defined in $P(A^*)$. If the following two conditions are satisfied, then any language satisfying P is regular:*

(a) *L satisfies $P \Rightarrow L_\sigma$ satisfies P for any $\sigma \in A^*$,*

(b) *the languages of A^* satisfying P are finitely many.*

Proof. Let L be a language satisfying P and let N_L be the Nerode congruence of L. One has that for any x, $y \in A^*$

$$x N_L y \Leftrightarrow L_x = L_y.$$

Since the languages L_σ with $\sigma \in A^*$ are finitely many, then N_L has finite index and, by the Myhill–Nerode theorem, L is regular. □

LEMMA 3.4. *Let $L \subseteq A^*$ be a language satisfying PP_m; then for any $\sigma \in A^*$, L_σ satisfies PP_m.*

Proof. Since L satisfies PP_m, there exist i, j with $1 \leq i < j \leq m+1$ such that for any $s > 0$

$$\sigma x w_1 \cdots w_m y \in L \Leftrightarrow \sigma x w_1 \cdots w_{i-1} (w_i \cdots w_{j-1})^s w_j \cdots w_m y \in L.$$

Therefore, for any $s > 0$ one has

$$x w_1 \cdots w_m y \in L_\sigma \Leftrightarrow x w_1 \cdots w_{i-1} (w_i \cdots w_{j-1})^s w_j \cdots w_m y \in L_\sigma,$$

and L_σ satisfies PP_m. □

LEMMA 3.5. *Let $L \subseteq A^*$ be a language satisfying PP_m. Then the syntactic monoid $M(L)$ is periodic.*

Proof. Let $w \in L$. We prove that for any x, $y \in A^*$

$$x w^m y \in L \Leftrightarrow x w^{m+m!} y \in L.$$

The condition PP_m, applied to $w_1 = w_2 = \cdots = w_m = w$, shows that there exists an integer $k < m$ such that for any $s > 0$

$$x w^m y \in L \Leftrightarrow x w^{m+sk} y \in L.$$

Since k divides $m!$, there exists an integer $s > 0$ such that $sk = m!$ and the statement is true. □

Remark. Let L, $L' \subseteq A^*$ be two languages such that $M(L)$ and $M(L')$ are periodic. Then for any $w \in A^*$ there exist positive integers n, k, n', k' such that $w^n \equiv_L w^{n+k}$ and $w^{n'} \equiv_{L'} w^{n'+k'}$. We can obviously suppose that $n = n'$ and $k = k'$, otherwise n and n' could be replaced by $\max(n, n')$ and k, k' by $\operatorname{lcm}(k, k')$ and the former relations still would hold.

LEMMA 3.6. *The languages of A^* satisfying PP_m are finitely many.*

Proof. Suppose that there exists an integer $k > 0$ such that for any L_1, $L_2 \subseteq A^*$ satisfying PP_m one has

$$L_1 \cap A^{[k]} = L_2 \cap A^{[k]} \Rightarrow L_1 = L_2.$$

Then the statement is trivially true.

Let us suppose by contradiction that there exist infinitely many languages of A^* satisfying PP_m. Then such an integer k could not exist. Hence for any $k > 0$ there exists a word $w_k \in A^*$ and two languages $L_{1,k}$, $L_{2,k}$ satisfying PP_m such that

$$L_{1,k} \cap A^{[k]} = L_{2,k} \cap A^{[k]}$$

and

$$w_k \in L_{1,k}, \qquad w_k \notin L_{2,k}.$$

Moreover we may suppose that w_k has minimal length; that is, for any $l < |w_k|$ one has

$$L_{1,k} \cap A^{[l]} = L_{2,k} \cap A^{[l]};$$

moreover, $|w_k| > k$.

Let us consider now the language $L = \{w_1, w_2, \ldots, w_k, \ldots\}$. Since L is infinite, by Theorem 2.1 there exists a uniformly recurrent infinite word $b \in A^\omega$ such that $F(b) \subseteq F(L)$. Let C, K: $\mathbb{N} \to \mathbb{N}$ be the functions associated with the infinite word b as in Definition 2.1 and Lemma 2.2. Let M be a positive integer satisfying Lemma 3.2 for $k = 4$.

Let $N = (M+1) \cdot C(K(M))$. Let u be a factor of b such that $|u| = N$. Since $F(b) \subseteq L$, there exist $w \in L$, x, $y \in A^*$ such that $w = xuy$. Moreover, there exist two languages L_1, $L_2 \subseteq A^*$ satisfying PP_m such that for any $l < |w|$

$$L_1 \cap A^{[l]} = L_2 \cap A^{[l]}$$

and

$$w \in L_1, \qquad w \notin L_2.$$

Since $|u| = N$, we can write $u = u_1 u_2 \cdots u_{M+1}$, with $|u_i| \geq C(K(M))$ and all u_i are in $F(b)$. Therefore, there exists a word $v \in F(b)$ with $|v| = K(M)$ such that

$$u_i = \lambda_i v \mu_i,$$

$\lambda_i, \mu_i \in A^*, 1 \leq i \leq M+1$. Hence, since $w = xuy$, one can write

$$w = \lambda v_1 v_2 \cdots v_{M+1} \mu,$$

where $\lambda = x\lambda_1, \mu = \mu_{n+1} y, v_i \in vA^*, |v| = K(M)$, and $v_1 v_2 \cdots v_{M+1} \in F(b)$.

By Lemma 3.2 there exist i, j with $1 \leq i < j \leq M+1$ such that $z = v_i, \cdots v_{j-1}$ is a positive pump for $w = \lambda v_1 v_2 \cdots v_{M+1} \mu$ relative to L_1 and L_2. Therefore, letting $x' = \lambda v_1 v_2 \cdots v_{i-1}, y' = v_j \cdots v_{M+1} \mu$, one has $w = x'zy', y' \in vA^*$, $zv \in F(b)$, and for any $s > 0$

$$w = x'zy' \in L_t \Leftrightarrow x'z^s y' \in L_t \tag{3.1}$$

for $t = 1, 2$.

By Lemma 3.5, $M(L_1)$ and $M(L_2)$ are periodic. Then there exist n, $k > 0$ such that

$$z^n \equiv_{L_t} z^{n+k} \tag{3.2}$$

for $t = 1, 2$.

From (3.1) for $s = k$ we have

$$x'zy' \in L_t \Leftrightarrow x'z^k y' \in L_t \tag{3.3}$$

for $t = 1, 2$.

Now let z' be the maximal common prefix of z^n and y'. Since $v < z^n$ and $v < y'$ we have that $|z'| \geq |v| \geq K(M)$. Let $y'' \in A^*$ such that $y' = z'y''$.

We now prove the following claim.

Claim A. There exists $\lambda \in A^*$ such that

$$x'z'y'' \in L_t \Leftrightarrow x'z^n \lambda y'' \in L_t \tag{3.4}$$

and

$$x'z^k z'y'' \in L_t \Leftrightarrow x'z^k z^n \lambda y'' \in L_t \tag{3.5}$$

for $t = 1, 2$.

The proof is by induction on $q = |z^n| - |z'|$. If $q = 0$ then $z' = z^n$ and (3.4), (3.5) hold for $\lambda = \Lambda$. Let $q > 0$ and z' be a proper prefix of z^n. Let z'' be the suffix of z' such that $|z''| = K(M)$ and let $a \in A$ such that $z'a < z^n$. Since $|v| = K(M) = |z''|$ and $z \in vA^*$, as $z''a \in F(z^n)$, one can easily see that $z''a \in F(zv) \subseteq F(b)$. Then by Lemma 2.2 we can write

$$z'' = \zeta w_1 w_2 \cdots w_M,$$

with $\zeta \in A^*$, $w_i \in aA^*$ for $i \in \{1, \ldots, M\}$ and $w_i \in w_{i+1} \cdots w_M a A^*$ for $i \in \{1, \ldots, M-1\}$. Since z'' is a suffix of z', one has

$$z' = \zeta' w_1 w_2 \cdots w_M$$

for a suitable $\zeta' \in A^*$.

Now we may apply Lemma 3.2 to the four terms $(x'\zeta', y'', L_1)$, $(x'\zeta', y'', L_2)$, $(x'z^k\zeta', y'', L_1)$, and $(x'z^k\zeta', y'', L_2)$. Therefore, there exist i, j with $1 \le i < j \le M+1$ such that

(3.6)
$$x'z'y'' = x'\zeta' w_1 \cdots w_M y'' \in L_t \Leftrightarrow x'\zeta' w_1 \cdots w_{i-1}(w_i \cdots w_{j-1})^2 w_j \cdots w_M y'' \in L_t$$

and

(3.7)
$$x'z^k z'y'' = x'z^k\zeta' w_1 \cdots w_M y'' \in L_t \Leftrightarrow x'z^k\zeta' w_1 \cdots w_{i-1}(w_i \cdots w_{j-1})^2 w_j \cdots w_M y'' \in L_t$$

for $t = 1, 2$.

Now, considering that $w_i \in w_j \cdots w_M a A^*$ for $i < j$, then for a suitable $\xi \in A^*$ $w_i \cdots w_M = w_j \cdots w_M a \xi$, and substituting on the right side of (3.6) and (3.7) we have

$$x'z'y'' \in L_t \Leftrightarrow x'z'a\xi y'' \in L_t \tag{3.8}$$

and

$$x'z^k z'y'' \in L_t \Leftrightarrow x'z^k z'a\xi y'' \in L_t \tag{3.9}$$

for $t = 1, 2$.

Therefore, since $z'a < z^n$ and $|z^n| - |z'a| = q - 1$, by the induction hypothesis there exists $\xi' \in A^*$ such that

$$x'z'a\xi y'' \in L_t \Leftrightarrow x'z^n\xi'\xi y'' \in L_t \tag{3.10}$$

and

$$x'z^k z'a\xi y'' \in L_t \Leftrightarrow x'z^k z^n \xi'\xi y'' \in L_t \tag{3.11}$$

for $t = 1, 2$.

So if we set $\lambda = \xi'\xi$, comparing (3.8) with (3.10) and (3.9) with (3.11), we obtain (3.4) and (3.5), respectively, and Claim A is proved.

From (3.3), (3.5), and (3.2), considering that $z'y'' = y'$, one derives

$$x'zy' \in L_t \Leftrightarrow x'z^k y' = x'z^k z'y'' \in L_t \Leftrightarrow x'z^{k+n}\lambda y'' \in L_t \Leftrightarrow x'z^n \lambda y'' \in L_t$$

for $t = 1, 2$.

Therefore, from (3.4) it follows that

$$x'zy' \in L_t \Leftrightarrow x'y' \in L_t \tag{3.12}$$

for $t = 1, 2$.

We recall that $L_1 \cap A^{[l]} = L_2 \cap A^{[l]}$ for $l < |w|$ and therefore

$$x'y' \in L_1 \Leftrightarrow x'y' \in L_2.$$

Hence by (3.12),

$$x'zy' \in L_1 \Leftrightarrow x'zy' \in L_2.$$

From here, since $x'zy' = w$, we have

$$w \in L_1 \Leftrightarrow w \in L_2,$$

which is a contradiction, because $w \in L_1$ and $w \notin L_2$. □

THEOREM 3.7. *Let $L \subseteq A^*$. Then L is regular if and only if L satisfies the positive block pumping property.*

Proof. If L satisfies the positive block pumping property then there exists an integer m such that L satisfies PP_m. From Lemmas 3.6, 3.4, and 3.3 it follows that L is regular. Conversely, if L is regular, then it is easy to prove [3] that L satisfies PP_m for a suitable $m > 0$. □

REFERENCES

[1] A. DE LUCA AND S. VARRICCHIO, *A positive pumping condition for regular sets*, Bull. EATCS, 39 (1989), pp. 171–175.

[2] A. DE LUCA AND S. VARRICCHIO, *Finiteness and iteration conditions for semigroups*, Theoret. Comput. Sci., 87 (1991), pp. 315–327.

[3] A. EHRENFEUCHT, R. PARIKH, AND G. ROZENBERG, *Pumping lemmas for regular sets*, SIAM J. Comput., 10 (1981), pp. 536–541.

[4] S. EILEMBERG, *Automata, Languages and Machines*, Vol. A, Academic Press, New York, 1974.

[5] H. FUSTEMBERG, *Poincaré recurrence and number theory*, Bull. Amer. Math. Soc., 5 (1981), pp. 211–234.

[6] R. L. GRAHAM, B. L. ROTSHILD, AND J. H. SPENCER, *Ramsey Theory*, 2nd ed., J. Wiley, New York, 1990.

[7] J. JAFFE, *A necessary and sufficient pumping lemma for regular languages*, SIGACT News, 10 (1978), pp. 48–49.

[8] K. HASHIGUSHI, *Notes on congruences relations and factor pumping conditions for rational languages*, Theoret. Comput. Sci., 57 (1988), pp. 303–316.

[9] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages and Computations*, Addison-Wesley, New York, 1979.

[10] J. MYHILL, *Finite automata and the representation of events*, WADD Technical Report 57-624, Wright–Patterson Air Force Base, Dayton, OH, 1957.

[11] A. NERODE, *Linear automata transformations*, Proc. Amer. Math. Soc., 9 (1958), pp. 541–544.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 772–785, June 1997

011

LEADER ELECTION IN COMPLETE NETWORKS*

GURDIP SINGH†

Abstract. Leader election is a fundamental problem in distributed computing and has a number of applications. This paper studies the problem of leader election in complete asynchronous networks. We present a message-optimal protocol that requires $O(N \log N)$ messages and $O(N/\log N)$ time, where N is the number of nodes in the system. The time complexity of this protocol is a significant improvement over currently known protocols for this problem. We also give a family of protocols with message and time complexities $O(Nk)$ and $O(N/k)$, respectively, where $\log N \leq k \leq N$. Many problems such as spanning-tree construction and computing a global function are equivalent to leader election in terms of their message and time complexities, and therefore our result improves the time complexity of these problems as well.

Key words. distributed algorithms, leader election, time complexity, complete networks

AMS subject classifications. 68Q22, 68Q25, 68N25

PII. S0097539794276865

1. Introduction.

In the leader-election problem, there are N nodes in the network, each having a unique identity. Initially all nodes are *passive*. An arbitrary subset of nodes, called the *candidates*, wake up spontaneously and start the protocol. On the termination of the protocol, exactly one node announces itself the leader. Leader election is a fundamental problem in distributed computing and has several applications. It is an important tool for breaking symmetry in a distributed system. By distinguishing a single node as a leader, it is possible to execute centralized protocols in a decentralized environment.

The problem of leader election has been studied in various computation models and topologies. For an asynchronous network with an arbitrary topology, a lower bound of $\Omega(E + N \log N)$ messages holds, where E is the number of edges in the network, and a protocol with this complexity was given in [5]. The time complexity of this protocol is $O(N \log N)$. In [3], an improved protocol with time complexity $O(N)$ was proposed. Several protocols for ring topologies have also been proposed [6, 4]. In this paper, we consider the problem of electing a leader in asynchronous complete networks. In a complete network, each pair of nodes is connected by a bidirectional link. Complete networks have been studied extensively since they provide bounds for other types of networks (with less connectivity). In the rest of the paper, by a network, we will mean a complete network.

In [2], a lower bound of $\Omega(N \log N)$ was proved on the message complexity of leader election for synchronous networks and showed that any protocol sending at most $O(N \log N)$ messages must require at least $\Omega(\log N)$ time. For asynchronous networks, it was shown in [7] that $\Omega(N \log N)$ messages are required for leader election and proposed a protocol which requires $O(N \log N)$ messages and $O(N \log N)$ time. In [1], the complexity was further improved by giving a series of protocols for asynchronous networks, each with $O(N \log N)$ message complexity and $O(N)$ time complexity.

*Received by the editors November 9, 1994; accepted for publication (in revised form) July 17, 1995. This research was supported by NSF grants CCR8901966, CCR9211621, and CCR9502506. Some results contained in this paper appear in a preliminary form in *Proc. ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1992, pp. 179–190.

http://www.siam.org/journals/sicomp/26-3/27686.html

†Department of Computing and Information Sciences, Kansas State University, 234 Nichols Hall, Manhattan, KS 66506 (singh@cis.ksu.edu).

In [2], it was conjectured that $\Omega(N)$ is a lower bound on the time complexity of any message-optimal election protocol for asynchronous networks. We disprove this conjecture by proposing a protocol which requires $O(N \log N)$ messages and $O(N/\log N)$ time. There are many problems such as spanning-tree construction, computing a global function, and counting which are equivalent to leader election in terms of message and time complexities [3]. Therefore, our protocol leads to improvements in the time complexity of these problems as well. To arrive at this protocol, we first develop two protocols, Γ and $\mathcal{A}$. Γ has message and time complexities $O(N^2)$ and $O(1)$, respectively. $\mathcal{A}$ is a variation of the protocol **A** of [1] and has message and time complexities $O(N \log N)$ and $O(N)$, respectively. We combine features of these protocols to obtain our final protocol with the desired complexity. We find that the problem of awakening N nodes using $O(N \log N)$ messages dominates the time complexity of our protocol. We also present a modified message optimal protocol which requires $O(\min(r, N/\log N) + \log N)$ time. In this protocol, the $\min(r, N/\log N)$ factor corresponds to the time required to awaken all nodes to participate in the protocol. In [8], it was also shown that any protocol sending $O(N \log N)$ messages requires at least $\Omega(N/\log N)$ time, which implies that our protocol is message and time optimal.

The interaction between time and message complexities for any problem is an important issue and it is desirable to have a spectrum of protocols which trade off message complexity for execution time. For asynchronous complete networks, we give a family of protocols, $D(k)$, where $\log N \leq k \leq N$. The message and time complexities of $D(k)$ are (Nk) and $O(N/k)$, respectively.

This paper in organized as follows. In the next section, we give a brief description of our model of distributed computation. In section 3, we describe the component protocols Γ and $\mathcal{A}$. Section 4 presents our protocol and its complexity analysis and correctness. In section 5, we extend this protocol to obtain a family of protocols. Section 6 deals with extensions of our protocol. Finally, we conclude in section 7.

2. Model. We model the communication network as a complete graph (N, E), where N and E represent the processors and communication links, respectively. We assume that each node has a unique identity. By a node i, we will mean a node with identity i. We assume that the network is asynchronous, i.e., message-transmission and processing times are unpredictable but finite. Messages sent over a link arrive at their destination in the order sent and are not lost.

The message complexity of a protocol is the maximum number of messages sent during any possible execution of the protocol. The time complexity of a protocol is the worst-case execution time assuming that each message takes at most one time unit to reach its destination and computation time is negligible. Furthermore, intermessage delay on a link is assumed to be at most one time unit. This assumption is made only for calculating the time complexity and the correctness of the protocol does not depend on this assumption.

In the leader-election problem, an arbitrary subset of nodes, called the *candidates*, wake up spontaneously to start the protocol. All other nodes are *passive*. A passive node wakes up on receiving a message of the protocol. Initially, a node knows only its own identity. At the termination of the protocol, exactly one node among the candidates is elected and all nodes know the identity of this node. All additions in the paper are assumed modulo N.

3. Component leader-election protocols. In this section, we develop two protocols, Γ and $\mathcal{A}$, for leader election. Protocol Γ has $O(N^2)$ message complexity

and $O(1)$ time complexity while $\mathcal{A}$ has $O(N \log N)$ message complexity and $O(N)$ time complexity. We will then combine these protocols to obtain the final protocol.

3.1. Protocol Γ. In this protocol, a *candidate* attempts to capture all other nodes. The node that is able to capture all other nodes declares itself the leader. On waking up, a candidate sends its identity in an *elect* message on all incident edges. When a node j receives an *elect*(i) message over edge e, it behaves as follows:

If j is a candidate and $j > i$, then no response is sent over e.

Otherwise, j sends an *accept* message over e.

A node that receives an *accept* message on all incident edges declares itself the leader and notifies all nodes of this fact. In this protocol, the candidate node with the largest identity is elected the leader. The time complexity of this protocol is $O(1)$. However, its message complexity is $O(N^2)$ since the number of candidate nodes may be $O(N)$, each of which sends $O(N)$ messages.

3.2. Protocol $\mathcal{A}$. Protocol $\mathcal{A}$ is a modification of the protocol **A** of [1]. We cannot use **A** directly to obtain the protocol with $O(N/\log N)$ time complexity since it suffers from link congestion, which is one of the sources of $O(N)$ time complexity. We will first describe the protocol **A** of [1], which has $O(N \log N)$ message complexity. Section 3.3 describes the modifications required to obtain $\mathcal{A}$.

The protocol described here is a slight variation of **A** (for simplicity, we use additional message types). The outline of the protocol is as follows:

A candidate node tries to capture other nodes in a sequential manner by sending *capture* messages on its incident edges one at a time. A node that is successful in capturing all other nodes is elected the leader. A candidate node i sends its identity and a variable, $level_i$, in the *capture* messages to contest with other nodes ($level_i$ is the number of nodes which i has captured so far). If a capture message from i reaches a node j which has not yet been captured and $(level_i, i)$ is greater than $(level_j, j)$ (lexicographically), then i captures j; otherwise, i is killed. If j is a captured node, then i has to kill j's owner before claiming j. If i is successful in capturing j, then it increments $level_i$ and proceeds with its conquest by sending a capture message to another node.

We will now describe the protocol in detail. Each node maintains the following variables:

- Each node has a variable $state_i$ which is either *passive*, *candidate*, *captured*, or *killed*. Initially, $state_i$ is *passive* for all nodes i.
- Every candidate node i maintains a list of edges, called $untraversed_i$, over which it has not yet sent a message to capture the node at the other end. Initially, $untraversed_i$ is the set of edges incident on i.
- The $owner_i$ of a node i is the identity of the node which captured it, and $owner\text{-}link_i$ is the edge leading to $owner_i$.
- Each node i has a variable $level_i$ which is initially 0. For a candidate node i which has not yet been captured, $level_i$ is the number of nodes it has captured so far. For any other node i, $level_i$ is the maximum level number received in any message by i.
- $maxid_i$ is the identity of the node from which the message containing the maximum level number was received. ($maxid_i$ is initially 0, and for an uncaptured candidate node, $maxid_i = i$.) Hence at any time, as far as i can tell, $(level_i, maxid_i)$ is the largest such pair in the system.

In the following, by setting (l, d) to (l', d'), we will mean the execution of the statements $l := l'$ and $d := d'$. Also, $(l, d) > (l', d')$ if $(l > l')$ or $(l = l'$ and $d > d')$.

On waking up spontaneously, a candidate node i sends a $capture(level_i, i)$ message on an edge in $untraversed_i$. When node j receives a $capture(l, i)$ message over edge e, it behaves as follows:

• *Case* 1: $state_j \neq$ captured.

⋆ If $(level_j, maxid_j) > (l, i)$, then j does not send any response over e. (If no response is sent to i, then i is blocked and will therefore never become the leader.) In this case, j knows that there is a node with a higher $(level, id)$ than (l, i).

⋆ Otherwise, j sets $(level_j, maxid_j)$ to (l, i). i becomes j's owner and e is marked as $owner\text{-}link_j$. Further, j sends an *accept* message over e and changes $state_j$ to captured.

• *Case* 2: $state_j =$ captured.

⋆ If $(level_j, maxid_j) > (l, i)$, then no response is sent to i.

⋆ Otherwise, j sets $(level_j, maxid_j)$ to (l, i). In this case, i must kill j's owner before claiming j. This is done to ensure that, at any time, the sets of nodes captured by candidate nodes are disjoint. For this purpose, j sends a $kill\text{-}owner(l, i)$ message over $owner\text{-}link_j$.

When $owner_j$ receives a $kill\text{-}owner(l, i)$ message over edge e', it behaves as follows:

If $(level_{owner_j}, maxid_{owner_j}) > (l, i)$, then $owner_j$ does not respond.

Otherwise, $owner_j$ sets $(level_{owner_j}, maxid_{owner_j})$ to (l, i) and sends an $owner\text{-}accept(i)$ message over e'. If $owner_j$ is not a captured node, then it changes *state* to *killed*. A *killed* node does not attempt to capture any more nodes. In this case, i cannot include $owner_j$ in its set of captured nodes. To capture $owner_j$, it must send a *capture* message directly to it.

If j receives an $owner\text{-}accept(i)$ message and $maxid_j$ is i (i.e., no other *capture* message with a higher $(level, id)$ has been received), then i becomes j's owner and the edge over which the *capture* message from i was received is marked as $owner\text{-}link_j$. Further, an *accept* message is sent over this edge.

If node i receives an *accept* message and is not *killed* or *captured* then it increases $level_i$ by 1 and proceeds by sending a *capture* message over the next edge in $untraversed_i$. A node that captures all other nodes declares itself the leader.

A candidate node stops capturing nodes if it is *killed* or *captured* or its *capture* message reaches a node with a higher $(level, id)$. We say that a candidate node becomes *dormant* when it stops capturing nodes. Notice that node i may not know that it has become *dormant* if this is due to its *capture* message reaching a node with a higher $(level, id)$.

The message and time complexities of protocol **A** are $O(N \log N)$ and $O(N)$, respectively [1]. The following lemma is given in [1]. (The proof of this lemma relies on the property that at any time, the sets of nodes captured by different candidates are disjoint.)

LEMMA 3.1. *For any given k, where $1 \leq k < N$, the number of nodes that own N/k nodes or more is at most k.*

3.3. Modifications to protocol A. One of the sources of $O(N)$ time complexity of **A** is possible congestion on the links. For example, consider the situation where j is a captured node, *capture* messages from nodes $i_1, i_2, i_3, \ldots, i_m$ arrive at node j in the order given, and in each case j sends a *kill-owner* message to $owner_j$. Further, assume that only the *kill-owner* message of i_m is successful in killing $owner_j$. *kill-owner* messages for all nodes i_l, $1 \leq l \leq m$, are sent over the same edge and each may take one time unit to arrive at $owner_j$. Since the intermessage delay on a particular

edge may be one time unit, the *kill-owner* message for i_m may arrive at $owner_j$ only after m time units. Since m could be $O(N)$, it may take $O(N)$ time for i_m to capture j. Thus capturing a node may take up to $O(N)$ time.

In the following, we modify **A** to obtain $\mathcal{A}$, in which if a node is able to capture another node, then it does so in a constant amount of time. The algorithm for a node j when $state_j$ = candidate is given in Figure 3.1, and the algorithm for a node j when $state_j \neq$ candidate is given in Figure 3.2. We have omitted the subscript j from variables in the figures for clarity.

```
state := candidate; level := 0;
untraversed := list of incident edges; maxid := j
while state = candidate
do
    if level = N − 1
        then send elected(j) to all incident edges; state := elected
        else select an edge e from untraversed;
             delete e from untraversed;
             send capture(level, j) over e;
             Received := false
    while state = candidate and ¬Received
    do
        receive M on edge e
            Case M of
            capture(l, i):   if (level, maxid) < (l, i) then
                                 (level, maxid) := (l, i); owner := i;
                                 owner-link := e; send accept over e;
                                 state := captured
            accept:          level := level + 1; Received := true
            kill-owner(l, i): if (level, maxid) < (l, i) then
                                 (level, maxid) := (l, i); state := killed;
                                 send owner-accept(i) over e;
                              else send killed(i) over e
            elected(i):      state := captured
    od
od
```

FIG. 3.1. *Protocol for a candidate node j.*

The only modification to **A** is in the response to the *capture* message by a captured node. (The rest of the protocol is the same.) Each node j maintains a variable $waiting_j$ which is true if j has sent a *kill-owner* message but has received no response from $owner_j$. Let a *captured* node j receive a $capture(l, i)$ message over edge e.

• If $(level_j, maxid_j) > (l, i)$, then no response is sent over e. Otherwise, j sets $(level_j, maxid_j)$ to (l, i). Further,

⋆ if $\neg waiting_j$, then the $kill\text{-}owner(l, i)$ message is sent over $owner\text{-}link_j$ and $waiting_j$ is set to true;

⋆ if $waiting_j$, then the response to node i is delayed.

• When $owner_j$ receives a $kill\text{-}owner(l, i)$ message over e', it behaves as follows: If $(level_{owner_j}, maxid_{owner_j}) > (l, i)$, then $owner_j$ sends a $killed(i)$ message over e' (indicating the fact that node i did not survive). Otherwise, it sets $(level_{owner_j}, maxid_{owner_j})$

```
finished := false; waiting := false
while ¬finished
do
    receive M over edge e
    if M is the first message received then
                        state := captured; maxid := 0; level := 0
    case M of
    capture(l, i):      if (level, maxid) < (l, i) then
                            if state = killed then
                                send accept over e; (level, maxid) := (l, i);
                                state := captured; owner-link := e
                            else
                                (level, maxid) := (l, i); potential-owner := e
                                if ¬waiting then
                                    send kill-owner(l, i) over owner-link;
                                    waiting := true;
    kill-owner(l, i):   if (level, maxid) < (l, i) then
                            (level, maxid) := (l, i);
                            send owner-accept(i) over e;
                        else send killed(i) over e
    owner-accept(i):        owner := maxid; owner-link := potential-owner;
                            send accept over owner-link
    killed(i):          if maxid ≠ i then
                            send kill-owner(level, maxid) over owner-link
                        else waiting := false;
    elected(i):         finished := true
od
```

FIG. 3.2. *Protocol for a noncandidate node j.*

to (l, i) and sends an *owner-accept*(i) message over e'. If $owner_j$ is not a captured node, then it changes *state* to *killed*.

• If j receives a *killed*(i) message from its owner and $maxid_j$ is not i, then j sends a *kill-owner*$(level_j, maxid_j)$ message over $owner\text{-}link_j$. If j receives an *owner-accept*(i) message, then $maxid_j$ becomes j's owner. The edge over which the capture message from $maxid_j$ was received is marked as $owner\text{-}link_j$ and an *accept* message is sent over this edge.

LEMMA 3.2. *If node i is successful in capturing node j, then it does so in a constant amount of time.*

Proof. Consider a node i which is successful in capturing node j. After the *capture*(l, i) message from i reaches j, then depending on the state of j, we have the following cases:

(1) If j is not a captured node, then j will send an *accept* message. This message will reach i within two time units (because there may already be a message in transit from j to i). Hence capturing j in this case will take at most three time units.

(2) Let j be a captured node. If j is not *waiting*, then a *kill-owner* message will be sent to $owner_j$ and $owner_j$ will respond with an *owner-accept* message. This message will be received by j within three time units of it sending the *kill-owner*

message (because there may be an *accept* message in transit from j to $owner_j$ when the *kill-owner* message is sent). Node j will then send an *accept* message to i. This capturing will take a total of at most five time units.

If j is *waiting*, then a response from j's owner will be received within three time units. If an *owner-accept* message is received by j, then it will send an *accept* message to i (since i captures node j). If a *killed* message is received, then a *kill-owner*(l, i) message will be sent to $owner_j$. A response to this will be received within two time units. Thus, if j is *waiting*, it will take at most seven time units to capture j.

Hence if node i successfully captures node j, then it does so in a constant amount of time (at most seven time units).

4. A composite protocol. We will now combine Γ and $\mathcal{A}$ to obtain a protocol $\mathcal{B}$. Protocol $\mathcal{B}$ has $O(N \log N)$ message complexity and requires $O(N/\log N)$ time after the node which is elected the leader wakes up. However, due to a specific wakeup pattern, the time complexity of $\mathcal{B}$ is still $O(N)$. We first present $\mathcal{B}$ and then discuss the modification to eliminate this second source of $O(N)$ time complexity.

Protocol Γ requires $O(N^2)$ messages because the number of candidates for election (which are the *candidates*) can be $O(N)$. However, if the number of candidates is reduced to $O(\log N)$ before this protocol is executed, then it will require $O(N \log N)$ messages. In protocol $\mathcal{A}$, there can be at most $\log N$ nodes at level $N/\log N$ (from Lemma 3.1). In protocol $\mathcal{B}$, we use protocol $\mathcal{A}$ to reduce the number of candidates for protocol Γ by requiring a node to execute $\mathcal{A}$ until its level number reaches $N/\log N$ and Γ thereafter. Protocol $\mathcal{B}$ is as follows:

On waking up spontaneously, a node starts executing protocol $\mathcal{A}$. When a node reaches level $N/\log N$, it sends an *elect* message with its identity on all incident edges. Let node j receive an *elect*(i) message over e. If $(level_j, maxid_j)$ is less than $(N/\log N, i)$, then j changes *state* to *killed* and sends an *accept* message over e. A node which receives an *accept* message on all incident edges declares itself the leader.

Since the message complexity of $\mathcal{A}$ is $O(N \log N)$ and at most $\log N$ nodes broadcast an *elect* message, the message complexity of $\mathcal{B}$ is $O(N \log N)$. Since it takes a constant amount of time to capture a node (from Lemma 3.2), it will take $O(N/\log N)$ time for a node to reach level $N/\log N$ after it wakes up (if it reaches this level). After a node reaches this level, it broadcasts an *elect* message. If it is elected the leader, it will receive all replies in a constant amount of time and will declare itself the leader. Therefore, the node which is elected the leader takes $O(N/\log N)$ time after waking up spontaneously to declare itself the leader. (Note that if we had used **A** instead of $\mathcal{A}$, then $\mathcal{B}$ would still require $O(N)$ time after the leader wakes up.)

However, a node may wake up spontaneously $O(N)$ time units after the first node wakes up and be elected the leader as shown in the following example. Assume that nodes have identities in the range $1, \ldots, N$ (as shown in Figure 4.1) and node 1 is the first node to wake up spontaneously. After waking up spontaneously, node i sends a *capture* message to node $i+1$. Assume that the message from i reaches $i+1$ just after $i+1$ wakes up and $i+1$ sends the message to capture $i+2$ before receiving the message from i. In this case, no response will be sent to i since it has the same level number as $i+1$ but a smaller identity. If this happens for all sites i, $1 \leq i \leq N$, then only node N will survive and capture all other nodes. If the capture message for each node takes exactly one time unit to arrive (as shown in Figure 4.1), node N will wake up at time $N-1$ and therefore the protocol will require $O(N)$ time units.

Protocol $\mathcal{B}$, however, satisfies the following two lemmas. In the following, when we say that a node wakes up, we will mean either spontaneously or on receiving a

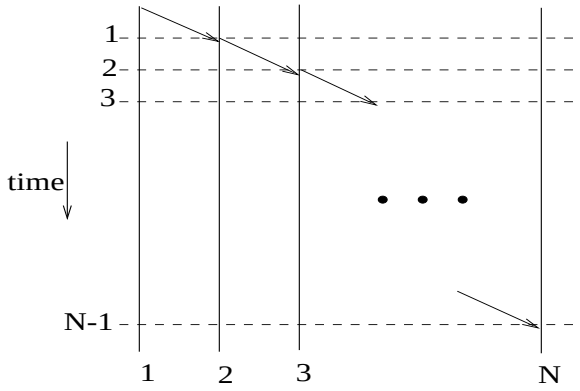


FIG. 4.1. *Example illustrating $O(N)$ time complexity.*

message of the protocol.

LEMMA 4.1. *If all nodes wake up within $O(N/\log N)$ time of each other, then $\mathcal{B}$ will terminate in $O(N/\log N)$ time.*

LEMMA 4.2. *After the level number of a node reaches $\log N$, $\mathcal{B}$ terminates within $O(N/\log N)$ time.*

Proof. There can be at most $N/\log N$ nodes at level $\log N$ (from Lemma 3.1). Let i be the first node to reach level $\log N$. Then in the next seven time units, i will either become *dormant* or increase its level number (because it takes at most seven time units to capture a node). Only a node at level $\log N$ can cause i to become dormant. If j causes i to become *dormant*, then by the same reasoning, in the next seven time units, j will either become *dormant* or increase its level number. Therefore, in every interval of seven time units after the first node reaches level $\log N$, the node with the highest $(level, id)$ at the beginning of the interval will either become *dormant* or increase its level number. Consider a sequence of $(2N/\log N) - \log N - 1$ slots in an execution after the first node reaches level $\log N$, where each slot is seven time units. At most $N/\log N - 1$ slots can be associated with nodes at level at least $\log N$ becoming *dormant*. Each of the remaining $N/\log N - \log N$ slots must correspond to a level increase. Therefore, $7((2N/\log N) - \log N - 1)$ time units after the first node reaches level $\log N$, the level number of a node will reach at least $N/\log N$. After a node reaches level $N/\log N$, Γ is executed, which takes $O(1)$ time. Hence $\mathcal{B}$ will terminate in $O(N/\log N)$ time units.

Given Lemma 4.2, we are led to the following protocol:

On waking up spontaneously, node i executes an initial phase in which it sends $\log N$ *capture*$(0, i)$ messages simultaneously on different edges. (The rules for capturing are as in $\mathcal{A}$.) If it receives an *accept* response to all *capture* messages and it has not been *killed* or *captured*, then it increases its level from 0 to $\log N$ (thus $level_i$ never has a value in between 0 and $\log N$) and proceeds with protocol $\mathcal{B}$.

Since each node can send $\log N$ *capture* messages on waking up spontaneously, the message complexity of this protocol is $O(N \log N)$. However, for this protocol, we can also construct an example similar to the one in Figure 4.1 in which a node reaches level $\log N$ after $N - \log N$ time units of the first node waking up. In this protocol, when node i tries to capture $\log N$ nodes in parallel, it is allowed to capture nodes that may have awakened at any time in the past. In the following, we develop a protocol $\mathcal{C}$ in which the nodes that i can capture when it is executing its initial phase

are restricted to those which wake up within a constant time of i waking up.

If $\log N$ nodes wake up every c time units, where c is a constant, then all nodes will wake up in $cN/\log N$ time units. Then from Lemma 4.1, the protocol will terminate in $O(N/\log N)$ time. Thus if we can ensure that

in every interval of c time units, either at least $\log N$ nodes wake up or some node reaches level at least $\log N$ by the end of this interval,

then from Lemmas 4.1 and 4.2, the protocol will require $O(N/\log N)$ time. For this purpose, we require a candidate node to execute two initial phases on waking up spontaneously. If it successfully executes these phases, then it qualifies as a candidate for election and proceeds by executing $\mathcal{B}$. In the first phase, a node requests permission to enter the second phase from $\log N$ different nodes. A node which has finished executing its first phase denies the request while others grant it. As we will show later, nodes which grant i permission to enter the second phase are those which wake up within c time units of i waking up. In the second phase, node i tries to capture the $\log N$ nodes from which it obtained permission in the first phase. The second phase ensures that if fewer than $\log N + 1$ nodes wake up in an interval of c time units (as a result, any new node that wakes up in this interval will not get permission to enter the second phase), then some candidate node reaches level $\log N$ by the end of this interval. For the initial two phases, the algorithm for a node j with $state_j =$ candidate is given in Figure 4.2, and the algorithm for a node j with $state_j \neq candidate$ is given in Figure 4.3. In Figure 4.3, for a response to the *capture* message, only those actions which correspond to the initial two phases are given. (In addition to these actions, any action required by protocol $\mathcal{B}$ must also be performed.)

First phase.

• On waking up spontaneously, node i enters the first phase. It selects $\log N$ incident edges and sends a $first\text{-}phase(i)$ message on each of these edges.

• When node j receives a $first\text{-}phase(i)$ message over e, it behaves as follows:

(1) If j is not captured, then

⋆ if j has finished executing the first phase, then j sends a $finish$ message over e;

⋆ if j is *passive*, then i becomes j's owner and j marks e as *owner-link*$_j$; it sends an *accept* message over e and changes its state to *captured*;

⋆ if j is in the first phase, then j sends a *proceed* message over e.

(2) If j is *captured*, then it has to check whether its owner has finished the first phase. For this purpose, it sends a *check* message over *owner-link*$_j$. If j has already sent a *check* message to which no response has been received, then the response to i is delayed.

If *owner*$_j$ has finished its first phase when the *check* message reaches it, then it responds with a $finish$ message. Otherwise, it responds with an *in-first-phase* message.

If j receives a $finish$ message over *owner-link*$_j$, then it sends a $finish$ message over e and also on any other edge over which it may have received a *first-phase* message in the meantime (or will receive one in the future). If j receives an *in-first-phase* message from its owner, then it sends a *proceed* message over e and also on any other edge over which it may have received a *first-phase* message in the meantime.

(3) If j has already received a *capture* message or receives a *capture* message after receiving the *first-phase* message from i (a *capture* message can be sent by a node which is in the second phase or has finished the second phase) and no response has been sent to i, then it sends a $finish$ message to i over e.

```
state := candidate; phase := 1;
select a set S of log N edges from untraversed;
delete S from untraversed;
send first-phase(j) over edges in S; Response := 0
while state = candidate and Response < log N
do
    receive M on edge e
    Case M of
    first-phase(i):     send proceed over e
    proceed:            Response := Response + 1
    accept:             Response := Response + 1; S = S − {e}
    finish:             state := killed
    check:              send in-first-phase over e
    capture(l, i):      state := captured; owner := i; owner_link := e;
                        maxid := i; phase := 2
                        send accept over e
od
if Response = log N then phase := 2;
if state = candidate then
    send capture(0, j) over edges in S; Response = 0
    while state = candidate and Response < |S|
    do
      receive M on edge e
      Case M of:
      capture(l, i):      if (l, i) > (0, j) then
                              state := captured; maxid := i; owner-link := e;
                              send accept over e
      kill-owner(l, i):   if (l, i) > (0, j) then
                              state := killed; send owner-accept over e
                          else send killed(i) over e
      accept:             Response := Response + 1
      first-phase(i):     send finish over e
      check:              send finish over e
    od
if state = candidate then level := log N
```

FIG. 4.2. *Initial phases for a candidate node j.*

• After node i has received responses to all $\log N$ *first-phase* messages, it behaves as follows: It exits the first phase. If it has received a *finish* message, then it does not enter the second phase and it changes *state* to *killed*. Otherwise, it proceeds to the second phase.

Second phase.

• In the second phase, node i tries to reach level $\log N$ by capturing $\log N$ nodes. For this purpose, it sends a *capture*$(0, i)$ message on each edge on which it received a *proceed* message in the first phase. (It has already captured nodes from which it received an *accept* message.) The rules for capturing are the same as in protocol $\mathcal{B}$

```
done := false; waiting := false;
while ¬done
do
    receive M on edge e
    if M is the first message then
                        state := passive; phase := 1;
    Case M of
    first-phase(i):     if state = passive then
                            state := captured; maxid := i; owner-link := e;
                            send accept over e
                        else if phase = 2 then send finish over e
                        else if state = captured then
                            if ¬waiting then
                                send check over owner-link;
                                waiting := true; Wait := {}
                            else WAIT := WAIT ∪ {e}
    finish:             if owner-link = e then
                            phase := 2;
                            send finish over each edge in WAIT; WAIT := {}
    in-first-phase:     send proceed over each edge in WAIT; WAIT := {}
    capture(l, i):      phase := 2;
                        send finish over each edge in WAIT; WAIT := {}
    proceed:            skip
    accept:             skip
    check:              if phase = 1 then send in-first-phase over e
                        else send finish over e
    elected:            done := true
od
```

FIG. 4.3. *Initial phases for a noncandidate node j.*

with the following changes:

A node which has not started the second phase is regarded as *passive*. Therefore, if a $capture(0, i)$ message reaches j and j is in the first phase or has finished the first phase but was not successful in entering the second phase, then i becomes j's owner. In case j is a captured node, then j sends a *kill-owner* message as in $\mathcal{B}$. If $owner_j$ has not started its second phase, then it is *killed*. Further, a node increases its level number from 0 to $\log N$ only after receiving an *accept* response to all *capture* messages.

Since the level number sent in the *capture* messages is 0, capturing in the second phase is only on the basis of node identities. If node i receives all *accept* responses and it is not *killed* or *captured*, then it sets $level_i$ to $\log N$ and proceeds with protocol $\mathcal{B}$. It deletes the $\log N$ edges from $untraversed_i$ over which it had sent the *first-phase* messages.

Hence in protocol $\mathcal{C}$, a node executes the first and second phases before executing $\mathcal{B}$. In order to show that $\mathcal{C}$ terminates with an elected leader, we have to show that some node will finish its second phase and participate in $\mathcal{B}$. Since nodes in $\mathcal{B}$ cannot be killed by nodes in the initial two phases (nodes in $\mathcal{B}$ have a higher level number)

and $\mathcal{B}$ does elect a leader, $\mathcal{C}$ also terminates with an elected leader. This is shown in the following two lemmas.

LEMMA 4.3. *In any execution of $\mathcal{C}$, there will be a node which enters the second phase.*

Proof. Assume that no node enters its second phase in an execution. Then every candidate node must have received a $finish$ message in response to at least one of its $first\text{-}phase$ messages. Let $i_1, i_2, \ldots$ be a sequence of candidate nodes such that i_l receives a $finish$ message from i_{l+1} or from one of the nodes captured by i_{l+1}. Node i_{l+1} (or a node captured by i_{l+1}) can send a $finish$ message only after i_{l+1} has finished its first phase. (The $finish$ message cannot be sent by i_{l+1} due to the fact that it received a *capture* message from some other node because a *capture* message can only be received from a node in the second phase or which has finished the second phase, which contradicts the assumption.) Therefore, the first phase of i_{l+1} terminates before the first phase of i_l. This is true for all l's, and therefore $i_k \neq i_m$, where $k < m$. This is a contradiction since the number of nodes is finite. Hence there will be a node in the sequence which does not receive a $finish$ message and which will therefore enter the second phase.

LEMMA 4.4. *In any execution of $\mathcal{C}$, there will be a node which finishes the second phase.*

Proof. By Lemma 4.3, there is a node which enters the second phase. Only a node in its second phase (or which has finished the second phase) can cause another node in its second phase to become *dormant*. Further, any two nodes in the second phase cannot both cause each other to become *dormant* (because only a node with a higher identity can cause a node to become dormant). Therefore, there will be a node which will finish its second phase and participate in $\mathcal{B}$.

LEMMA 4.5. *The time complexity of $\mathcal{C}$ is $O(N/\log N)$.*

Proof. A candidate node finishes its first phase within five time units of waking up. (The proof is similar to the proof of Lemma 3.2.) If node i wakes up spontaneously at time t, then for i to participate in the second phase, each of its $first\text{-}phase$ messages must go to a node which is in its first phase (this node must have awakened spontaneously after time $t-5$; otherwise it would have finished its first phase by time t) or is captured (in this case, its owner must have awakened after time $t-5$; otherwise it would have finished its first phase by time t) or is *passive* (this node will wake up by time $t+1$ as a result of i's message). Therefore, i is able to proceed to the second phase only if at least $\log N$ nodes other than i wake up in the interval $[t-5, t+1]$.

Consider an interval of five time units, say $[m, m+5]$, where $m \geq 0$, during the execution of the protocol. We have the following cases:

(1) At least $\log N + 1$ nodes wake up in the interval $[m-5, m+6]$.

(2) Fewer than $\log N + 1$ nodes wake up in the interval $[m-5, m+6]$. In this case, we will show that some node will reach level at least $\log N$ (i.e., some node will successfully finish the second phase). We will first show that there must exist a node which has entered the second phase by time $m+5$. On the contrary, assume that no such node exists. All nodes that wake up before time m finish the first phase by time $m+5$ (since it takes five time units to execute the first phase), and by assumption, none of these nodes enter the second phase. Furthermore, any node which wakes up in the interval $[m, m+5]$ will be denied permission to enter the second phase (since fewer than $\log N$ nodes wake up in the internal $[m-5, m+6]$). If in this execution no node wakes up spontaneously after time $m+5$, then there will not exist a node which enters the second phase, which is a contradiction (Lemma 4.4). Hence there

must exist a node which enters the second phase by time $m + 5$.

Among the nodes which enter the second phase, if no node has finished the second phase by time $m + 5$, then there must exist at least one node in the second phase at time $m + 5$. Let i be the node with the highest identity among the nodes which are in the second phase at time $m + 5$. Let a *capture* message from i reach a node j. If j is not a captured node, we have the following cases:

(a) If j has not started the second phase, then it will respond with an *accept* message.

(b) If j has started the second phase, then it must be the case that j entered the second phase at or before time $m + 5$. Assume not. Since nodes that wake up before time m enter the second phase before time $m + 5$ and nodes that wake up in the interval $[m, m + 5]$ do not participate in the second phase, j must have awakened after time $m + 5$. Because i sent a capture message to j in the second phase, it must have also sent a *first-phase* message to j. In this case, the *first-phase* message from i must have reached j before time $m + 5$. Since j is passive before time $m + 5$, j will be captured by i's message, and hence j cannot enter the second phase. Thus j must have entered the second phase at or before time $m + 5$. In this case, $j < i$ (by assumption), and therefore j will respond with an *accept* message.

If j is a captured node, then it will forward the message to its owner. If the $owner_j$ has not started the second phase, then it will send an *accept* message. Otherwise, $owner_j$ must have entered the second phase at or before time $m + 5$. Assume not. Since nodes that wake up before time m enter the second phase before time $m + 5$ and nodes that wake in the interval $[m, m + 5]$ do not participate in the second phase, $owner_j$ must have awakened after time $m + 5$. Because i sent a capture message to j in the second phase, it must have also sent a *first-phase* message to j. In this case, the *first-phase* message from i must have reached j before time $m + 5$. In this case, it is not possible for $owner_j$ to capture j. ($owner_j$ could not have captured j in its first phase because only a passive node can be captured in the first phase; if $owner_j$ captured j in its second phase, then it would have received a *proceed* message for $\log N$ nodes which awakened in the interval $[m - 5, m + 6]$, which is a contradiction.) Thus $owner_j$ must have entered the second phase at or before time $m + 5$. In this case, $owner_j$ will be killed and an *accept* message will be sent to i since $owner_j < i$ by assumption. Thus i will receive all *accept* responses, and therefore i will finish its second phase.

Hence in each interval of 11 time units ($[m-5,\ m + 6]$), either at least $\log N$ nodes wake up or some node will reach level $\log N$. Therefore, by time $11N/\log N$, either all nodes will be awake or some node will have reached level $\log N$. Then from Lemma 4.1 and Lemma 4.2, the protocol will terminate in $O(N/\log N)$ time units.

5. A family of protocols. In this section, we generalize protocol $\mathcal{C}$ to obtain a family of protocols, $D(k)$, where $\log N \leq k \leq N$. Informally, $D(k)$ is the same as $\mathcal{C}$ except that k is used instead of $\log N$. $D(\log N)$ is the same as $\mathcal{C}$. The changes required in $\mathcal{C}$ to obtain $D(k)$ are as follows:

In the first phase, node i sends k *first-phase* messages (instead of $\log N$). If it does not receive a *finish* message in response to any of the *first-phase* messages, then i enters the second phase. In the second phase, node i sends a *capture* message on all edges on which it received a *proceed* response in the first phase. If it receives an *accept* response to all *capture* messages, then it increases its level to k and proceeds with protocol $\mathcal{A}$. When the level number of i reaches N/k, it executes protocol Γ, i.e., it sends its identity in an *elect* message over all edges. If all responses are *accept*,

then it declares itself the leader.

The time complexity of $D(k)$ is $O(N/k)$. The execution of the first and second phases requires $O(Nk)$ messages. Since there can be at most k nodes at level N/k, the execution of protocol Γ after the level number of a node reaches N/k requires $O(Nk)$ messages. Hence the message complexity of the protocol is $O(Nk)$.

6. Extensions to the protocol. The execution time of $\mathcal{C}$ depends on its ability to wake up all nodes within $O(N/\log N)$ time units of each other. Let r be the number of candidate nodes. In the following, we present a modification of the protocol to achieve a time complexity of $O(\min(r, N/\log N) + \log N)$. In protocol $\mathcal{A}$, nodes are captured in a sequential manner. In [2], a protocol for synchronous networks was presented in which a node is allowed to capture many nodes in parallel. In particular, a node i in phase l tries to capture 2^l nodes in parallel by sending $capture(l, i)$ message over 2^l untraversed edges. Node i increases its level number from l to $l + 1$ only on receiving an *accept* response to all 2^l capture messages. The time complexity of this protocol of [2] is $\log N$ since a node has to go through $\log N$ phases. By employing this style of capturing, we can extend our protocol to obtain a protocol with $O(\min(r, N/\log N)+\log N)$ time. In this protocol, the $\min(r, N/\log N)$ factor is the time required to awaken all nodes (and is the complexity of the first two phases). Since at most r nodes wake up, the chain of messages shown in Figure 4.1 can be at most r units (and hence the node which is elected leader must wake up within r time units of the first node waking up). Thus if $r < N/\log N$, then it will take r time units in the first two phases; if $r \geq N/\log N$, then the first two phases ensure that some node will reach level $\log N$ within $O(N/\log N)$ time units. After the second phase, a node which is elected leader has to go through $\log N$ phases, where each phase requires a constant amount of time.

7. Conclusion. In this paper, we have presented a distributed algorithm for leader election in an asynchronous complete network. The protocol has $O(N \log N)$ message complexity and $O(N/\log N)$ time complexity. This is an improvement over existing protocols for the this problem. We further extended the result to obtain a family of protocols, $D(k)$, where $\log N \leq k \leq N$. The message and time complexities of $D(k)$ are $O(Nk)$ and $O(N/k)$, respectively.

REFERENCES

[1] Y. Afek and E. Gafni, *Simple and efficient distributed algorithms for election in complete networks*, in Proc. 22nd Annual Allerton Conference on Communication, Control and Computing, Allerton, IL, 1984, pp. 689–698.

[2] Y. Afek and E. Gafni, *Time and message bounds for election in synchronous and asynchronous complete networks*, SIAM J. Comput., 20 (1991), pp. 376–394.

[3] B. Awerbuch, *Optimal distributed algorithms for minimal weight spanning tree, counting, leader election and related problems*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 230–240.

[4] G. Frederickson and N. Lynch, *The impact of synchronous communication on the problem of electing a leader in a ring*, in Proc. ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 493–503.

[5] R. G. Gallager, P. A. Humblet, and P. M. Spira, *A distributed algorithm for minimal spanning tree*, ACM Trans. Programming Languages Systems, 30 (1983), pp. 66–77.

[6] D. Hirschberg and J. Sinclair, *Decentralized extrema-finding in circular configurations of processors*, Comm. Assoc. Comput. Mach., 23 (1980), pp. 627–628.

[7] E. Korach, S. Moran, and S. Zaks, *Optimal lower bounds for some distributed algorithms for a complete network of processors*, Theoret. Comput. Sci., 64 (1989), pp. 125–132.

[8] G. Singh, *Leader election in complete networks*, in Proc. ACM Symposium on Principles of Distributed Computing, ACM, New York, 1992, pp. 179–190.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 786–802, June 1997

012

THE COST OF DERANDOMIZATION: COMPUTABILITY OR COMPETITIVENESS*

XIAOTIE DENG† AND SANJEEV MAHAJAN‡

Abstract. Recently, much work has been done in game theory towards understanding the bounded rationality of players in infinite games. This requires the strategies of realistic players to be restricted to have bounded resources of reasoning. (See [H. Simon, *Decision and Organization*, North–Holland, Amsterdam, 1972, pp. 161–176] for an extensive discussion; also see [X. Deng and C. H. Papadimitriou, *Math. Oper. Res.*, 19 (1994), pp. 257–266], [C. Futia, *J. Math. Econom.*, 4 (1977), pp. 289–299], [V. Knoblauch, *Games Econom. Behav.*, 7 (1994), pp. 381–389], [E. Kalai and W. Stanford, *Econometria*, 56 (1988), pp. 397–410], [A. Neyman, *Econom. Lett.*, 19 (1985), pp. 227–229], and [C. H. Papadimitriou, *Game Theory Econom. Behav.*, 4 (1992), pp. 122–131].) In this paper, we discuss infinite two-person games, focusing on the case where our player follows a computable strategy and the adversary may use any strategy, which formulates the notion of computer against extremely formidable nature. In this context, we say that an infinite game is semicomputably determinate if either the adversary has a winning strategy or our player has a computable winning strategy.

We show that, whereas all open games are semicomputably determinate, there is a semicomputably indeterminate closed game. Since we want to prove an indeterminacy result for closed games and since the adversary's strategy set is uncountable and our player's strategy set is countable, our proof for the indeterminacy result requires a new diagonalization technique, which might be useful in other similar cases. Our study of semicomputable games was inspired by online computing problems. In this direction, we discuss several possible applications to derandomization in online computing, with the restriction that the strategies of our player should be computable.

We also study the power of randomization for the classical case where our player is allowed to play according to unrestricted strategies. An indeterminate game is obtained for which both players have a simple randomized winning strategy against all of the deterministic strategies of the opponent.

Key words. online algorithms, computability, game theory, derandomization

AMS subject classifications. 68Q99, 90D20, 04A10

PII. S0097539791202301

1. Introduction.

In an online problem, requests come one at a time and need to be served as soon as they come. A cost function is associated with serving each request, which depends on the history of requests and the serving strategy. The object is to minimize the total cost incurred while serving the request sequence. If the request sequence is known in advance, it can be served optimally. However, since the future requests are not known in advance, we cannot in general design a strategy which will optimally serve the requests. To measure the performance of a strategy σ, the concept of the competitive ratio was introduced [ST, KMRS, MMS], i.e., the worst-case ratio (over all finite request sequences) of the cost of serving a request sequence by σ (one may allow an extra addition constant) to the minimum cost of serving the

* Received by the editors July 22, 1991; accepted for publication (in revised form) July 18, 1995. This research was supported in party by a grant from the Natural Sciences and Engineering Research Council of Canada and the Advanced Systems Institute of British Columbia to Tiko Kameda. This paper is based on an earlier work published in *Proc. 23rd Annual ACM Symposium on Theory of Computing*, ACM, New York, 1991.

http://www.siam.org/journals/sicomp/26-3/20230.html

† School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada. Current address: Department of Computer Science, York University, North York, ON, M3J 1P3, Canada (deng@cs.yorku.ca). The research of this author was supported by an NSERC International Postdoctoral Fellowship.

‡ School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada (mahajan@mpi-sb.mpg.de).

same request sequence. A strategy of serving requests is called *α-competitive* if its competitive ratio is no more than α.

A request–answer game formulation is introduced to explore the power of randomization in online problems for finite games in [BBKTW] and for infinite games in [RiS]. The idea is that, since we defined the competitive ratio as a worst-case ratio, we can think of the requests as coming from an adversary. The adversary and our player alternate between giving requests and serving them. Since the adversary is assumed to be omnipotent and omniscient, we also assume that the adversary serves her own requests with the minimum cost. An infinite game can be described as an infinite tree on which two players make their moves in turn, starting at the root.

In this formulation, we have two distinguished players: Player I, *the adversary*; and Player II, *our player*. The adversary makes choices at even levels, starting at level 0, the root, and our player makes choices at odd levels. The set of all the infinite paths is partitioned into two subsets: the winning set for our player and the winning set of the adversary. A strategy for the adversary (our player) corresponds to a pruned tree from the original game tree on which each branching at even (odd) levels is pruned to allow at most one possible child. The resulting play for a given pair of strategies corresponds to a path in the game tree. A strategy for our player is a winning strategy if all of the paths in the corresponding pruned tree belong to the winning set of our player. The winning strategies for the adversary are defined similarly. A game is called *determinate* if one of the players has a winning strategy.

Denote by OC the cost of our player and by AC the adversary's cost. Then by the above discussion, a strategy of our game is *α-competitive* if there is a constant c such that the request–answer sequence generated by playing with any adversary has the property that $OC \leq \alpha \cdot AC + c$. Under this formulation, the winning paths of our player form a closed set under the Hausdorff topology, and hence the corresponding game is called a *closed* game. (A formal definition is introduced in the next section). We call this *type* I *competitiveness*. In [RaS], a more relaxed definition is introduced by requiring

$$\cup_i \cup_j \cap_{k \geq j} \{(x_0, x_1, \ldots) : OC(x_0, \ldots, x_k) - \alpha \cdot AC(x_0, \ldots, x_k) \leq i\},$$

which leads to a countable union of closed sets, called by an F_σ game. This case considers how our cost increases as the adversary cost grows unboundedly in each branch of the game tree. We call it *type* II *competitiveness*.

Since both closed games and F_σ games are determinate [GS], it follows that, when playing against an offline adaptive adversary, randomization does not yield more competitive algorithms [BBKTW, RaS]. In essence, it is shown in [RaS] that for any determinate game, *a randomized winning strategy guarantees a deterministic winning strategy.* In this sense, we can say that the *derandomization hypothesis* holds for online problems. Raghavan and Snir point out that although the randomized strategy may be computable, the deterministic strategy guaranteed by this theorem may not be computable [RaS].

To address this question, we formulate the *computable derandomization hypothesis: a computable randomized winning strategy guarantees a computable deterministic winning strategy.* Following the approach in [BBKTW, RaS], we want to know if closed games or F_σ games are all semicomputably determinate. If this is true for closed (F_σ) games, then type I (type II) competitive randomized algorithms would guarantee computable type I (type II) competitive deterministic algorithms. On the contrary, we show that, whereas all open games are semicomputably determinate,

there is a semicomputably indeterminate closed game. This opens the door for the possibility that there exists an online problem for which there is a computable randomized competitive strategy but there is no computable deterministic competitive strategy. Indeed, we are able to construct an F_σ game for which there is a simple randomized strategy which wins almost surely but there is no computable deterministic winning strategy. Thus the computable derandomization hypothesis does not hold, even for F_σ games. This confirms the belief in [RaS]. With a certain cost function assigned for the game, this shows that there is a simple randomized strategy which is type II competitive almost surely, but there is no computable type II competitive deterministic strategy. We can also extend the result, in a slightly weaker form, to closed games for the notion of type I competitiveness. Therefore, the computable derandomization hypothesis does not hold for online problems, neither under the notion of type I competitiveness nor under the notion of type II competitiveness.

In our discussion, on one hand, we take an extremely pessimistic view of nature: it is formulated as an offline adaptive adversary [BBKTW] with complete power. On the other hand, we restrict our player to computable strategies. Such situations occur when strategies are designed to make dynamic decisions with an unpredictable future. One case is in the task of designing online algorithms to serve an unknown sequence of future requests, such as paging [ST], server problems [MMS, KP], and metrical task systems [BLS]. Robot navigation in an unknown world [PY, BRS] and robot learning [RiS, DP2] are similar situations. In all of these cases, while we may assume that the adversary may have unlimited power, our player is restricted by the computing machinery that is available. In a not unrelated problem, Papadimitriou has discussed asymmetric players in two-person games in terms of computational complexity by formulating nature as an amiable indifferent adversary: a randomized adversary with equal probabilities on choices of its moves [Pa2]. Many authors have considered players with restricted computational power [DP1, Fu, KS, Ne, Si].

The study of situations dealing with unknown future events leads to the concept of *competitiveness*. An online algorithm is α-competitive if its cost on any finite sequence of requests is within a factor of α of the optimal cost (up to an additive constant) by an algorithm that knows all of the future requests [KMRS]. For one problem of this type, the k-server problem, a matched upper bound and lower bound of k are conjectured [MMS]. The conjecture remains unsolved in spite of many efforts, even though it is shown to be true on several particular metric spaces [CL, CKPV, MMS]. A recent result [KP] comes very close to resolving this problem (see also [FRR, Gro]). The above-mentioned result in [BBKTW, RaS] provides another approach to solving the conjecture. Thus if one can design a k-competitive randomized online algorithm for the k-server problem, the above conjecture is proven even though one may not be able to design a computable deterministic algorithm. This approach depends very much on the possibility of a k-competitive algorithm against the offline adaptive adversary. However, all known k-competitive randomized algorithms for the server problem are against an online adaptive adversary [RaS, CDRS]. The failure in resolving the conjecture for general metric space, however, may not be attributed to the conjecture being false. *The optimal strategy may simply be noncomputable.* Though one may doubt whether this is the case for this natural game of the k-server problem since there is a computable strategy conjectured to be optimal, it is always helpful to look into other possibilities. Indeed, there are known cases of naturally defined games for which the optimal response strategy of our player to an adversary strategy cannot be computable. A very interesting example is the well-

known prisoner's dilemma game [Kn]. This motivation provides another perspective to the formalism discussed in this paper. However, this does not exclude the possibility that for some games, computable optimal strategies are possible to construct. In fact, Ben-David et al. show several such cases for the request–answer game, which can be applied to the k-server problem.

In section 2, we will formally introduce necessary notations and related definitions. We will discuss semicomputable determinacy of infinite games in section 3. In section 4, we look into possible implications of our main result to the semicomputable derandomization hypothesis for games and for online computation problems. Section 5 contains a discussion on the derandomization hypothesis for indeterminate games, a classic case without the computability restriction on our player. Section 6 concludes the paper with remarks and some open problems.

2. Definitions and notation. An infinite game can be described as an infinite tree on which two players make their moves in turn, starting at the root. We distinguish the players by naming Player I as *the adversary* and Player II as *our player*. The adversary makes choices at even levels, starting at level 0, the root, and our player makes choices at odd levels. We also call a pair of two moves a *stage*; thus level $2n$ and $2n+1$ are classified as stage n, $n = 0, 1, \ldots$. For convenience, we restrict the choice space of each player's moves to be finite, though some results may also extend to infinite choice spaces. The set of all of the infinite paths is partitioned into two subsets A and B, where A is the winning set for our player and B is the winning set of the adversary. A strategy τ (σ) for the adversary (our player) corresponds to a pruned tree T_τ (T_σ) from the original game tree on which each branching at even (odd) levels is pruned to allow at most one possible child. The resulting play (σ, τ) for a given τ and a given σ corresponds to a path in the game tree. We call σ a *winning strategy* for our player if for every τ, the path (σ, τ) belongs to A. The winning strategies for the adversary are defined similarly. Finally the game is *determinate* if either player has a winning strategy.

We now define a topology on the set of all plays. A subset S of all the paths is defined as being *open* iff for every path $(a_0, a_1, \ldots) \in S$, there exists a number n such that for all $b_{n+1}, b_{n+2}, \ldots$,

$$(a_0, a_1, \ldots, a_n, b_{n+1}, b_{n+2}, \ldots) \in S.$$

Let $O(a_0, a_1, \ldots, a_n) = \{(a_0, a_1, \ldots, a_n, b_{n+1}, b_{n+2}, \ldots) : \text{ all } b_{n+1}, b_{n+2}, \ldots\}$ and call it an elementary open set. A set of paths is *closed* if it is the complement of an open set. An important feature of this topology is that an elementary open set is also closed [Mos].

A game is *open* (*closed*) if the winning set of our player is open (closed). In their classical paper on infinite games [GS], Gale and Stewart show that all open and closed games are determinate and that there exists a game that is indeterminate. Martin [Mar] further proves that all Borel games are determinate. (A game is *Borel* if the winning set of Player I (or II) is Borel under the topology defined above.) Observe that in classical infinite-game theory, there is no restriction on the strategies in terms of computability.

In [BBKTW], online problems are formulated as finite games, and in [RaS], an infinite-game formulation is given. Depending on the criteria of competitiveness, one may get different winning sets for an online problem. Let us denote by OC the cost of our player and by AC the adversary's cost. A simple α-competitiveness requirement is defined by $OC \leq \alpha \cdot AC$. We will call it the type I *competitive* condition. When the cost

function is accumulative, e.g., in the case of the server problem, type I competitiveness will define a closed game: the adversary wins iff the play reaches a node where the condition is violated. A slightly different definition is to allow an additive constant $c > 0$ and use $OC \leq \alpha \cdot AC + c$ instead. This is not much different. Raghavan and Snir [RaS] use a formulation which allows an arbitrary additive constant, which gives rise to a game with a winning set in F_σ (the countable union of closed sets):

$$\cup_i \cup_j \cap_{k \geq j} \{(x_0, x_1, \ldots) : OC(x_0, \ldots, x_k) - \alpha \cdot AC(x_0, \ldots, x_k) \leq i\}.$$

Thus infinite paths of constant costs are always in the winning set of our player. In section 1, this was defined as the *type* II *competitive* condition.

A randomized strategy for our player is a function that makes an assignment of probability to all the possible choices of our player depending on the position of the node on the game tree. We say a randomized strategy is *computable* if, firstly, the probability density function on the choice space is a computable function and, secondly, the rules for the assignment of probabilities for different nodes on the game tree are computable. As noted in [RaS, HT], a statement about a randomized algorithm is true if it is true against all adversary strategies. Thus, given a randomized strategy, we consider each adversary deterministic strategy τ and the induced probability distribution for the pruned tree T_τ. We specify a topology and a probability measure on the smallest σ-algebra generated by the topology by specifying them on all the basic open sets. A basic open set U is specified by a node x on T_τ such that it contains all of the paths passing through x and its probability measure is the probability that the randomized strategy reaches x. The measure is extended to all of the Borel sets in the topology by the standard method [CT]. When we specify an adversary strategy τ, a similar method is applied to define the conditional distribution on the pruned tree T_τ. Again, a randomized strategy is α-competitive almost surely iff for all of the pruned trees T_τ, it is α-competitive almost surely with respect to this probability distribution.

With the terminologies defined above, it is not difficult to construct a semicomputably indeterminate game if we drop the requirement that it is closed (for the purpose of applications to online problems.) We simply take the list of all of the computable strategies that each player can have: $\tau_1, \tau_2, \ldots, \tau_n, \ldots$ for the adversary and $\sigma_1, \sigma_2, \ldots, \sigma_n, \ldots$ for our player. Using a standard diagonalization technique, we construct the winning paths of our player and the adversary as follows. Initially, choose $\langle \sigma_1, \tau_1 \rangle$ to be B_1, denote $\tau_1 = \tau_{\sigma_1}$, and let $A_0 = \emptyset$. In general, suppose for $k \geq 1$ that B_k and A_{k-1} are constructed, with $B_k = \{\langle \sigma_i, \tau_{\sigma_i} \rangle : i = 1, 2, \ldots, k\}$, $A_{k-1} = \{\langle \sigma_{\tau_i}, \tau_i \rangle : i = 1, 2, \ldots, k-1\}$, and $B_k \cap A_{k-1} = \emptyset$. Let m to be the minimum index such that $\langle \sigma_m, \tau_k \rangle \notin B_k$. This is well defined since $\{\langle \sigma_n, \tau_k \rangle : n = 1, 2, \ldots\}$ is an infinite set but B_k is finite. Denote $\sigma_m = \sigma_{\tau_k}$ and set $A_k = A_{k-1} \cup \{\langle \sigma_m, \tau_k \rangle\}$. Similarly, let p to be the minimum index such that $\langle \sigma_{k+1}, \tau_p \rangle \notin A_k$. This is well defined since $\{\langle \sigma_{k+1}, \tau_n \rangle : n = 1, 2, \ldots\}$ is an infinite set but A_k is finite. Denote $\tau_p = \tau_{\sigma_{k+1}}$ and set $B_{k+1} = B_k \cup \{\langle \sigma_{k+1}, \tau_p \rangle\}$. Take $A = \cup_{n=1}^{\infty} A_n$ and $B = \cup_{n=1}^{\infty} B_n$. We have two subsets A and B of infinite paths of plays such that for each computable strategy σ of the adversary, there is a computable strategy τ_σ of our player with the path $\langle \sigma, \tau_\sigma \rangle \in A$; for each computable strategy τ of our player, there is a computable strategy σ_τ of the adversary with the path $\langle \sigma_\tau, \tau \rangle \in B$.

The above sets of A and B may not partition the set of all plays. We can put all of the other paths in the winning set of our player. This proves the following:

0. There exists an indeterminate game when both players are restricted to using only computable strategies.

Moreover, this is in fact a semicomputably indeterminate game.

$0'$. There exists a semicomputably indeterminate game.

With possible applications to online problems in mind, however, we want to study the semicomputable determinacy of F_σ games and closed games. Our main results are the following:

1. There is a semicomputably indeterminate closed game.

Since a closed game is also in F_σ (that is, games in which the winning set is a countable union of closed sets), this also means there is a semicomputably indeterminate F_σ game. In contrast, for open games, the following is true.

2. All open games are semicomputably determinate.

To study randomized strategies for infinite games, we also need to specify the proper probability distribution. Raghavan and Snir formally defined it through a probability distribution over the strategy space of our player [RaS]. When the adversary's strategy is specified, it induces a probability distribution over the pruned tree corresponding to the adversary strategy. Halpern and Tuttle [HT] had a similar idea for distributed systems. Thus a statement about a randomized algorithm is true iff it is true for all the pruned trees. We prove that the computable derandomization hypothesis is not always true, even for F_σ games.

3. There exists an F_σ game
   - which is semicomputably indeterminate and
   - in which our player has a simple randomized computable strategy that wins all most surely.

We can also assign a cost function on paths of this game to transform it into an online problem to obtain the following result.

$3'$. There exists an F_σ game on which we can define an online problem such that our player has a randomized computable strategy which is type II competitive almost surely but has no deterministic computable type II competitive strategy.

Similarly, we can construct a closed game and get a slightly weaker result for type I competitiveness.

$3''$. For any $\epsilon > 0$, there is a closed game on which we can define an online problem such that our player has a randomized computable strategy which is type I competitive with probability $1 - \epsilon$ but has no deterministic computable type I competitive strategy.

The above result for type II competitiveness also holds when we use the expected value instead of high probability. A weaker result holds for the type I competitiveness. When none of the players is restricted in computational power, we show that the derandomization hypothesis is not true in general for indeterminate games.

4. There exists an indeterminate game for which both players have a randomized strategy which wins against all of the deterministic strategies of the other player.

3. Semicomputable determinacy of infinite games. While all Borel games are determinate [Mar], we would like to know under what topological conditions a game is semicomputably determinate. First, we have the following.

THEOREM 1. *There is a semicomputably indeterminate closed game.*

We give both players two choices of actions: $r0$ and $r1$ for the adversary and $a0$ and $a1$ for our player. We first give some intuition on the proof of the theorem. We need to partition the set of all of the paths into two sets A (the winning set for our player, which is closed) and B (the winning set for the adversary) such that for each computable strategy σ of our player, there exists an adversary strategy τ such that (σ, τ) is in B (call it condition C1), and for each adversary strategy τ of the adversary, there exists a computable strategy of our player σ such that (σ, τ) is

in A (call it condition C2). C1 and C2 force certain plays to be put in A and B, respectively, and we should make sure that (A, B) is a partition. Moreover, we want a construction that makes A a closed set.

Observe that the indeterminacy proof given in [GS] cannot be translated into this case. Our result is obtained via a new method which may be useful in other similar situations.

We construct A and B in stages. Let the computable strategies of our player be ordered as $\sigma_i, i = 0, 1, 2, \ldots$. Say that a strategy σ is *killed* in stage j if we put $(\sigma, \tau') \in B$ for some τ' of the adversary in B in stage j (and similarly for an adversary strategy). At each stage, we kill at least one σ and perhaps an uncountable number of τ's so that A and B remain disjoint, and we make sure that each σ and each τ is killed in some finite stage without destroying the disjointness criterion. An indeterminate game is thus constructed. The construction will guarantee that A constructed thusly is closed. We now give the technical details of the result.

Proof of Theorem 1. For simplicity, we assume that the adversary's choice space is $r0$, $r1$ and our player's choice space is $a0$, $a1$. We list all (computable) strategies of our player in the set

$$\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots\}$$

such that σ_0 is the strategy that chooses move $a0$ all the time. Informally, we need to construct a game with winning sets A for our player and B for the adversary such that the following conditions hold.

C1. For each $\sigma \in \Sigma$, there is $\tau \in \mathcal{T}$ and $(\sigma, \tau) \in B$.

C2. For each $\tau \in \mathcal{T}$, there is $\sigma \in \Sigma$ and $(\sigma, \tau) \in A$.

Construction of A, B. Initially, set A_0 contain all of the paths along which the adversary never requests $r1$. $B_0 = \emptyset$. Let $A = A_0$ and $B = B_0$. Denote the root of the game tree as being level 0. Incrementally assign level numbers to the tree. We will prune the tree in levels. First, level 0 is processed, and then we show inductively how to process level n for each $n = 1, 2, 3, \ldots$.

Level 0. Denote by $\mathcal{T}_1$ all of the adversary strategies that make $r1$ as their first request. Choose some $\tau_0 \in \mathcal{T}_1$ and assign the path $\langle \sigma_0, \tau_0 \rangle$ in B_1. Moreover, we make B_1 an elementary open set containing the path $\langle \sigma_0, \tau_0 \rangle$. To be specific, from the root on the path $\langle \sigma_0, \tau_0 \rangle$, we take the $(2K + 1)$-st node N_1, for some $K > 1$, and make B_1 contain all of the infinite paths that pass N_1. Assign all of the other paths that start with $r1$ to A_1. Update $A \leftarrow A \cup A_1$ and $B \leftarrow B \cup B_1$. Thus condition C1 holds for σ_0 and condition C2 holds for all $\tau \in \mathcal{T}_1$. Remove σ_0 from the strategy set of our player. Now we consider all of the paths that start with $r0$. At level 1, according to the choice of our player, the strategy set for our player is partitioned into two subsets Σ_0 and Σ_1, where the lists for Σ_0 and Σ_1 keep the same order as the list in Σ.

Level $2i - 1$, $i \geq 1$. We keep the following inductive assumption for the pruning process at the end of level $2i - 2$. Each remaining node at level $2i - 1$ is a descendent of the adversary that played $r0$ at all of the past i requests. Thus each node can be denoted by an i-bit binary number that corresponds to plays made by our player from the root to the node. At node j, Σ_j represents all of our player strategies that are consistent with j up to this node. Σ_j's, $j = 0^i, 0^{i-1}1, 0^{i-2}10, \ldots, 1^i$, form a partition of the remaining members in Σ which do not yet satisfy condition C1. All of the adversary strategies that remain at node j are those which make i consecutive requests of $r0$'s when played against our player which answers j correspondingly. We will denote them by $\mathcal{T}_j$. From the above discussion for levels 0 and 1, this holds for $i = 1$.

Level $2i$. Consider each node independently. Without loss of generality, let us look at node 0^i. Let

$$\Sigma_{0^i} = \{\sigma_{0^i{}_0}, \sigma_{0^i{}_1}, \ldots, \sigma_{0^i{}_n}, \ldots\}.$$

Denote by $\mathcal{T}_{0^i1}$ all of the strategies in $\mathcal{T}_{0^i}$ that make the $(i+1)$st request as $r1$. We assign the path $\langle \sigma_{0^i{}_0}, \tau_{0^i{}_0}\rangle$ to the set B_{0^i1} for some $\tau_{0^i{}_0} \in \mathcal{T}_{0^i1}$. Moreover, we make B_{0^i1} an elementary open set containing the path $\langle \sigma_{0^i{}_0}, \tau_{0^i{}_0}\rangle$. To be specific, from the node j on the path $\langle \sigma_{0^i{}_0}, \tau_{0^i{}_0}\rangle$, we take the $(2K+1)$st node N_{0^i1}, for the same $K > 1$ as in level 0, from the first $r1$ request and make B_{0^i1} contain all of the paths that pass N_{0^i1}. All of the other paths that start from 0^i and continue with $r1$ are assigned to A_{0^i1}. At level $2i+1$, according to the choice of our player, the strategy set for our player is partitioned into two subsets Σ_{0^i0} and Σ_{0^i1}, where the lists for Σ_{0^i0} and Σ_{0^i1} keep the same order as the list in Σ. Thus condition C1 holds for $\sigma_{0^i{}_0}$ and condition C2 holds for all $\tau \in \mathcal{T}_{0^i1}$. We also perform similar operations on all of the nodes j of i bits. For all j nodes of i bits, condition C1 holds for σ_{j_0} and condition C2 holds for all $\tau \in \mathcal{T}_{j1}$. Update the set A and B by assigning $A \leftarrow A \cup \bigcup_{j=0^i}^{1^i} A_{j1}$ and $B \leftarrow B \cup \bigcup_{j=0^i}^{1^i} B_{j1}$. It follows from these definitions that the inductive hypothesis holds for level $(2i+1)$.

Correctness proof. We now prove that conditions C1 and C2 are true for all adversary strategies and our player strategies. Notice that our player's strategies are first enumerated in the set Σ, and the ordering is kept when they are partitioned at each level. For the first strategy σ in Σ_j, there is an adversary strategy τ such that $(\sigma, \tau) \in B$ according to our pruning process. Therefore, for each $i = 1, 2, \ldots$, σ_i satisfies condition C1 no later than at level $2i-1$ in our construction. To prove that condition C2 holds for all adversary strategies, we consider two cases: one is the case where the adversary plays $r0$ all the time; the other is the case where the adversary plays $r1$ at least once for some strategy. The first case is done by the initial assignment of set A. For the second case, we notice that for any other strategy τ of the adversary, it will play $r1$ at least once for a strategy of our player at a finite level. If the strategy of our player is not a computable strategy, we can simply truncate the infinite strategy at that finite level and append it by always playing $a0$. This will be a computable strategy $\sigma(\tau)$. Suppose j is the node for the first step where the adversary plays an $r1$; then the adversary strategy τ will lose to $\sigma(\tau)$ at one path in $\mathcal{T}_{j1}$.

To show that A is a closed set, we notice that B_1 is an open set and so are all the B_{j1}'s. Thus B is open since it is the union of a countably many open sets. A and B partition the whole space; therefore, A is closed. □

In contrast, all open games with a finite choice space for the adversary have enough mathematical structure to make them semicomputably determinate. To prove this, we need the following lemma (see, e.g., [Ku].)

LEMMA 2 (Koenig). *Every finitely branching infinite tree has an infinite path.* □

THEOREM 3. *All open games are semicomputably determinate if the choice space for the adversary is finite.*

Proof. Suppose no adversary strategy wins over all strategies of our player. Since open games are determinate [GS, Mar], there is a strategy σ of our player which wins against all of the adversary strategies (though σ may not be computable). Consider the pruned tree T_σ. Every infinite path in T_σ is in the winning set A of our player. Since A is open, for each infinite path in T_σ, there is a node x on the path such that all the paths passing through x are in A. We can thus remove the branch from x

down and make x a terminal node on which our player wins. This will not change the win/lose situation of the tree since all of the paths that pass x are in A. The game tree thus pruned has no infinite path. By Lemma 2, since this pruned tree is finitely branching and has no infinite paths, it is finite. Therefore, if there is no adversary winning strategy, our player can win by simply coding the structure of the pruned *finite* tree and choosing its moves accordingly. □

By exchanging the role of our player and the adversary, we can further obtain the following corollary.

COROLLARY 4. *For closed games, there is either a finite-state adversary strategy that wins against all strategies of our player or a strategy of our player that wins against all adversary strategies if the choice spaces for both the adversary and our player are finite.*

For the notion of type I competitiveness, the winning set of our player is closed. If there is no deterministic winning strategy of our player, then the winning strategy of the adversary will enable us to prune the tree to a finite tree according to Corollary 4. We thus easily conclude that there is no competitive randomized strategy for our player. The result of [BBKTW, RaS] for infinite games follows immediately. Corollary 4 also implies that if we allow our player to use unlimited power, we only need to look for a lower bound forced by adversaries with a simple computational power: finite-state machines. Thus it is easier to establish a general lower bound than a lower bound tailored to computable strategies of our player.

4. Applications to online algorithms. While online problems are formulated as closed and F_σ games, we would also like to formulate closed and F_σ games as online problems such that there is a winning strategy for our player in a given game iff there is an α-competitive online algorithm for the corresponding online problem. This may not be true in general. However, for semicomputable indeterminate games constructed in this paper, we want to make sure that the above condition is satisfied. First, we construct a game similar to the one given in section 3 for this goal.

THEOREM 5. *There exists a semicomputably indeterminate F_σ game such that there is a computable randomized winning strategy (almost surely). Moreover, on this game, we can define an online problem of accumulative cost such that there is a computable randomized type* II *competitive strategy (almost surely) but there is no computable deterministic type* II *competitive strategy.*

An F_σ semicomputably indeterminate game. We construct a game similar to the one for proving Theorem 1. The changes are as follows. We choose A_0 in the same way as we did in Theorem 1. Notice that A_0 is closed because it can be obtained by removing countably many disjoint elementary open sets as follows:

for $n = 0$ **to** ∞ **do** remove the $r1$ branch for level $2n$.

Actually, A_0 is a Cantor-type set (see, e.g., [Bi]).

In level 0, we will put (σ_0, τ_0) in B_1, instead of an elementary open set containing the path, and all the other paths starting with $r1$ are put into A_1. Similarly, at node j, we also put one path $\langle \sigma_{j_0}, \tau_{j_0} \rangle$ in B_{j1}, and all of the other paths that start at node j and continue with $r1$ are put in A_{j1}. The rest of the construction follows the same pattern as before. Notice that A_{j1} consists of countably many elementary open sets. However, as noticed while defining the topology in section 2, an elementary open set is also closed. Thus each A_{j1} is an F_σ set. Therefore, A is also an F_σ set.

Although A is no longer closed, the proof that the game is semicomputably indeterminate follows from the same reason as in Theorem 1. The only differences are

assignments of paths that are irrelevant to the proof of semicomputably indeterminacy.

A simple randomized winning strategy. Consider the randomized algorithm that always chooses $a0$ and $a1$ with probability 0.5:0.5. We claim that this simple (computable) randomized algorithm wins almost surely.

To discuss the proof carefully, for any adversary strategy τ, consider its corresponding pruned tree T_τ. For the root level, if the adversary chooses $r1$, there is exactly one winning path of the adversary in its strategy tree T_τ. We can remove all of the nodes below this node to reduce this tree to one root node, leading to one leaf node through an edge labeled $r1$. On the collection of paths from this leaf node, our player wins with probability 1.

In general, our player wins with probability 1 in each branch on a node reached from the root by a consecutive sequence of $r0$ requests followed by the first $r1$ request. With this observation, we further prune the tree T_τ as follows. Move from the root down the tree T_τ until the first request $r1$ is encountered. Then delete the branch after that node. Thus infinite paths of the further pruned tree will all contain request $r0$ only. They are all in the winning set of our player. On all of the leaves created by the above pruning procedure, our player wins with probability 1. It follows immediately that this simple randomized strategy wins with probability 1 against any adversary strategy.

A cost function. We want to have an accumulative cost function such that it increases smoothly as plays proceed. All nodes at even levels are assigned a cost of 0. Thus in the following, all of the nodes assigned costs are nodes at odd levels. On the winning paths of the adversary in the game tree, we assign the cost of each node to be 1 until the first $r1$ is encountered. After the first $r1$ request, the cost of each node on the winning paths of the adversary in the game tree is its level minus the level of the node the first $r1$ request is made. At all other nodes, a cost of 1 is assigned. The accumulative cost (informally, the online cost) of our player along a path will be the sum of the costs of nodes it has passed on the game tree. The cost of the adversary (informally, the offline cost) will be the minimum cost over all the paths with the same request sequence. That is, we consider an offline adaptive adversary [BBKTW]. Therefore, for a fixed request sequence, the offline cost grows linearly with the length of the request sequence. Along a winning path of the adversary with at least one $r1$ request, the growth rate of the online cost is asymptotically quadratic in the length of the request sequence. Along all other paths, the growth rate of the online cost is asymptotically linear with the length of the request sequence.

Competitiveness. Denote an infinite path by $\langle \tau, \sigma \rangle$. We will denote by $\langle \tau, \sigma \rangle_k$ the subpath consisting of the first k edges in the infinite path $\langle \tau, \sigma \rangle$. From the above game construction and the cost function, we have the following:

(a) For any $\langle \tau, \sigma \rangle$,

$$AC(\langle \tau, \sigma \rangle_k) = \lceil k/2 \rceil.$$

This holds since we can always obtain a subpath with any given sequence of $\lceil k/2 \rceil$ requests which does not shares any edge with an adversary winning path except probably the first one. In either case, the cost on each odd-level node is 1. The total cost is thus $\lceil k/2 \rceil$.

(b) For any $\langle \tau, \sigma \rangle \in A$, there is a constant c such that

$$\lim_{k \to \infty} [OC(\langle \tau, \sigma \rangle_k) - \lceil k/2 \rceil] \leq c.$$

Suppose $\langle \tau, \sigma \rangle$ is not in the adversary winning set. Let f be the number of edges that it shares with an adversary winning path after the first $r1$ request. Then the cumulative cost along path P at a node of level $k = 2j$ is bounded by $j + (1 + 3 + \cdots + f) = j + (f+1)^2/4$. The offline cost is at least j. Therefore, $OC(\langle \tau, \sigma \rangle) - \lceil k/2 \rceil \leq (f+1)^2/4$, for all $k \geq 0$. The claim follows by choosing $c = (f+1)^2/4$.

(c) For any $\langle \tau, \sigma \rangle \in B$ and for any constant c,

$$\lim_{k \to \infty} [AC(\langle \tau, \sigma \rangle_k) - c\lceil k/2 \rceil] = \infty.$$

Let the first $r1$ request on $\langle \tau, \sigma \rangle$ is at level $2j$. Then the cost at level $2i+1$ is 1, if $0 \leq i < j$ and is $2(i-j)+1$ if $i \geq j$. Therefore, the cumulative cost at level $k = 2i+1$, or $2i+2$, for $i \geq j$, is $j + \sum_{t=j}^{i}(2(t-j)+1)$, which is $\Omega(k^2)$ as k goes to infinity. Then it cannot be bounded by any linear function. The claim follows.

From the construction of the game, for any deterministic strategy σ_s, there exists a τ_s such that $\langle \tau_s, \sigma_s \rangle \in B$. Therefore, σ_s cannot have any constant competitive ratio by (c). On the other hand, for any adversary strategy τ, at most one path of T_τ is in B, which is taken by our randomized strategy with probability 0. By (b), the randomized strategy has a constant competitive ratio on all the paths in B, which holds with probability 1. Theorem 5 follows. Notice that the value f and thus c in (b) depends on the particular strategy T_τ. It follows that the competitive ratio on any infinite path in the winning set of our player is one according to the definition of type II. □

With regard to the issue of derandomization for type I competitiveness, one may insist that we compare randomized competitive algorithms with type I deterministic competitive ones. This would lead to a closed game. In this case, we can use the closed game constructed for Theorem 1 and use the same cost function as defined above.

COROLLARY 6. *For any integer $K > 0$, there exists a closed game for which one can define an online problem such that there is no computable deterministic type* I *competitive strategy but there is a computable randomized strategy that is type* I *K-competitive with probability $1 - 2^{-K}$.*

Proof. For simplicity, we first consider adversary strategies that make $r1$ the first request. The cost as defined above of each node for the tree T_τ will be as follows:

1. All nodes at even levels have cost 0.

2. There is only one node at level $l = 2j+1$ with cost $2j+1$ for $j \leq K$, which is the node on the winning path of the adversary; all other nodes on level $l = 2j+1$ have cost 1 for $j \leq K$.

3. There are only 2^{j-K} nodes at level $l = 2j+1$, with cost $2j+1$ ($j \geq K$); all other nodes on level $l = 2j+1$ have cost 1.

Since nodes at even levels will have the same cumulative cost as their parents, we focus our discussion on nodes at odd levels. Therefore, the cumulative cost of nodes of the pruned tree T_τ at level $k = 2j+1$ ($j \geq K$) will be as follows:

1. 2^{j-K} nodes of cumulative cost $1+3+\cdots+(2j-3)+(2j-1)+(2j+1) = (j+1)^2$;

2. 2^{j-K} nodes of cumulative cost $1 + 2 \cdots + (2K-1) + \sum_{t=K}^{j} 1 = K^2 + (j-K)$;

3. 2^{j-K+1} nodes of cumulative cost $1 + 2 + \cdots + (2K-3) + \sum_{t=K-1}^{j} = (K-1)^2 + (j-K+1)$;

...

($i+2$). 2^{j-K+i} nodes of accumulative cost $(K-i)^2 + j - K + i$, for $i : 1 \leq i < K$,

...

$(K+1)$. 2^{j-1} nodes of accumulative cost $j+1$.

Therefore, for any deterministic strategy σ_s of our player, there exists an adversary strategy τ_s such that $(\langle \tau_s, \sigma_s \rangle)$ is in the winning set of the adversary. From the above, we have $OC(\langle \tau_s, \sigma_s \rangle_k) = (\lceil k/2 \rceil)^2$. Similarly to the proof of Theorem 5, $AC(\langle \tau, \sigma \rangle_k) = \lceil k/2 \rceil$. σ_s cannot be type I competitive.

The fraction of nodes with cumulative cost $(j+1)^2$ at level $2j+1$ is 2^{-K}. The cumulative cost for any other node at level $2j+1$ is no more than K^2+j-K, which grows linearly with j as j goes to infinity. On these paths, we have $OC(\langle \tau, \sigma \rangle_k) - 2 \cdot AC(\langle \tau, \sigma \rangle_k) \leq K^2 - K - j$. Since $K^2 - K - j \leq 0$ as $j \to \infty$, choosing $c = 0$, we have that $\lim_{k\to\infty}[OC(\langle \tau, \sigma \rangle_k) - 2 \cdot AC(\langle \tau, \sigma \rangle_k)] \leq c$ holds with probability $1 - 2^{-K}$.

If the adversary does not make $r1$ its first request, the analysis is similar and the same claim holds for j greater than K plus the number of requests before the first $r1$. □

The above results are for randomized algorithms that are competitive with high probability. For randomized algorithms that are competitive in expected values, we have a similar result for type II competitiveness and a slightly weaker result for type I competitiveness.

COROLLARY 7. *There exists a game on which one can define an online problem such that there is no computable deterministic type* II *competitive strategy* (*no computable deterministic type I* $(2^K - 1)$*-competitive strategy*) *but there is a computable randomized strategy which is type* II *competitive strategy in the expected value* (*a computable randomized type* I $(K+2)$*-competitive strategy*).

Proof. For the type II competitiveness, we use the same game and the same cost function as defined in the proof of Theorem 5. In fact, one may conclude that the claim holds in the expected value by verifying conditions for the *monotone convergence theorem* or *Fatou's lemma* [CT] in this case. For completeness, we present a direct proof. Suppose our player faces an adversary τ which makes its first move $r1$. The cost as defined before of each node for the tree T_τ will be as follows:

1. All nodes at even levels have cost zero.

2. There is only one node at level $l = 2j+1$ with cost $2j+1$ $(j \geq 1)$, which is the node on the winning path of the adversary; all other nodes on level $l = 2j+1$ have cost 1.

Therefore, the cumulative cost on nodes of the pruned tree T_τ at level $l = 2j+1$ $(j \geq 1)$ will be as follows:

1. one node of cumulative cost $1+3+\cdots+(2j-3)+(2j-1)+(2j+1) = (j+1)^2$;
2. one node of cumulative cost $1+3\cdots+(2j-3)+(2j-1)+1 = j^2+1$;
3. two nodes of cumulative cost $1+3+\cdots+(2j-3)+1+1 = (j-1)^2+2$;

...

$(i+1)$. 2^{i-1} nodes of cumulative cost $(j-i+1)^2+i$, for $i < j$;

...

$(j+1)$. 2^{j-1} nodes of cumulative cost $j+1$.

Therefore, there are $2^{j-\sqrt{j}+1}$ nodes in level $l = 2j+1$ with cumulative cost at least $(j-j+\sqrt{j})^2 + j - \sqrt{j} + 1$. In other words, at level $l = 2j+1$, with probability at most $2^{-\sqrt{j}}$, the randomized strategy has cumulative cost at least $2j - \sqrt{j} + 1$ (but less than or equal to $(j+1)^2$). Therefore, the expected value is bounded by $(j+1)^2 2^{-\sqrt{j}} + (2j - \sqrt{j} + 1)$, which is dominated by the first term $2j$ as $j \to \infty$. Since the optimal cost is j, the claim holds in this case.

However, in general, the adversary may start with the first request $r0$. We prune T_τ further as follows. Start from the root until a request $r1$ is encountered, delete the

branch after that node, and mark this node. Thus the only infinite paths of the newly pruned tree will contain only request $r0$. We may, however, have an infinite number of marked nodes. Now consider all of the nodes at level $l = 2j + 1$. For each marked node v at level $2j_1$, let $j_2 = j - j_1$. From the above discussion, the cumulative cost at a leaf node (of level $2j + 1$) in the subtree rooted at v is at most $j_1 + (j_2 + 1)^2$ according to our cost function. Thus if $j_2 \leq \sqrt{j}$, this is bounded by $2j + 1$; we again have a competitive ratio bounded by 2. However, if $j_2 \geq \sqrt{j}$, the cumulative cost at a leaf node (of level $2j + 1$) in the subtree rooted at v is at most $j_1 + 2j_2 + 1 \leq 2j$ with probability at least $1 - 2^{-\sqrt{j_2}-1}$; the cumulative cost for remaining leaf nodes (of level $2j+1$) in the subtree rooted at v is at most is $(j+1)^2$ (with probability at most $2^{-\sqrt{j_2}-1}$). The competitive ratio of 2 again holds. For nodes at level $2j + 1$ with no ancestors marked, their competitive ratio is 1. The claim for type II competitiveness follows.

For type I competitiveness, we use the game constructed in Theorem 1. We need a special cost function. Again, we make all of the nodes at even levels cost 0 and discuss nodes at odd levels only.

1. For the adversary strategies that start with request $r1$, the construction chooses τ_0 according to σ_0 and takes the $(2K + 1)$st node N_1, $K > 1$, on the path $\langle \sigma_0, \tau_0 \rangle$, making all of the infinite paths that pass N_1 winning paths of the adversary. Along the path from the root to N_1, we assign the first node after the first $r1$ request a cost of $2^0 = 1$, the third node a cost of 2, ..., and the $(2i + 1)$th node at odd levels a cost of 2^i, $0 \leq i \leq K$. Thus the node N_1 is assigned a cost of 2^K. For any other node w in the branch that started with the $r1$ request at the root, its cost is assigned to be the maximum cost of nodes that we meet when moving down from the root to w. Thus at level $l = 2j + 1$ for $j \leq K$, there are one node of cost 2^j, one node of cost 2^{j-1}, two nodes of cost $2^{j-2}, \ldots, 2^{i-1}$ nodes of cost $2^{j-i}, \ldots, 2^{j-1}$ nodes of costs 1. Let S_j be the sum of the cost at level $2j + 1$. $S_j = 2^j + j2^{j-1} = (j + 2)2^{j-1}$ for $j \leq K$. Let E_j be the expected cumulative cost at level $2j + 1$. Thus E_j $(j \leq K)$ of the simple randomized algorithm is $1/2^j$ times $\sum_{i=0}^{j} 2^{j-i} \cdot S_i$ since each node at level $2i + 1$ is used in 2^{j-i} paths leading to level $(2j + 1)$. Thus the expected cost is $(j + 4)(j + 1)/4$ for level $2j + 1$ $(j \leq K)$. This is $(K + 4)(K + 1)/4$ at level $2K + 1$ while its optimal cost is $K + 1$. The competitive ratio is $K + 4/4$. Then $S_{j+1} = 2S_j$ and $E_j 2^j + S_{j+1} = E_{j+1} 2^{j+1}$. Thus $E_{j+1} = E_j/2 + S_{j+1}/2^{j+1}$ for $j \geq K$. As j goes to infinity, E_j goes to $2 \cdot S_K/2^K = 2 \cdot (K + 2)2^{K-1}/2^K = K + 2$. Therefore, the competitive ratio is $K + 2$.

2. For adversary strategies that start with $r0$, we assign a cost of 1 to a node from the root down until the first $r1$ request is reached. Then we will use the same cost function from the $r1$ request on. The same competitive ratio $(K + 2)$ holds for the simple randomized algorithm.

On the other hand, any computable strategy will results in a path with all nodes after a certain node costing 2^K, while an optimal path with the same request sequence will have all nodes costing 1. Thus the competitive ratio cannot be better than $2^K - 1$. □

5. Indeterminacy and randomization. In this section, we digress from our requirement that our player be restricted to a computable strategy and discuss the power of randomization in classical indeterminate games.

The result of [BBKTW, RaS] says basically that for a determinate game, whenever there is a randomized strategy for Player I which wins with probability 1, there is a deterministic winning strategy for Player I. If this result can be extended to inde-

terminate games, it means that for every indeterminate game, there is no randomized winning strategy for any of the players. The following theorem gives a negative answer to this question. Moreover, the game constructed for our theorem has another counterintuitive implication. Even though Player I has a randomized strategy that wins almost surely against all of the deterministic strategies of Player II, that randomized strategy does not necessarily win almost surely against all of the randomized strategies of Player II.

THEOREM 8. *Assuming the axiom of choice and the continuum hypothesis, there is an indeterminate game for which Player* I *has a randomized winning strategy which wins almost surely against any deterministic strategy of Player* II *and vice versa.*

The theorem shows that the result of [BBKTW, RaS] is the best possible in the sense that if we drop the condition that the game be determinate, the derandomization hypothesis does not always hold. This result assumes the axiom of choice and the continuum hypothesis. The axiom of choice seems necessary here because it is not even known if indeterminate games exist in its absence.

Proof of Theorem 8. We assume that each player has two choices at any point in the game. By the axiom of choice, we can well order Player I's deterministic strategies as σ_α for $\alpha < 2^{\aleph_0}$ and Player II's deterministic strategies as τ_β for $\beta < 2^{\aleph_0}$.

Our randomized strategy for either Player I or II (δ and γ, respectively) assigns a probability of 0.5 to each of the two possible moves. We now construct the winning set for Player I and the winning set for Player II so that both of these strategies are winning strategies if the other player uses only deterministic strategies.

We need to satisfy the following conditions:

1. For each deterministic strategy σ of Player I, only countably many paths in the pruned tree corresponding to σ belong to the winning set of Player I, and the rest belong to the winning set of Player II.

2. For each deterministic strategy τ of Player II, only countably many paths in the pruned tree corresponding to τ can belong to the winning set of Player II.

It is clear that if we can satisfy these conditions, then δ wins with probability 1 against any deterministic strategy of Player II, and γ wins with probability 1 against any deterministic strategy of Player I. (Each path has probability measure 0, and by the countable additivity of probability measure, countably many such paths will have measure 0.)

We say a deterministic strategy σ is *killed* if we can satisfy condition 1 for this σ (define killing of τ symmetrically). We kill σ's and τ's in stages. At stage $\alpha < 2^{\aleph_0}$, we kill σ_α and then τ_α, making sure that the winning sets of the two players are disjoint. We denote the winning set of Player I by A and the winning set of Player II by B. Initially, they are empty. They are updated in each stage by transfinite induction on the stages.

At stage α, we put in B all paths of the pruned tree T_{σ_α} corresponding to σ_α that have not already been put in A. Then we put in A all paths of the pruned tree T_{τ_α} corresponding to τ_α that have not already been put in B.

This completes the construction. It is easy to see that the disjointness condition of A and B was automatically satisfied when these sets were constructed. We prove conditions 1 and 2 by transfinite induction on stages. To verify condition 1, consider stage α. In the pruned tree T_{σ_α} for Player I's strategy σ_α, the paths already put in A are $T_{\sigma_\alpha} \cap A$. Since $A \subseteq \cup_{\beta<\alpha} T_{\tau_\beta}$, $T_{\sigma_\alpha} \cap A \subseteq \cup_{\beta<\alpha}(T_{\sigma_\alpha} \cap T_{\tau_\beta})$. By the continuum hypothesis, each $\alpha < 2^{\aleph_0}$ is either finite or countable. $T_{\sigma_\alpha} \cap T_{\tau_\beta}$ is a single path. Therefore, $T_{\sigma_\alpha} \cap A \subseteq \cup_{\beta<\alpha}(T_{\sigma_\alpha} \cap T_{\tau_\beta})$ contains only a countable number of paths.

This proves condition 1. Condition 2 can be proven similarly. □

We notice that the above proof still works with minor modifications even if the continuum hypothesis is replaced by a strictly weaker axiom, Martin's axiom. One of the consequences of Martin's axiom is that for any cardinal κ strictly between $\aleph_0$ and $2^{\aleph_0}$, the union of κ sets (as subsets of R) of Lebesgue measure 0 has Lebesgue measure 0. The topology that we use for the game tree is similar to the real line, and the probability measure on the paths induced by γ or δ is similar to the Lebesgue measure. This consequence thus applies to our case. In fact, we could even replace Martin's axiom by the following strictly weaker axiom: the union of κ many sets of measure 0 has measure 0 for any $\kappa < 2^{\alpha}$.

Although the randomized strategy δ for Player I (γ for Player II) wins against all deterministic strategies of Player II (Player I), it does not win against all randomized strategies of Player II (Player I). In particular, δ does not win against γ (and vice versa). Perhaps the power of randomization in this case results from its easy access to all deterministic strategies at once.

We have addressed the situation when one player uses a randomized strategy and the other uses a deterministic strategy. What happens when both use randomized strategies? Is it possible to obtain an equilibrium solution? That is, is there a pair of randomized strategies of the players such that none can gain by deviating from this randomized strategy? The problem has long been open when the choice space is continuous [Mc]. We conjecture that this is true for games with universally measurable winning sets. A set is universally measurable if it is measurable under any probability measure. Even if the conjecture is confirmed, we still need to know if there is an indeterminate game which is also universally measurable. For a complete understanding of the exact power that randomization provides to infinite games, we need to resolve these problems.

6. Remarks and open problems. While the result of [BBKTW, RaS] is a first step in understanding the relationship between randomized strategies and deterministic strategies in online problems, our study attempts to obtain a more refined understanding of this relationship in terms of computability. For the derandomization hypothesis to hold, we may have to drop in general the computability requirement for our strategies. The answer to our main question is not very satisfactory since it comes from an artificially constructed problem. It would be much more interesting if it could come from natural problems. In [Kn], there is a discussion of computable strategies in the prisoner's dilemma, a natural two-person game that has perplexed game-theorists for years.

Although online problems can be easily formulated as infinite games [BBKTW, RaS], there is no immediate transformation from the latter to the former. Even though we tried to construct a game to emulate the behavior of online problems, one may notice that the construction of the specific game for our main result needs the power of enumerating all computable strategies, which makes the game noncomputable. Thus there is still a gap to be filled between our result and the result of [BBKTW, RaS]. More legitimate candidates for infinite games as online problems are closed computable games. A closed game is computable if there is a Turing machine that can test for membership of basic open sets of the winning set of the adversary. We thus have an immediate question.

1. Does computable derandomization hypothesis hold for all computable closed games?

For a large class of problems, it has been shown in [BBKTW] that derandomiza-

tion may cost an extra multiplicative factor of $(1+\epsilon)$ on the competitive ratio. A related question is whether or not the result of [BBKTW, RaS] can be strengthened to apply to all semicomputably determinate games.

2. If there is an α-competitive randomized strategy against an offline adaptive adversary for a semicomputably determinate game, is there always an α-competitive *computable* deterministic strategy?

We know that the k-server game whose winning set is defined by the set of all those paths which achieve a ratio of less than c for any $c < k$ is semicomputably determinate from the lower-bound results of [MMS]. We also know from the results of Koutsoupias and Papadimitriou [KP] that when $c \geq 2k$, the k-server game is semicomputably determinate for every metric space. (Observe that the two-server game is semicomputably determinate for any c and any metric space since there is a computable algorithm [MMS] whose competitive ratio is 2).

3. Can we show that the k-server game is semicomputably determinate when c is in neither of these ranges?

Acknowledgments. We owe many thanks to the anonymous referees. Their critical readings of the manuscript were very important and allowed the authors to correct several errors in early drafts. We wish to thank Tiko Kameda for many discussions, his encouragement, and the support from his grant during the course of this research. Thanks are also due to Christos Papadimitriou, Prabhakar Raghavan, and Mike Saks for their many helpful suggestions and comments on the early versions of this paper and to Randall Dougherty and Gerald A. Edgar for suggesting some of the references.

REFERENCES

[BBKTW] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson, *On the power of randomization in online algorithms*, Algorithmica, 11 (1994), pp. 2–14.

[Bi] P. Billingsley, *Probability and Measure*, John Wiley, New York, 1986.

[BLS] A. Borodin, N. Linial, and M. Saks, *An optimal online algorithm for metrical task systems*, J. Assoc. Comput. Mach., 39 (1992), pp. 745–763.

[BRS] A. Blum, P. Raghavan, and B. Schieber, *Navigating in unfamiliar geometric terrain*, in Proc. 23rd Annual ACM Symposium on the Theory of Computing, ACM, New York, 1991, pp. 494–504; SIAM J. Comput., 26 (1997), pp. 110–137.

[CDRS] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir, *Random walks on weighted graphs, and application to on-line algorithms*, J. Assoc. Comput. Mach., 40 (1993), pp. 454–476.

[CKPV] M. Chrobak, H. J. Karloff, T. Payne, and S. Viswanathan, *New results on server problems*, SIAM J. Discrete Math., 4 (1991), pp. 172–181.

[CL] M. Chrobak and L. L. Larmore, *An optimal online algorithm for k servers on trees*, SIAM J. Comput., 20 (1991), pp. 144–148.

[CT] Y. S. Chow and H. Teicher, *Probability Theory*, Springer-Verlag, New York, 1978.

[DP1] X. Deng and C. H. Papadimitriou, *On the complexity of cooperative solution concepts*, Math. Oper. Res., 19 (1994), pp. 257–266.

[DP2] X. Deng and C. Papadimitriou, *Exploring an unknown graph*, in Proc. 31st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 355–361.

[FRR] A. Fiat, Y. Rabani, and Y. Ravid, *Competitive k-server algorithms*, J. Comput. System Sci., 48 (1994), pp. 410–428.

[Fu] C. Futia, *The complexity of economic decision rules*, J. Math. Econom., 4 (1977), pp. 289–299.

[Gro] E. F. Grove, *The harmonic online k-server algorithm is competitive*, in Proc. 21st Annual ACM Symposium on Theory of Computing, ACM, New York, 1991, pp. 260–266.

[GS] D. Gale and F. M. Stewart, *Infinite games with perfect information*, in Contributions to the Theory of Games, Vol. II, W. H. Kuhn and A. W. Tucker, eds., Ann. Math. Stud. 28, Princeton University Press, Princeton, New Jersey, 1953, pp. 245–266.

[HT] J. Y. Halpern and M. R. Tuttle, *Knowledge, probability, and adversaries*, J. Assoc. Comput. Mach., 40 (1993), pp. 917–960.

[KMRS] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, *Competitive Snoopy caching*, Algorithmica, 3 (1988), pp. 79–119.

[Kn] V. Knoblauch, *Computable strategies for repeated prisoner's dilemma*, Games Econom. Behav., 7 (1994), pp. 381–389.

[KP] E. Koutsoupias and C. H. Papadimitriou, *On the k-server conjecture*, in Proc. 26th Annual ACM Symposium on Theory of Computing, ACM, New York, 1994, pp. 507–511.

[KS] E. Kalai and W. Stanford, *Finite rationality and interpersonal complexity in repeated games*, Econometrica, 56 (1988), pp. 397–410.

[Ku] K. Kunen, *Set theory: An Introduction to Independence Proofs*, North–Holland, New York, 1980.

[Mar] D. A. Martin, *Borel determinacy*, Ann. Math., 102 (1975), pp. 363–371.

[Mc] J. McKinsey, *Introduction to the Theory of Games*, McGraw–Hill, New York, 1952.

[MMS] M. S. Manasse, L. A. McGeoch, and D. D. Sleator, *Competitive algorithms for on-line problems*, J. Algorithms, 11 (1990), pp. 208–230.

[Mos] Y. N. Moschovakis, *Descriptive Set Theory*, North–Holland, New York, 1980.

[Ne] A. Neyman, *Bounded complexity justifies cooperation in the finitely repeated prisoner's dilemma*, Econom. Lett., 19 (1985), pp. 227–229.

[Pa1] C. H. Papadimitriou, *On games played by automata with a bounded number of states*, J. Game Theory Econom. Behav., 4 (1992), pp. 122–131.

[Pa2] C. Papadimitriou, *Game against nature*, in Proc. 24th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1983, pp. 446–450.

[PY] C. Papadimitriou and M. Yannakakis, *Shortest paths without a map*, Theoret. Comput. Sci., 84 (1991), pp. 127–150.

[RaS] P. Raghavan and M. Snir, *Memory vs. randomization in online algorithms*, IBM J. Res. Develop., 38 (1994), pp. 683–707.

[RiS] R. L. Rivest and R. E. Schapire, *Inference of finite automata using homing sequences*, Inform. and Comput., 103 (1993), pp. 299–347.

[Si] H. Simon, *Theories of bounded rationality*, in Decision and Organization, R. Radner, ed., North–Holland, Amsterdam, 1972, pp. 161–176.

[ST] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. Assoc. Comput. Mach., 28 (1985), pp. 202–208.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 803–856, June 1997

013

TIGHTER UPPER BOUNDS ON THE EXACT COMPLEXITY OF STRING MATCHING*

RICHARD COLE† AND RAMESH HARIHARAN‡

Abstract. This paper considers how many character comparisons are needed to find all occurrences of a pattern of length m in a text of length n. The main contribution is to show an upper bound of the form of $n + O(n/m)$ character comparisons, following preprocessing. Specifically, we show an upper bound of $n + \frac{8}{3(m+1)}(n-m)$ character comparisons. This bound is achieved by an online algorithm which performs $O(n)$ work in total and requires $O(m)$ space and $O(m^2)$ time for preprocessing. The current best lower bound for online algorithms is $n + \frac{16}{7m+27}(n-m)$ character comparisons for $m = 16k + 19$, for any integer $k \geq 1$, and for general algorithms is $n + \frac{2}{m+3}(n-m)$ character comparisons, for $m = 2k + 1$, for any integer $k \geq 1$.

Key words. string matching, exact complexity, comparisons, periodicity

AMS subject classifications. Primary, 68R15; Secondary, 68Q25, 68U15

PII. S009753979324694X

1. Introduction.

String matching is the problem of finding all occurrences of a pattern $p[1 \ldots m]$ in a text $t[1 \ldots n]$. We assume that the characters in the text are drawn from a general (possibly infinite) alphabet unknown to the algorithm. We investigate the time complexity of string matching measuring both the exact number of comparisons and the time complexity counting all operations. As is standard, the time complexity refers to operations performed following preprocessing of the pattern; prepossessing of the text is not allowed. Our goal is to minimize the number of comparisons while still maintaining a total linear-time complexity and a polynomial-in-m preprocessing cost.

Note that if the algorithm is permitted to know the alphabet, then there is a finite automaton which performs string matching by reading each text character exactly once (which can be obtained from the failure function of [KMP77]). However, in this case the running time depends on the alphabet size.

Perhaps the most widely known linear-time algorithms for string matching are the Knuth–Morris–Pratt [KMP77] and Boyer–Moore [BM77] algorithms. We refer to these as the KMP and BM algorithms, respectively. The KMP algorithm makes at most $2n - m + 1$ comparisons and this bound is tight. The exact complexity of the BM algorithm was an open question until recently. It was shown in [KMP77] that the BM algorithm makes at most $6n$ comparisons if the pattern does not occur in the text. Guibas and Odlyzko [GO80] reduced this to $4n$ under the same assumption. Cole [Co91] finally proved an essentially tight bound of $3n - \Omega(n/m)$ comparisons for the BM algorithm, whether or not the pattern occurs in the text. Colussi [Col91] gave a simple variant of the KMP algorithm which makes at most $\frac{3}{2}n$ comparisons. Apostolico and Giancarlo [AG86] gave a variant of the BM algorithm which makes at

* Received by the editors April 12, 1993; accepted for publication (in revised form) July 19, 1995. This research was supported in part by NSF grants CCR-8902221 and CCR-8906949.
http://www.siam.org/journals/sicomp/26-3/24694.html

† Courant Institute of Mathematical Sciences, New York University, New York, NY 10012 (cole@cs.nyu.edu).

‡ Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India (ramesh@csa.iisc.ernet.in). This research was performed while this author was at the Courant Institute of Mathematical Sciences, New York University.

most $2n - m + 1$ comparisons. Crochemore et al. [CCG92] showed that remembering just the most recently matched portion reduces the upper bound of BM from $3n$ to $2n$ comparisons.

Recently, Galil and Giancarlo [GG92] gave a string-matching algorithm which makes at most $\frac{4}{3}n$ comparisons. This was the strongest upper bound for string matching known prior to our work. In fact, [GG92] gave this bound in a sharper form as a function of the period z of the pattern; the bound becomes $n + \min\{\frac{1}{3}, \frac{\min\{z, m-z\}+2}{2m}\}$ $(n - m)$.

Galil and Giancarlo [GG91] gave a lower bound of $n(1 + \frac{1}{2m})$ comparisons. For online algorithms, [GG91] showed an additional lower bound of $n(1+\frac{2}{m+3})$. An online algorithm is an algorithm which examines text characters only in a window of size m sliding monotonically to the right; further, the window can slide to the right only when all matching pattern instances to the left of the window or aligned with the window have been discovered. Recently, Zwick and Paterson gave additional lower bounds, including a bound of $\frac{4n}{3}$ for patterns of length 3 in the general case [ZP92].

Our contribution is a linear-time online algorithm for string matching which makes at most $n(1 + \frac{8}{3(m+1)})$ character comparisons. Our algorithm requires $O(m)$ space and $O(m^2)$ preprocessing time and runs in $O(m + n)$ time overall (exclusive of preprocessing). Independently, Breslauer and Galil discovered a similar algorithm which performs at most $n + O(\frac{n \log m}{m})$ comparisons [BG92]; this algorithm requires $O(m)$ preprocessing space and time and runs in linear time. Recently, Hancart [Ha93] and Breslauer et al. [BCT93] have independently shown an upper and lower bound of $(2 - \frac{1}{m})n$ on the number of comparisons required for string matching when comparisons must involve only text characters in a window of size one sliding monotonically to the right.

Nearly matching lower bounds are given in a companion paper [CHPZ92]. They show the following bounds: for online algorithms, a bound of $n + \frac{16}{7m+27}(n - m)$ character comparisons for $m = 16k + 19$, for any integer $k \geq 1$; and for general algorithms, a bound of $n + \frac{2}{m+3}(n - m)$ character comparisons, for $m = 2k + 1$, for any integer $k \geq 1$.

Even if exponential (in m) preprocessing and exponential space are available, it is not clear that the above upper bound can be achieved (assuming that a result independent of the alphabet size is sought). The difficulty is that text characters which are mismatched may need to be compared repeatedly. In order to minimize the total number of comparisons, this has to be offset by other text characters which do not need to be compared. The hardest patterns to handle are those which have proper suffixes which are also prefixes of the pattern. We refer to such substrings as *presufs*.[1] Our algorithm has two parts: a basic algorithm and a presuf handler. The basic algorithm handles primary patterns, i.e., patterns with no presufs; this is also the core of the algorithm for the general case. The presuf handler copes with presufs; its design constituted the main challenge in this work. Understanding the structure of the presufs was a key ingredient in its design. Understanding this structure also led to the new lower bound constructions given in [CHPZ92].

The flavor of the algorithm is as follows. Initially, the pattern is aligned with the left end of the text. Repeatedly, an attempt to match the pattern against the text is made. When a mismatch is found or the pattern is fully matched the pattern is shifted to the right. The goal is to maximize this shift without missing any possible

[1] Presufs are also called *borders* in the literature. Strings without presufs are called *primary* strings.

matches. The basic algorithm has the property that the length of each shift is at least equal to the number of comparisons since the previous shift (or the start of the algorithm). This results in an algorithm that performs at most n comparisons if the pattern has no presuf (the algorithms of [GG92] and [CP89] also have this property).

The presuf handler cannot quite match the performance of the basic algorithm (which is not surprising given that the lower bounds for this problem are larger than n comparisons). Here the approach is to follow the basic algorithm until a suffix which is also a prefix is matched. The only possible matches in which a new instance of the pattern overlaps the current partially (or fully) matched instance arise with an overlap by a presuf. Ignoring, for the moment, problems introduced by periodic patterns, it is the case that at most one of these overlapping pattern instances can result in a match. An elimination is performed to determine which one, if any, of the overlapping pattern instances might result in a match. Following this elimination, a further nontrivial sequence of comparisons is made; this can lead to one of two situations: another match of a suffix which is also a prefix, or a mismatch which causes a return to the basic algorithm. The presuf handler is invoked at most once for every $\frac{m}{2}$ text characters and performs a number of comparisons at most two greater than the number of characters shifted over. (Actually, there are two possible scenarios: an invocation after $\frac{3}{4}m$ text characters and at most two excess comparisons, or an invocation after $\frac{m}{2}$ text characters and at most one excess comparison.) Periodic patterns have the added difficulty that the presuf handler could be invoked more frequently. In this case, we show the additional fact that if the presuf handler is invoked after fewer than $\frac{m}{2}$ text characters, then the number of comparisons is at most the number of characters shifted over.

This structure of the algorithm of Breslauer and Galil is similar; their analogue of the presuf handler works in a completely different way, however.

Section 2 provides several definitions. The basic algorithm is described in section 3. In section 4, the presuf handler for nonperiodic strings is presented. Section 5 gives a technical construction deferred from section 4. Finally, in section 6, the result is extended to periodic patterns.

We remark here that the properties of strings which we develop in section 4 and later are mostly new and appropriate references are given otherwise.

2. Definitions and preliminaries. A string v is a *presuf* of p if it is both a proper suffix and a proper prefix of p. Let x be the length of the largest presuf of p. The *period* of a pattern p with length m is defined to be $m - x$. x is called the *s-period* (or shift period) of p. A string p is *cyclic* in string v if it is of the form v^k, $k > 1$. A *primitive* string is a string which is not cyclic in any string.

A string p is *periodic* if $p = wv^k$, where w is a (possibly null) proper suffix of v and $k > 1$. The smallest such v is called the *core* of p and the corresponding w is called the *head* of p. Note that the core is primitive. A *cyclic shift* of p is any string vu where $p = uv$. $|v$ and $v|$ refer, respectively, to the leftmost and rightmost characters in string v; on occasion, we will call these characters, respectively, the left end and right end of v. Two characters are said to be *distance* d apart if they are separated by $d - 1$ other characters.

For the rest of the paper, let p be a pattern with length m. Let the text t have length n. $p[i]$ denotes the ith character of p, reading from the left end; i is called the *index* of $p[i]$ in p. The same notation and terminology is used for string t.

The algorithm will be comparing the pattern with substrings of the text with which the pattern is aligned; as the algorithm proceeds, the pattern is shifted to the

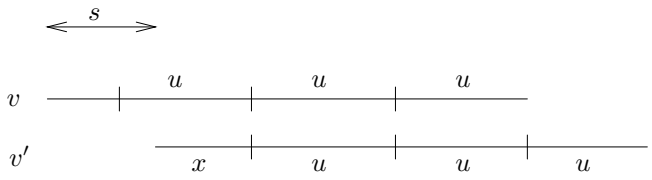


FIG. 1. *Periodicity.*

right across the text. Each possible alignment of the pattern with the text is called an *instance* of the pattern. Note that an instance is not necessarily an occurrence.

For each pair of overlapping instances of the pattern a location at which the two differ, if any, will be precomputed. This location is called the *difference point* of the two instances. Note, however, that for a given pair, a difference point may not exist, but this can happen only if the pattern has a nonempty presuf. Let p_1 and p_2 denote two pattern instances, where $p_1[i]$ is aligned with $p_2[1]$; then dif_i is the difference point if any; i.e., $p_1[dif_i] \neq p_2[dif_i - i + 1]$.

Let q be a pattern instance. Those pattern instances to the right of q, overlapping q, but which do not have a difference point with q are called the *presuf overlaps* of q.

We quote a few standard results concerning strings.

LEMMA 2.1. *Let w be a presuf of string v. If $|w| > \frac{|v|}{2}$, then v is periodic.*

Proof. See Fig. 1. Let $s = |v| - |w|$. Let v' denote string v shifted distance s to the right. Then the portion of v' overlapping v is presuf w which matches the corresponding portion of v. Let u denote the suffix of v' of length s. An easy induction shows that $v = xu^k$ for some $k \geq 2$, where x is a proper suffix of u. □

The following appear in different forms in [Lo82] (see Propositions 1.3.2, 1.3.4, and 1.3.5 there).

LEMMA 2.2 (see [LS62, FW65]). *If x and y are two distinct periods of a string v such that $x + y \leq m + \gcd\{x, y\}$, then $\gcd\{x, y\}$ is also a period of v.*

LEMMA 2.3. *Suppose that $v = xy$, where both x and y are presufs of v. Then v is cyclic in some string w of length $\gcd\{|x|, |y|\}$.*

LEMMA 2.4. *If v is periodic and can be expressed both as $x_1u_1^{k_1}$ and $x_2u_2^{k_2}$, where x_i is a suffix of u_i, $u_1 > u_2$, and $k_1, k_2 \geq 2$, then either u_1 is cyclic in u_2 or both u_1 and u_2 are cyclic in some smaller string.*

3. The basic algorithm. The algorithm in this section also appears in [Col91] and is also exposed in [GG92]. We describe it again for the sake of completeness.

If all the characters in p are identical, then it is easily seen that the KMP algorithm makes at most n character comparisons. Further, if $m = 2$ and p consists of two distinct characters, then the BM algorithm makes at most n character comparisons. Henceforth, we assume that $m > 2$ and that p has at least two distinct characters.

The algorithm proceeds by eliminating pattern instances as possible matches. It repeatedly performs the following two steps: first, it attempts to match the leftmost surviving pattern instance with the aligned text substring; then, it shifts to the next leftmost surviving pattern instance.

After a shift occurs, the strategy followed depends on the nature of the shift. The order in which pattern characters are compared ensures that all the shifts satisfy one of the following two properties.

1. A shift has size greater than or equal to the number of comparisons made since the previous shift. This is called a *basic shift.*
2. When property 1 is not true, a proper prefix x of p is completely matched with the text after the shift. Moreover, x is also a suffix of p. This is called a *presuf shift.*

Following a basic shift, the basic algorithm is continued; a presuf shift results in a transfer to the presuf handler.

The following observation is the key to the basic algorithm. Consider two overlapping instances of the pattern p. Then comparing either of the two pattern characters at their difference point with the aligned text character is sure to eliminate one of the two pattern instances from being a potential match. As long as the overlap is not a presuf of p, there will be a difference point. This is exactly the notion of *duelling* introduced by Vishkin [Vi85].

More formally, let p_a and p_b be the two leftmost surviving pattern instances, where p_b is not a presuf overlap of p_a. Let d be the difference point of p_a and p_b. $p_a[d]$ is compared with the aligned text character. A match eliminates p_b; a mismatch eliminates p_a.

Next, we give the exact sequence of comparisons made by the above strategy. We precompute the following sequence S. S is the sequence of indices $dif_2, dif_3, \ldots, dif_m$ omitting repetitions and undefined indices. Henceforth, where no ambiguity will result, we will use the sequence S to refer both to the indices it contains and to the corresponding characters in p_a.

The characters in p_a are compared with their corresponding text characters in two passes, stopping if a mismatch is found. In pass 1, those characters in p_a contained in S are compared in sequence. If all of these match, then the remaining pattern characters are compared from right to left in pass 2.

LEMMA 3.1. *If a mismatch occurs at the character given by the kth index in S, then the resulting shift has size at least k.*

Proof. Let the kth index in S be dif_l. Note that $k < l$. Recall that $l \leq dif_l \leq m$ and $p[dif_l] \neq p[dif_l - l + 1])$. Suppose for a contradiction that the shift was of length $j < k$. Let p_a and p_b be the pattern instances as specified in the algorithm above, before this shift. Note that p_b becomes p_a after the shift; i.e., p_b is p_a shifted j units. But then p_a and p_b have a difference point and hence dif_{j+1} is defined. dif_{j+1} is the ith index in S, for some $i \leq j$; hence since $j < k$, dif_{j+1} occurs prior to dif_l in S. Therefore, $p_a[dif_{j+1}]$ would have been matched against the text and one of p_a or p_b eliminated before $p_a[dif_l]$ was compared. The contradiction proves the lemma. □

Consequently, all shifts resulting from mismatches in pass 1 are basic shifts. When a basic shift is made, the basic algorithm is restarted. It is easy to see that if all shifts are basic shifts, then the total number of comparisons made is upper bounded by n.

Next, suppose that all comparisons in pass 1 result in matches.

LEMMA 3.2. *Suppose pass 2 results in a mismatch at $p_a[l]$. The resulting shift has length at least l.*

Proof. Suppose for a contradiction that the resulting shift has length i, $i < l$. Let p_b be p_a shifted distance i. Then l is a difference point for p_a and p_b; hence one of p_a and p_b would have been eliminated in pass 1, a contradiction. □

Consequently, for each shift resulting from pass 2 with length less than the number of comparisons made since the previous shift, a proper prefix of p (which is also a suffix of p) is matched with the text; i.e., it is a presuf shift. The main challenge in minimizing the exact number of comparisons is to handle presuf shifts.

Preprocessing. The sequence S, as defined above, is not unique. We show that a particular instance of S can be precomputed in a manner akin to the computation of the KMP shift function or the BM shift function. The KMP shift function comprises, for each j, $1 < j \leq m$, a number s_j. s_j is the largest i, $i < j$, such that $p[1 \ldots i-1] = p[j-i+1 \ldots j-1]$ and $p[i] \neq p[j]$; note that $i = dif_{j-i+1}$. If no such i exists, then s_j is defined to be zero. Consider the set of all those values of j for which $s_j > 0$. Furthermore, let this set be ordered by the increasing value of $j - s_j + 1$. This provides the sequence S. For every k, $2 \leq k \leq m$, if dif_k is defined, then for some $l \in S$, $k \leq l \leq m$, $p[1 \ldots l-k] = p[k \ldots l-1]$ and $p[l-k+1] \neq p[l]$; hence the value dif_k occurs in S (though not necessarily indexed by k). Finally, it is straightforward to compute S in $O(m)$ time.

4. The presuf handler. In this section, the presuf handler for nonperiodic patterns p is described. This presuf handler also deals with some presuf shifts for periodic p, as specified in the next few paragraphs.

With each presuf shift, we associate a presuf x'_1 of p, defined as follows. If p is not periodic, then x'_1 is the longest presuf of p. Otherwise, suppose $p = u_p v_p^{i_p}$ is periodic with core v_p and head u_p. Then if the presuf of p matching the text is at least $|v_p|$ long, $x'_1 = u_p v_p^{i_p - 1}$. Otherwise, if the above presuf is shorter than $|v_p|$, then x'_1 is defined to be the longest presuf of p of length less than $|v_p|$.

In this section, we give an algorithm for handling presuf shifts for the case $|x'_1| < \frac{m}{2}$. The case $|x'_1| \geq \frac{m}{2}$ is considered in section 6. Note that $|x'_1| < \frac{m}{2}$ always holds for nonperiodic p and may hold for periodic p.

Consider the situation immediately following a presuf shift. Some prefix of p, which is also a presuf, matches the text substring that it is aligned with. It is convenient for the presuf handler to assume that the pattern was shifted by $m - |x'_1|$ characters and that x'_1 matches the text. Note that this will not be the case if pass 2 in the basic algorithm mismatches before x'_1 is completely matched. A simple check will prevent the declaration of any incorrect complete match that might result from the above assumption. To facilitate this check, a variable t_{last} is used. Suppose pass 2 in the basic algorithm ends in a mismatch. Then t_{last} is set to the index of the text character where the mismatch occurred. Otherwise, if no mismatch occurs, $t_{\text{last}} \leftarrow \phi$.

Since $|x'_1| < \frac{m}{2}$, $p = x'_1 u x'_1$, for some string u. Let t_A be the substring of the text aligned with the prefix x'_1 of p immediately following the presuf shift; note that x'_1 matches t_A. Order all the presufs of x'_1 by decreasing length and let this order be $x_1, x_2, x_3, \ldots, x_k, x_{k+1}$, where x_k is the smallest nonnull presuf of x'_1 and x_{k+1} is the null string and hence a trivial presuf of x'_1. Note that $x_1 = x'_1$. Let the future instances of p (i.e., potential match instances) before its left end slides beyond $t_A|$, in left to right order, be $p_1, p_2, \ldots, p_k$. Let p_{k+1} be the pattern instance whose left end is to the immediate right of $t_A|$. Then p_i, $1 \leq i \leq k+1$, is the pattern instance with the prefix x_i of p aligned with the suffix x_i of t_A. x_i is said to be the presuf associated with p_i. $p_1, p_2, \ldots, p_k, p_{k+1}$ are called the *presuf pattern instances.*

LEMMA 4.1. *If $|x'_1| < \frac{m}{2}$, then at most one of $p_1, \ldots, p_k, p_{k+1}$ can lead to a complete match.*

Proof. This is a proof by contradiction. Suppose some two of them, say p_i and p_j, $i < j$, each result in a complete match. It follows that there is a prefix of p of size $m - |x_i| + |x_j|$ that matches a suffix of p. Since $|x_i| - |x_j| < \frac{m}{2}$, $m - |x_i| + |x_j| > \frac{m}{2}$; also, $|x_i| - |x_j| \leq |x'_1|$. This implies that p is periodic with core of length at most $|x'_1|$, contrary to our assumption. □

The presuf handler begins by eliminating all but at most one of $p_1, p_2, \ldots, p_k, p_{k+1}$.

This is carried out by a procedure that performs $j \leq k$ comparisons; at most two of these comparisons are unsuccessful. We seek to minimize the number of unsuccessful comparisons because while successful comparisons can be remembered, unsuccessful comparisons may lead to repeated comparison of some text characters.

The elimination procedure is described in section 4.1. The remainder of the presuf handler procedure for all but two special cases is given in section 4.2, and its analysis is presented in section 4.3. The special cases are handled in section 4.4. Finally, data structure details are described in section 4.5.

4.1. Elimination strategy. Before describing the exact sequence of comparisons made by the elimination strategy, we need to understand some structural properties of these overlapping instances of p.

LEMMA 4.2. *Suppose $x_i = uv^l$ is the ith presuf, where u is a proper suffix of v, v is primitive, and $l \geq 2$. Then $x_{i+1} = uv^{l-1}$.*

Proof. Certainly, uv^{l-1} is a presuf, so the only question is whether there is a presuf x between uv^l and uv^{l-1}. Suppose there is such an x. Since $|uv^{l-1}| < |x| < |uv^l|$ and since x is a prefix of uv^l, the suffix of x of length $|v|$ is a cyclic shift of v. But x is a suffix of uv^l, which implies that a proper cyclic shift of v matches v. By Lemma 2.3, v is cyclic, contrary to our assumption. □

LEMMA 4.3. *The presuf pattern instances can be partitioned into $g = O(\log m)$ groups*[2] *$A_1, A_2, \ldots, A_g$. The groups preserve the left-to-right ordering of the pattern instances; i.e., the pattern instances in group A_i are all to the left of those in group A_{i+1}, for $i = 1, \ldots, g-1$. Let B_i be the set of presufs associated with the pattern instances in A_i. Then either $B_i = \{u_i v_i^{k_i}, \ldots, u_i v_i^3, u_i v_i^2\}$ or $B_i = \{u_i v_i^{k_i}, \ldots, u_i v_i\}$ or $B_i = \{u_i v_i^{k_i}, \ldots, u_i v_i, u_i\}$, where $k_i \geq 1$ is maximal, u_i is a proper suffix of v_i, and v_i is primitive.*

Proof. The proof is by construction. The groups are constructed in left-to-right order. Inductively suppose A_i is being built presently and all presuf pattern instances with associated presufs longer than $u_i v_i^{k_i}$ have been placed in groups to the left of A_i.

$\{u_i v_i^{k_i}, \ldots, u_i v_i^2\}$ are all added to B_i. $u_i v_i$ is also added if and only if it is not periodic; otherwise, $u_i v_i$ starts set B_{i+1}. By Lemma 4.2, all presuf pattern instances with associated presufs longer than $u_i v_i$ are in group A_i or by induction in a group to its left. In addition, if u_i is empty and v_i has no presufs, then u_i is also added.

The maximality of k_i can be seen as follows. Suppose k_i is not maximal; i.e., there exists a presuf w of the form $u_i v_i^{k_i+1}$, $k_i+1 \geq 2$. By the inductive hypothesis describing the construction, this presuf would already be in one of the groups $B_1, \ldots, B_{i-1}$. By Lemma 4.2, it follows that w is the smallest presuf in B_{i-1}. w is clearly periodic. By construction, $w = u_{i-1} v_{i-1}^2 = u_i v_i^{k_i+1}$, $k_i + 1 \geq 2$. Then by Lemma 2.4, v_{i-1} must be cyclic, which contradicts the assumption that v_{i-1} is primitive. Thus k_i must be maximal.

This shows that the presuf pattern instances are partitioned into groups. It remains to show that there are only $O(\log m)$ groups. Let x_{j_i} be the leftmost presuf in B_i. If $x_{j_{i+1}} = u_i v_i$, then $|x_{j_{i+1}}| \leq \frac{2}{3}|x_{j_i}|$, and otherwise $|x_{j_{i+1}}| \leq \frac{1}{2}|x_{j_i}|$. (The latter claim follows because $x_{j_{i+1}}$ is both a prefix and a suffix of x_{j_i} and this prefix and suffix are nonoverlapping.) The $O(\log m)$ bound follows immediately. □

LEMMA 4.4. *The groups satisfy the following properties.*

Property 1. *Consider the presufs x_i corresponding to the pattern instances p_i in some group A_j. For $j \neq g$, all of these presufs x_i, except possibly the rightmost one,*

[2] Actually, a sharper bound of $\log_\phi m$ groups is known [KMP77, B94], where ϕ is the golden ratio.

are periodic with the same core and head. For $j = g$, all but the rightmost two presufs are periodic with the same core and head.

Property 2. *Let p_i be the rightmost instance in its group. If x_i is periodic then so is x_{i+1}.*

Property 3. *Suppose p_i is the rightmost instance in its group A_j and x_i is periodic with head u and core v; then $|x_{i+2}| < |v|$. Further, suppose $x_{i+1} = u'(v')^l$, where v' is primitive and u' is a proper suffix of v'. Then $|v'| > |u|$.*

Property 4. *Suppose p_i is the rightmost instance in its group A_j, where $|A_j| > 1$; further, suppose that x_{i-1} is periodic with core v and x_i is not periodic. Then $|x_{i+1}| < |v|$.*

Property 5. *Both p_k and p_{k+1} are in the group A_g.*

Proof. Let p_i be in group A_j.

Property 1 is true by definition. To see Property 2, note that since x_i is periodic, $x_i = uv^l$, where u is a proper suffix of primitive v and $l \geq 2$. But if $l > 2$, then the pattern instance corresponding to either presuf uv^2 or presuf uv would be the rightmost item in A_j. Thus $l = 2$; however, by definition, the pattern instance corresponding to uv is not in A_j only if uv is periodic. Finally, by Lemma 4.2, $x_{i+1} = uv$.

Property 3 can be seen as follows. As in the previous paragraph, $x_i = uv^2$ and $x_{i+1} = uv$. Again uv is periodic; that is, $uv = u'(v')^l$ for some $l \geq 2$, where u' is a proper suffix of v' and v' is primitive. By Lemma 4.2, $x_{i+2} = u'(v')^{l-1}$. Suppose $|v'| \leq |u|$. Since v is primitive, there must be a substring v' of $u'(v')^l = uv$ which straddles the boundary between u and v. Thus the substring of $u'(v')^l$ aligned with the rightmost $|v'|$-sized substring of u is a proper cyclic shift of v'. But since u is a suffix of v this substring is also identical to v'. By Lemma 2.3, v' is cyclic, a contradiction. Thus $|v'| > |u|$ and hence $|x_{i+2}| < |v|$.

Property 4 can be seen as follows. As with the previous properties, it follows that $x_i = uv$, where u is a proper suffix of v and v is primitive. If $|x_{i+1}| \geq |v|$, then $|x_{i+1}| > |x_i|/2$. But x_{i+1} is a presuf of x_i; by Lemma 2.1, x_i would be periodic, a contradiction.

Property 5 can be seen as follows. Since x_k is the smallest nonnull presuf of p, no nonnull prefix of x_k matches a suffix of x_k. Therefore, all strings in B_g have the form $u_g v_g^l$, $0 \leq l \leq k_g$, where u_g is the null string and $v_g = x_k$. Since both x_k and x_{k+1} have this form, Property 5 is true. □

Remark. The elimination strategy described below and the algorithm in section 4.2, which uses this elimination strategy to handle presuf shifts, work for most patterns p. However, there are some patterns for which presuf shifts must be handled differently. The reason for this is made clear in section 5, which gives a technical portion of the analysis of the algorithm in section 4.2. These exception patterns are precisely those in which x_k, the smallest nonnull presuf, is a single character and g, the number of groups, is one. Presuf shifts for these exception patterns are handled separately in section 4.4.

DEFINITION. *A clone set is a set $Q = \{s_1, s_2, \ldots\}$ of strings, with $s_i = uv^{k_i}$, where u is a proper suffix of primitive v and $k_i \geq 0$. A set U of pattern instances is* half-done *if $|U| \leq 2$ or the set of associated presufs forms a clone set.*

The following lemma is the key to our elimination strategy.

LEMMA 4.5. *Consider three presuf pattern instances p_a, p_b, p_c, $a < b < c$. (The order of the indices corresponds to the left-to-right order of the pattern instances.) Suppose the set $\{x_a, x_b, x_c\}$ is not a clone set. Then there exists an index d in p_1*

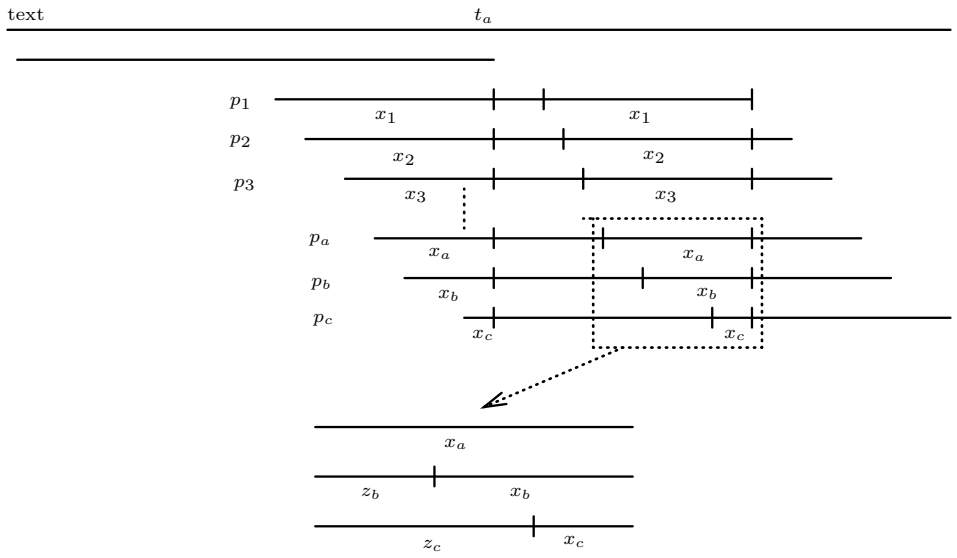


FIG. 2. *Overlapping pattern instances.*

with the following properties. The characters in $p_1, p_2, \ldots, p_a$ *aligned with* $p_1[d]$ *are all equal*; *however, the character aligned with* $p_1[d]$ *in at least one of* p_b *and* p_c *differs from* $p_1[d]$. *Moreover,* $m - |x_a| + 1 \leq d \leq m$; *i.e.,* $p_1[d]$ *lies in the suffix* x_a *of* p_1.

Proof. The substrings of $p_1, \ldots, p_a$ aligned with the suffix x_a of p_1 are all identical to the string x_a. Let the substring of p_b (respectively, p_c) aligned with the suffix x_a of p_1 be y_b (respectively, y_c). See Fig. 2. It suffices to show that at least one of y_b or y_c is not identical to x_a. Suppose for a contradiction that $y_b = y_c = x_a$.

Let $y_b = z_b x_b$ and $y_c = z_c x_c$. Note that z_b is a suffix of z_c. Since $y_b = y_c$, a simple induction shows that $y_b = uv^l$, where u is a proper prefix of v and $l \geq 1$, $|v|$ is either $|x_b| - |x_c|$ or some proper divisor of $|x_b| - |x_c|$, and v is primitive. Since $x_a = y_b$, if $l \geq 2$, x_a is periodic with core v.

First, suppose x_a is periodic with head u and core v. By Lemma 4.2, if $|x_b| > |uv|$, $x_b = uv^h$ for some h, $1 \leq h < l$. If $|x_b| < |uv|$, since $|x_b| - |x_c|$ is a multiple of $|v|$, $|x_b| = |v| + |x_c|$, so $x_b = wv$ for some string w, $|w| < |u|$. But then wv is a prefix of uv, which implies that v is cyclic; this is a contradiction. Thus $x_b = uv^h$. Since $|x_b| - |x_c|$ is a multiple of $|v|$, $x_c = uv^j$ for some j, $0 \leq j < h$, contradicting the fact that $\{x_a, x_b, x_c\}$ is not a clone set.

Consequently, $x_a = uv$. If x_a is not periodic, then $|x_b| < \frac{|x_a|}{2}$ and $|v| \leq |x_b| - |x_c| < \frac{|x_a|}{2}$. But then $|x_a| < 2|v| < |x_a|$, a contradiction. If x_a is periodic, $x_a = u'(v')^k$ for some $k \geq 2$. Also, $|v'| > |u|$ by Property 3 of Lemma 4.4. Hence $|x_b| < |v|$. But $|x_c| \leq |x_b| - |v| < 0$, a contradiction. □

Lemma 4.5 implies that a comparison of $p_1[d]$ with the aligned text character has the following effect: if it is a mismatch, all of $p_1, \ldots, p_a$ are eliminated, while if it is a match, at least one of p_b and p_c is eliminated.

Lemma 4.5 enables the elimination of essentially all but one group of pattern instances with at most one mismatch. At each step, for the rightmost d yielded by Lemma 4.5, $p_1[d]$ is compared with the aligned text character. If there is a mismatch, the surviving set of pattern instances is half-done, as we show in the following lemma. If there is no such d, the surviving set of pattern instances is half-done by Lemma

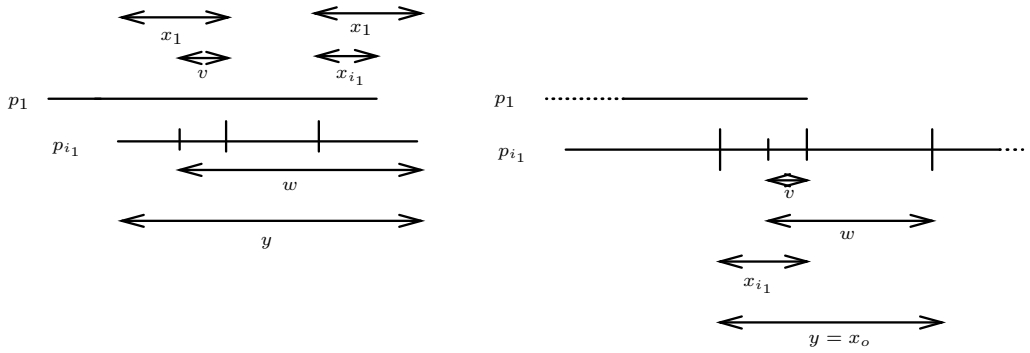


FIG. 3. (a) $p_{i_1} \in A_1$. (b) $p_{i_1} \notin A_1$.

4.5. This procedure comprises Phase 1 of the elimination procedure.

LEMMA 4.6. *If there is a mismatch in Phase* 1 *of the elimination procedure, the set* X *of surviving pattern instances is half-done.*

Proof. Suppose it was not; i.e., for some subset $\{p_a, p_b, p_c\}$ of X, $\{x_a, x_b, x_c\}$ is not a clone set. The characters in the x_i suffix of p_1, for $i = a, b, c$, match the aligned substring of p_i. Hence the mismatch at $p_1[d]$, which created set X, lies to the left of the suffix x_i of p_1, for $i = a, b, c$. However, by Lemma 4.5, at least one of p_a, p_b, and p_c could have been eliminated by a comparison made within the suffix of p_1 of size $\max\{|x_a|, |x_b|, |x_c|\}$. This contradicts the choice of d as the rightmost index at which a comparison eliminates some pattern instance. □

The elimination among the remaining half-done set of pattern instances also requires at most one mismatch.

LEMMA 4.7. *Let* $O = \{p_{i_1}, p_{i_2}, \ldots, p_{i_l}\}$, $l \geq 2$, *be an uneliminated half-done set and let* $p_{i_1} \in A_r$. *Then* $x_{i_j} = uv^{h-j}$, *where* u *is a proper suffix of primitive* v *and* $h \geq l$. *Further, there exists an index* d *such that the characters in* $\{p_{i_1}, p_{i_2}, \ldots, p_{i_{l-1}}\}$ *aligned with* $p_{i_l}[d]$ *are all equal, but differ from the character* $p_{i_l}[d]$. $p_{i_l}[d]$ *is aligned with or to the left of* $p_{i_1}[m]$. *In addition, if* $p_{i_1} \notin A_1$ *then* $p_{i_l}[d]$ *is to the right of* $p_1[m]$ *and within distance* $|x_o| - |x_{i_1}|$ *of* $p_1[m]$, *where* p_o *is the rightmost pattern instance in* A_{r-1}. *If* $p_{i_1} \in A_1$, *then* $p_{i_l}[d]$ *is to the right of* t_A *and aligned with or to the left of* $p_1[m]$.

Proof. Each x_{i_j}, $1 \leq j \leq l$, is of the form uv^{h_j}, for some $h_j \geq 0$, where u is a proper suffix of primitive string v.

See Fig. 3. Let y denote the string p_{i_1} if $p_{i_1} \in A_1$ and the string x_o otherwise. Clearly, y cannot be periodic with core v. If $p_{i_1} \in A_1$, then let w denote the suffix of p_{i_1} of length $m - |x_1| + |v|$. If $p_{i_1} \notin A_1$, then let w denote the substring of p_{i_1} which has length $|x_o| - |x_{i_1}| + |v|$ and which overlaps p_1 in exactly $|v|$ characters. Note that $w \neq uv^{h'}$, where $h' > 0$; otherwise, by Lemma 2.3 and the fact that v is primitive, the suffix of y of length $|w| - |v|$ is cyclic in v and therefore y is periodic with core v, contrary to our assumption.

Let w' be the smallest suffix of w which is not of the form $u'v^{h'}$, with u' a suffix of v and $h' > 0$. Define d to be the index in p_{i_1} corresponding to $|w'|$. Clearly, $|w'| > |x_{i_1}| \geq (l-1)|v|$. Therefore, $p_{i_1}[d + (l-1)|v|]$ is a character in w. In addition, if $p_{i_1} \in A_1$, then $|w'| > |x_1|$ and therefore $p_{i_1}[d + (l-1)|v|]$ is aligned with or to the left of $p_1[m]$. The lemma follows if $p_{i_l}[d]$ is aligned with $p_{i_1}[d + (l-1)|v|]$ and the characters in $p_{i_1}, \ldots, p_{i_{l-1}}$ which are aligned with $p_{i_l}[d]$ are all identical and different

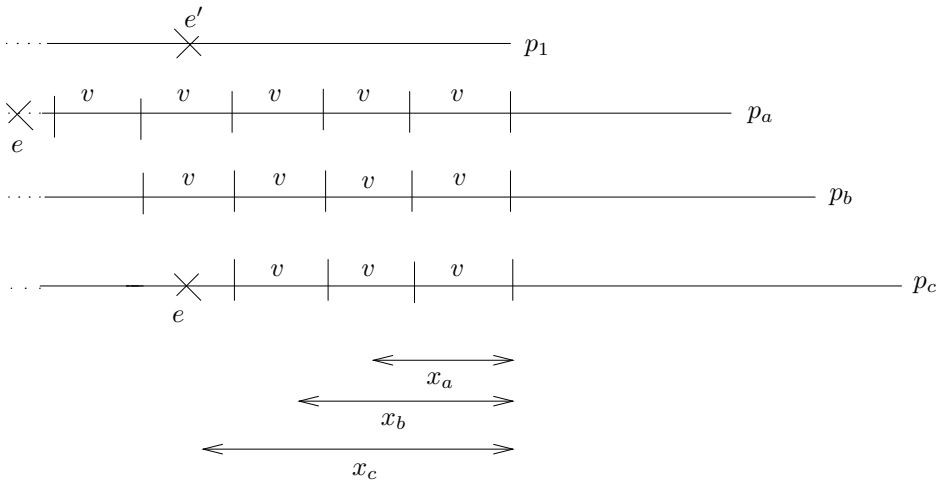


FIG. 4. *The half-done set is complete.*

from $p_{i_l}[d]$. We show that these two claims are indeed true.

First, we show that $|x_{i_j}| - |x_{i_{j+1}}| = |v|$, for $1 \le j < l$. Suppose for a contradiction that there is a pattern instance $p_b \notin O$ with $x_b = uv^{h''}$, $h'' > 0$, and there are pattern instances $p_a, p_c \in O$, p_a to the left of p_b and p_c to the right of p_b. See Fig. 4. Let $p_a[e]$ be the rightmost character in p_a such that the substring of p_a which starts at $p_a[e]$ and overlaps p_1 is longer than $|x_a|$ and not periodic with core v. Consider the character $p_1[e']$ aligned with $p_c[e]$. The portions of p_a, p_b, and p_c which overlap the suffix of p_1 to the right of e' are all identical. If Phase 1 had stayed to the right of e', then p_a, p_b, and p_c would all have been eliminated by the mismatch at the end of Phase 1. Thus $p_1[e']$ must have been compared in Phase 1. A mismatch at e' eliminates p_a and p_b while a match eliminates p_c. Either way, a contradiction results.

Finally, note that the character in p_{i_j}, $1 \le j \le l-1$, which is aligned with $p_{i_l}[d]$ is precisely the character $p_{i_l}[d + (l-j)|v|]$. But $p_{i_l}[d] \ne p_{i_l}[d + |v|] = p_{i_l}[d + 2|v|] = \cdots = p_{i_l}[d + (l-1)|v|]$. □

COROLLARY 4.8. *To eliminate all but one of the pattern instances in any half-done set (in particular, the Phase* 1 *survivors set)* $\{p_{i_1}, p_{i_2}, \ldots, p_{i_j}\}$*, it suffices to compare a sequence of characters with the property that any two consecutive characters in the sequence are distance* $|v|$ *apart, where* v *is the core of* x_{i_1}*. Further, the pattern instances in this set are eliminated in right-to-left order by this comparison sequence (i.e., in decreasing value of* j*).*

Let p_{i_l} be as in Lemma 4.7; the character in p_{i_1} aligned with $p_{i_l}[d]$ is compared with the aligned text character. A match eliminates p_{i_l}; a mismatch leaves only p_{i_l} surviving. Iteration of this step ends with one pattern instance surviving after at most one mismatch. This comprises Phase 2 of the elimination procedure.

The sequence of comparisons made in Phase 2 is clearly a right-to-left sequence. If p_1 is eliminated in Phase 1, then all comparisons in Phase 2 are made to the right of the characters compared in Phase 1. Otherwise, if p_1 is not eliminated in Phase 1, all comparisons in Phase 2 are made to the left of the characters compared in Phase 1.

We recapitulate the elimination strategy now. In Phase 1, characters in p_1 at in-

dices given by a precomputed sequence S_1 are compared in sequence until a mismatch occurs or until the sequence is exhausted. Associated with a mismatch at the ith comparison given by S_1 is an auxiliary sequence S_{2_i} of indices. If a mismatch occurs at the ith comparison in S_1, Phase 2 begins and comparisons are now made according to the auxiliary sequence S_{2_i}. A mismatch at any index in the relevant auxiliary sequence completes the elimination process as does the exhaustion of that auxiliary sequence. In either case, only one pattern instance from the set $\{p_1, \ldots, p_k, p_{k+1}\}$ survives.

Let $|S_1| = j$. The sequences S_1 and $S_{2_1}, S_{2_2}, \ldots, S_{2_j}$ collectively form a tree ET (the *elimination tree*). ET is a binary tree. Each internal node x of ET stores an index indicating the comparison to be made. Each internal node has two children. The computation continues at the left child if the comparison at x is successful and at the right child otherwise. The computation starts at the root of ET. Each external node stores the one pattern instance to survive the two phases of comparisons leading to that external node. The external nodes are also called terminal nodes. Note that no pattern instance p_i can be the survivor at two distinct terminal nodes of ET. This is because one of the two outcomes of the comparison at the least common ancestor of these two nodes in ET is bound to eliminate p_i. It follows that the size of ET is $O(k)$.

The total number of mismatches occurring in the elimination process is at most two because each phase terminates when a mismatch occurs.

LEMMA 4.9. *All but at most one of $p_1, \ldots, p_k, p_{k+1}$ can be eliminated by making up to k comparisons using the $O(k)$-sized binary comparison tree ET. At most two of these comparisons result in mismatches. The sequence of comparisons made by the elimination strategy consists of two left-to-right sequences. The second sequence is either entirely to the right or entirely to the left of the first one.*

4.2. Strategy for handling presuf shifts. Subsequent to the elimination due to tree ET, the presuf handler proceeds in a manner reminiscent of the basic algorithm. That is, there is a current pattern instance, p_a, which is being matched and which is the leftmost surviving pattern instance. The next leftmost surviving pattern instance, p_b, which has a difference point with p_a is a candidate for elimination. Indeed, a comparison of p_a with the text is made at the difference point.

The analysis of the presuf handler has the following flavor. With a few exceptions, comparisons are charged to distinct text characters. To be precise, for each suffix shift, at most two comparisons are charged to the shift rather than to text characters. Even more precisely, if two comparisons are charged to the shift, the next presuf shift is at distance at least $\frac{3(m+1)}{4}$ to the right, and otherwise it is at distance at least $\frac{m+1}{2}$ to the right. The complexity bound of the algorithm now follows readily.

THEOREM 4.10. *The algorithm performs at most $n + \frac{8}{3(m+1)}(n-m)$ character comparisons.*

There are three ways in which text characters are charged:

(i) The character compared is charged.

(ii) The text character aligned with the left end of the pattern instance eliminated by the comparison is charged.

(iii) The text character to the immediate right of $t_A|$ (i.e., aligned with $|p_{k+1}$) is charged.

The three charging methods do not interact readily. To ensure that no text character is charged twice, the switching from one charging method to the other will occur only at carefully selected points in the algorithm. In addition, mismatches

are not charged according to rule (i) since the text characters in question may be compared again.

Basically, charging method (i) is used if a pattern instance is successfully matched (at least up to a suffix which is a presuf). Charging method (iii) is used only for the comparison that eliminates the presuf pattern instance which survives the elimination procedure ET. Charging method (ii) is used otherwise. A partial exception arises for the characters compared by procedure ET; this is discussed further below.

Following the use of procedure ET, the aim is to perform comparisons essentially as in the basic algorithm, that is, to compare the character at the difference point of the two leftmost surviving pattern instances which are not prefix overlaps of each other. The analysis ceases to be as straightforward because of the additional $j \leq k$ comparisons performed by procedure ET; indeed, to cope with this, a modified form of the basic algorithm is needed.

There are two objectives:

1. to avoid repeating comparisons at the text characters successfully compared by procedure ET;

2. to perform essentially j fewer comparisons than in the basic algorithm.

Objective 1 is achieved by keeping a record of the successful comparisons in a bit vector of length roughly m.

The major difficulty, however, is caused by the method used for charging the j comparisons made by procedure ET. It is natural to charge these comparisons to the text characters compared. Unfortunately, this may conflict with charging using method (ii). To avoid this difficulty, a single additional comparison, with text character t_b, is performed before using procedure ET. The following lemma can then be shown.

LEMMA 4.11. *For each text character t_c compared by procedure ET, with at most $\alpha \leq 2$ exceptions, there is a distinct previously uncharged text character $t_{c'}$, with $t_{c'}$ aligned with or to the left of t_c and to the right of $|p_{k+1}$, such that the pattern instance q_c whose left end is aligned with $t_{c'}$ mismatches either the text character t_b or some text character matched in procedure ET.*

Let β be the number of mismatches performed by procedure ET. Then, in addition, $\alpha + \beta \leq 2$.

The lemma is proven by specifying a transfer function f, which associates c with c'. The form of f depends on the sequence of comparisons performed by procedure ET. The proof of the lemma is quite nontrivial; it is deferred until section 5.

Lemma 4.11 is used as follows. Let q_e be the next pattern instance to match the text (or at least to have a suffix, which is also a presuf, matching the text). All pattern instances to the left of q_e, eliminated by comparisons made after the use of procedure ET, are charged used charging method (ii). Using the transfer function, those comparisons to the left of $|q_e$ made by procedure ET are charged to text characters which are not otherwise charged. By contrast, text characters aligned with q_e are charged using charging method (i). There will be no more than two comparisons performed by the presuf handler that are not thereby charged to a text character; these comparisons are charged to the presuf handler itself.

The algorithm requires a total of five subphases, whose details depend on exactly how q_e arises.

It is helpful to distinguish three scenarios that may ensue. To this end, let p_e denote the presuf pattern instance to survive the elimination using tree ET. In addition, let t_a denote the text character $t_A|$.

The three scenarios follow:

1. All pattern instances overlapping p_e are eliminated apart from its presuf overlaps, and p_e or at least a suffix of p_e is matched.

2. p_e is eliminated. In addition, there is some pattern instance q_c overlapping p_e such that all pattern instances overlapping q_c are eliminated apart from its presuf overlaps; further, q_c or at least a suffix of q_c is matched.

3. p_e is eliminated, as are all pattern instances overlapping p_e. Let q_d denote the leftmost surviving pattern instance in this case.

The first scenario causes no problems from the perspective of the analysis. It suffices to ensure that none of the successful comparisons made by ET are repeated. The third scenario is handled by using charging scheme (iii) for the comparison which eliminates p_e and charging scheme (ii) for the remaining comparisons in the post-ET phase. Lemma 4.11 ensures that for each comparison made by ET (with at most two exceptions) with a text character strictly between t_a and $|q_d$, there is a distinct pattern instance whose left end lies strictly between $|p_{k+1}$ and $|q_d$ and which is eliminated by the comparisons made by ET plus the one other comparison at text character t_b. Again, this leads to the desired complexity bound without difficulty.

The second scenario provides the greatest difficulty. In order to avoid unnecessary comparisons, the locations of successful comparisons are recorded. Then if a difference point occurs at one of these matched text characters, the present pattern instance p_b (see the first paragraph of the subsection) can be removed without further comparisons. However, following a mismatch, it is not clear how to maintain this property: with only linear storage, it is not clear how to ensure that the current pattern instance following the mismatch agrees with the text on a previously matched character, at least if the total work bound is to be linear. (There is no problem if exponential-in-m space is available for precomputed structures.) To avoid this difficulty, only the successful comparisons since the last mismatch are recorded.

In fact, this is not quite good enough. It appears necessary to keep track of the characters compared by procedure ET regardless of how many characters are compared. This avoids subsequent comparison of these characters. Indeed, any pattern instances mismatching on one or more of these characters are eliminated immediately after the computation with procedure ET. This is done with the help of precomputed information.

With this motivation, we proceed with a precise description of the presuf handler procedure. It proceeds in five steps.

Step 1 (before the use of tree ET). The characters in $p_1, \ldots, p_k$ aligned with $p_1[m]$, the rightmost character in p_1, are identical. If the character in p_{k+1} aligned with $p_1[m]$ is also identical to it, then $p_1[m]$ is compared with the aligned text character. A mismatch eliminates all of $p_1, \ldots, p_k, p_{k+1}$ and the basic algorithm is restarted with $|p_a$ placed immediately to the right of $|p_{k+1}$. A match is not immediately beneficial since it does not eliminate any of $p_1, \ldots, p_k, p_{k+1}$. However, it ensures the elimination of sufficiently many appropriate pattern instances for scenarios 2 and 3 described above.

Step 2. The elimination strategy using tree ET is applied to the pattern instances $p_1, \ldots, p_k, p_{k+1}$.

Following Step 2, at most one presuf pattern instance survives. Call it p_e. Let Q denote the set of pattern instances which overlap p_e and have their left end to the right of $|p_{k+1}$. In the elimination process, some elements of Q may have also been eliminated from being potential matches. They need not be reconsidered. Indeed,

since the characters successfully matched in Step 2 must not be compared anew, it appears that these pattern instances must not be considered anew. To this end, a subset Q_x of Q is associated with each terminal node x in ET.

Let T_x denote the indices of the text characters successfully compared in Steps 1 and 2. Q_x contains those pattern instances in Q which match at all the text indices in T_x, except possibly the last. This seemingly odd exception is necessary in order to store Q_x efficiently. Actually, Q_x satisfies further constraints, but they are not needed for this section. The complete definition of Q_x and the method for computing it are described in section 4.5. Here it suffices to work with the following property: all but at most two of the comparisons in Steps 1 and 2 are successful and are remembered by pattern instances in Q_x.

Suppose that the elimination process terminates at terminal node x. Let $Q' = \{p_e\} \cup Q_x$. The elimination procedure of Step 3 is applied to the pattern instances in Q'.

Step 3. This step eliminates among the elements of Q'. q_c will denote the leftmost pattern instance to survive Step 3. If $q_c = p_e$, then every surviving pattern instance overlapping q_c will be a presuf overlap of q_c.

The strategy used here is similar to the one for the basic algorithm. One of two overlapping pattern instances is eliminated by comparing at the difference point of the two instances.

To prevent repeated comparisons of text characters to the right of t_a, two additional data structures are used. The first is a bit vector $BV[1 \ldots 2m]$. $BV[i] = 1$ if the ith text character to the right of t_a has been successfully compared in Steps 1 and 2 or in Step 3 since the last mismatch. The second is a list LBV; it stores the indices of the bits in BV set to one in Step 3 since the last mismatch. Initially, LBV is empty.

The elimination procedure for Step 3 follows. Let q_a and q_b denote the two leftmost uneliminated pattern instances in Q'. Suppose that q_b lies i units to the right of q_a. The reader is advised to refer to section 2 to review the definition of dif_{i+1}. If dif_{i+1} is undefined, then q_b is removed from Q'.

If dif_{i+1} is defined, then the bit in BV corresponding to the text character aligned with $q_a[dif_{i+1}]$ is read. If this bit is 1, then q_b is removed from Q'. (q_b can be eliminated since it does not match an already compared text character.) Otherwise, $q_a[dif_{i+1}]$ is compared with the aligned text character. If the two characters are equal, the corresponding bit in BV is set, the bit's index is added to LBV, and q_b is eliminated. If they are not equal, then the bits in BV at all indices currently in LBV are reset to 0, LBV is reset to empty, and q_a is eliminated.

The elimination procedure is iterated until only one pattern instance remains in Q'. Let q_c denote this remaining pattern instance.

Step 4. In this step, either all pattern instances overlapping q_c, apart from presuf overlaps, are eliminated or q_c is eliminated.

Let Q'' be the set of pattern instances whose left end lies to the right of $p_e|$ but not to the right of $q_c|$. The following step is repeated until either q_c is eliminated or $Q'' = \phi$. Let q_d be the leftmost pattern instance in Q''. Suppose q_d lies i units to the right of q_c. If dif_{i+1} does not exist, then q_d is removed from Q''. Otherwise, the following bit in BV is read: the bit corresponding to the text character aligned with $q_c[dif_{i+1}]$. If this bit is 1, q_d is eliminated. If it is 0, $q_c[i]$ and the aligned text character are compared. If they match, then the corresponding bit in BV is set, its index is added to LBV, and q_d is eliminated. Otherwise, q_c is eliminated and Step 4

comes to an end.

If q_c is eliminated, then LBV is reset to be empty, BV is reset to 0, and the basic algorithm is restarted with $p_a = q_d$. Otherwise, Step 5 is performed.

Step 5. This step seeks to complete the match of q_c. If at least a presuf of q_c is matched, the complete match or the partial match results in a new presuf shift. Otherwise, the basic algorithm is resumed with $|p_a$ immediately to the right of $q_c|$.

Step 5 compares the characters in q_c to the right of t_a, apart from those matched in Steps 1 and 2, and those matched in Steps 3 and 4 following the most recent mismatch. (Incidentally, there was no mismatch in Step 4 since q_c survived Step 4 if Step 5 is performed.) These characters are identified with the help of bit vector BV. They are matched in right-to-left order until either a mismatch occurs or they are all matched.

If they all match q_c is declared a complete match if either $t_{\text{last}} = \phi$ or t_{last} lies to the left of $|q_c$. Recall that t_{last} is the index of the text character mismatched, if any, immediately prior to the most recent presuf shift (if there was no mismatch, $t_{\text{last}} = \phi$).

Next, BV is reset to zero, LBV is reset to be empty, and t_{last} is updated as follows. If the above right-to-left pass results in a mismatch, then t_{last} is set to the index of the text character at which the mismatch occurs. Otherwise, t_{last} retains its value unless $|q_c$ is to its right. In the latter case, $t_{\text{last}} := \phi$.

The present situation is identical to that preceding a presuf shift in the basic algorithm. This resulting shift is treated in the same way; it too is called a presuf shift.

4.3. The analysis. The comparison complexity of the algorithm of section 4.2 is given by the following lemma.

LEMMA 4.12. *If p is not a special case pattern and $|x'_1| < \frac{m}{2}$ for each presuf shift, then the comparison complexity of the algorithm is bounded by $n(1 + \frac{8}{3(m+1)})$.*

Proof. We give a charging scheme to account for the comparisons made by the algorithm. This scheme charges almost every comparison to a distinct text character. The only exceptions are a few of the comparisons made by the presuf shift handler. For each presuf shift, depending on the distance between this presuf shift and the next one, the charging scheme fails to charge for up to two of the comparisons made by the presuf shift handler. We refer to the number of comparisons which the charging scheme fails to charge to distinct text characters as the *overhead* of the presuf shift. If a presuf shift has an overhead of two, we show that the next presuf shift must occur at least distance $\frac{3(m+1)}{4}$ to the right of the current presuf shift. The comparison complexity of our algorithm now follows from the fact that any two consecutive presuf shifts must occur at least distance $\frac{m+1}{2}$ apart.

Charging scheme. The charging scheme charges in phases. The phases begin and end at shifts and at reversions to the basic algorithm. There are four types of phases; for each phase type, a different charging scheme is used:

1. a phase beginning and ending with a basic shift;
2. a phase beginning in the basic algorithm and ending with a presuf shift;
3. a phase beginning with a presuf shift and ending with a reversion to the basic algorithm;
4. a phase beginning and ending with a presuf shift.

Consider any phase and let q_1 and q_2 refer to the leftmost surviving pattern instances at the beginning and end of the phase, respectively. Note that for Type 3 and Type 4 phases, q_1 is a presuf overlap of the pattern instance q', the leftmost uneliminated pattern instance prior to the presuf shift which initiated this phase.

Specifically, the prefix x'_1 of q_1 is aligned with the suffix x'_1 of q'. (Recall from the start of section 4 that on a presuf shift, we assume that the suffix x'_1 of q_1 matches the text.)

The charging scheme obeys the following properties.

1. At the start of a Type 1 or Type 2 phase, only text characters to the left of q_1 have been charged.

2. At the start of a Type 3 or Type 4 phase, only text characters aligned with or to the left of the prefix x'_1 of q_1 have been charged.

Type 1 *phase.* Suppose i comparisons were made in this phase. These i comparisons are charged to text characters which are aligned with q_1 but to the left of $|q_2$. By Lemma 3.1, $|q_2$ lies at least i characters to the right of $|q_1$. Thus each text character aligned with q_1 and to the left of $|q_2$ is charged at most once in this process. Clearly, property 1 holds at the start of the next phase.

Type 2 *phase.* In each comparison, a distinct character in q_1 is compared with the aligned text character. Each of these comparisons is charged to the text character compared. Thus each text character aligned with q_1 is charged at most once in this process. Clearly, property 2 holds at the start of the next phase.

The charging scheme for Type 3 and Type 4 phases is more involved. Before describing the scheme, we mention the ranges of the text characters charged in each case.

Type 3 *phase.* The text characters charged lie to the right of the right end of the prefix x'_1 of q_1 and to the left of $|q_2$. Each text character in this range is charged at most once. Clearly, property 1 holds at the start of the next phase.

Type 4 *phase.* The text characters charged lie to the right of the right end of the prefix x'_1 of q_1 and are aligned with or to the left of the rightmost character in the prefix x'_1 of q_2. Each text character in this range is charged at most once. Clearly, property 2 holds at the start of the next phase.

Clearly, the ranges of the text characters charged for different phases are disjoint. Next, we specify the charging scheme for Type 3 and Type 4 phases and justify the claims regarding the overhead.

Consider a presuf shift which initiates a new Type 3 or Type 4 phase. Let q' be the leftmost uneliminated pattern instance immediately before the presuf shift. Recall that t_a is the text character aligned with $q'|$. Consider the comparisons made by the current use of the presuf shift handler. If a mismatch occurs in Step 1, the current phase ends immediately and the basic algorithm is resumed. The presuf shift in this case has overhead 1 and the next presuf shift occurs at least distance $m+1$ to the right. Next, suppose that the comparison in Step 1 is successful. Let p_e be the presuf pattern instance that survives the elimination using tree ET in Step 2. After the presuf shift handler finishes, one of the three scenarios mentioned in section 4.2 ensues. We consider each in turn.

1. All pattern instances overlapping p_e are eliminated, apart from its presuf overlaps, and p_e or at least a suffix of p_e is matched. This is a Type 4 phase.

All comparisons made by the presuf shift handler, except the unsuccessful comparisons in Step 2, are charged to the text characters compared. The bit vector BV ensures that each of these comparisons involves a different text character. Thus each text character which lies to the right of t_a and is aligned with or to the left of $p_e|$ is charged at most once. At most two comparisons in Step 2 are unsuccessful, so this shift has overhead at most 2.

Consider the situation when there are exactly two mismatches in Step 2. p_1 is clearly eliminated in this case. In addition, we show in the next paragraph that if x_1 is periodic, with core v and head u, say, then all pattern instances whose associated presufs have the form uv^o, $o \geq 1$, are also eliminated. Let x_e be the presuf associated with p_e. It follows that $x_1 = x_e w x_e$ for some nonempty string w. Since $p = x_1 z x_1$, for some nonempty string z, $|x_e| \leq \frac{m-3}{4}$. This guarantees that the next presuf shift occurs at least distance $\frac{3(m+1)}{4}$ to the right. If there is just one mismatch in Step 2, then since $|x_1'| < \frac{m}{2}$, the next presuf shift occurs at least distance $\frac{m+1}{2}$ to the right.

To see that two mismatches in Step 2 eliminate all presuf pattern instances with associated presufs of the form uv^o, $o \geq 1$, it suffices to show that at most one such pattern instance survives the first mismatch; the second mismatch will surely eliminate this pattern instance. Suppose two pattern instances p_{i_1} and p_{i_2}, $i_1 < i_2$, $i_1, i_2 \neq 1$, $x_{i_1} = uv^{o_1}$, $x_{i_2} = uv^{o_2}$, $o_1, o_2 \geq 1$, survive the first mismatch, which occurs at text character t_x, say. The portions of p_1 and p_{i_1} to the right of t_x match each other while the characters in p_1 and p_{i_1} aligned with t_x are different. This implies that p_1 and p_{i_2} have a difference point strictly between t_x and t_b; more precisely, the character in p_1 which is distance $(o_2 - o_1)|v|$ to the right of t_x is a difference point. Therefore, either p_1 or p_{i_2} would have been eliminated before the first mismatch, which is a contradiction.

2. p_e is eliminated. In addition, there is some pattern instance q_c overlapping p_e such that all pattern instances overlapping q_c are eliminated apart from its presuf overlaps; further, q_c or at least a suffix of q_c is matched. This is also a Type 4 phase.

Each comparison in Steps 1 and 2 with a text character to the left of $|q_c$ for which function f is defined is charged to the text character specified by the function f, called its f value; f values are distinct by definition. Comparisons in Step 3 fall into one of three categories (see Lemma 4.11 and the following paragraph):

1. comparisons which eliminate pattern instances whose left ends lie to the right of $|p_{k+1}$ and to the left of $|q_c$;
2. comparisons which eliminate pattern instances whose left ends are aligned with or to the right of $|q_c$;
3. the comparison which eliminates p_e.

Each comparison in the first category is charged to the text character aligned with the left end of the pattern instance eliminated. By the definition of the function f, these text characters do not occur in the range of f values. Comparisons in the second category, along with the comparisons made in Steps 4 and 5 and those successful comparisons in Steps 1 and 2 that involve text characters overlapping q_c, are charged to the text characters compared. BV ensures that each of these comparisons involves a distinct text character. Thus each text character which lies to the right of $|p_{k+1}$ and is aligned with or to the left of $q_c|$ is charged at most once. The comparison that eliminates p_e is charged to the text character aligned with $|p_{k+1}$. Since all f values lie to the right of $|p_{k+1}$ and all pattern instances eliminated by comparisons in the first category are with left ends to the right of $|p_{k+1}$, this text character is charged exactly once. The two comparisons in Step 2 lacking f values constitute the overhead of this presuf shift. Since p_e is eliminated, the next presuf shift occurs at least distance $m+1$ to the right of the current presuf shift.

3. p_e is eliminated, as are all pattern instances overlapping p_e. This is a Type 3 phase.

Let q_d denote the leftmost surviving pattern instance. All comparisons in Steps 1 and 2 for which function f is defined are charged to their f values. f values are

distinct by definition. Excluding the comparison which eliminates p_e, each comparison in Steps 3 and 4 eliminates some pattern instance whose left end lies to the right of $|p_{k+1}$ and to the left of $|q_d$. Each such comparison is charged to the text character aligned with the left end of the pattern instance eliminated. These text characters cannot occur in the range of the function f and hence are charged only once. Thus each text character which lies to the right of $|p_{k+1}$ and to the left of $|q_d$ is charged at most once. The comparison that eliminates p_e is charged to the text character aligned with $|p_{k+1}$. The two comparisons in Step 2 lacking f values constitute the overhead of this presuf shift. Since p_e is eliminated, the next presuf shift occurs at least distance $m+1$ to the right of the current presuf shift. □

The following lemma is shown in section 4.5.

LEMMA 4.13. *The total space used by the algorithm for the case when $|x'_1| < \frac{m}{2}$ for all presuf shifts is $O(m)$. Further, for any terminal node x of ET, Q_x can be obtained in $O(m)$ time. The preprocessing required by the algorithm can be accomplished in $O(m^2)$ time.*

LEMMA 4.14. *Suppose that p is not a special-case pattern and $|x'_1| < \frac{m}{2}$ for all presuf shifts. Then the total time taken by the algorithm is $O(n+m)$, following preprocessing of the pattern, which takes $O(m^2)$ time.*

Proof. By Lemma 4.12, the number of character comparisons made is $O(n)$. It remains to count the time spent in all other operations. The basic algorithm makes only character comparisons. Next, consider the presuf handler of section 4.2. Steps 1 and 2 make only character comparisons. Following Step 2, computing Q_x takes $O(m)$ time by Lemma 4.13. Steps 3 and 4 take $O(m)$ time because $|Q'|, |Q''| = O(m)$ and each of the operations in these steps, except the operations used for resetting BV, leads to the removal of a pattern instance from one of Q'' or Q'. Further, the total time spent by Steps 3 and 4 in resetting BV is bounded by the time taken by these steps to set bits in BV, which is $O(m)$. Clearly, Step 5 takes $O(m)$ time. Thus the total time taken by the presuf handler of section 4.2 is $O(m)$. Since any two presuf shifts occur at least $m-|x'_1| > \frac{m}{2}$ distance apart, the total time taken by the algorithm is $O(n+m)$. □

4.4. Handling presuf shifts for special-case patterns. As mentioned in section 4.1, a different algorithm is needed to handle presuf shifts for patterns for which $|x_k| = 1$ and $g = 1$. We give an algorithm which handles presuf shifts for such patterns when $|x'_1| < \frac{m}{2}$. (Recall that x'_1 for a presuf shift was defined towards the start of section 4.) The case where $|x'_1| \geq \frac{m}{2}$ is handled in section 6.

The goal of this algorithm is to reach one of the following two situations:

1. the identification of a pattern instance q_c satisfying the following property: no pattern instance q_d which precedes q_c survives and a pattern instance overlapping q_c survives only if it is a presuf overlap of q_c;

2. a return to the basic algorithm.

Further, this is achieved with at most two mismatches.

Let $x_k = b$. Any character other than b is called a non-b character. Since we assume that the pattern contains at least two different characters, it contains a non-b character. Let $p[j]$ be the leftmost non-b character in p and let t_c denote the text character aligned with $|p_{k+1}$. Let t_d be the leftmost non-b text character, if any, to the right of, and including, t_c.

By the definition of special-case patterns, all presufs consist solely of b's. Therefore, $p_1[j]$ lies to the right of t_a. Note that no complete match can occur with one of $p[1 \ldots j-1]$ aligned with t_d. Thus if t_d lies to the left of $p_1[j]$, then the next potential

match instance of p would have its left end to the right of t_d. Otherwise, the next potential match instance of p has $p[j]$ aligned with t_d. Also notice that if t_d does not exist, then there are no more complete matches. These observations lead to the following three-step procedure.

Step 1. This step locates t_d and then eliminates all but at most one pattern instance q_c overlapping t_d. Starting at t_c, a left-to-right scan of the text is performed to locate t_d (i.e., each text character is compared to b; t_d is the character at which the first mismatch occurs). If t_d does not exist, the algorithm halts. If t_d exists and lies to the left of $p_1[j]$, then the basic algorithm is restarted with $|p$ placed to the immediate right of t_d. Otherwise, q_c is chosen to be the pattern instance such that $q_c[j]$ is aligned with t_d. q_c is the next potential match instance to be considered.

Step 2. In this step, either q_c is eliminated or all pattern instances overlapping q_c, except for presuf overlaps of q_c, are eliminated. This is done using the basic algorithm, slightly modified to account for the matched prefix. Suppose the leftmost difference points are used in the sequence S in the basic algorithm, as against any arbitrary difference points. Then $dif_2, \ldots, dif_j$ are all equal to j and $dif_{j+1}, \ldots, dif_m$ are all greater than j, whenever defined. In Step 2, the characters in q_c to the right of $q_c[j]$ which are at the indices given by S are compared with the aligned text characters in the order in which they appear in S. This continues until either a mismatch occurs or the sequence is exhausted. A mismatch leads to the basic algorithm with $|p$ shifted to the right of q_c by distance at least $j-1$ plus the number of comparisons made in this step. If no mismatch occurs, then Step 3 follows.

Step 3. Characters in q_c which are not yet matched are compared from right to left with their aligned text characters until a mismatch occurs or q_c is fully matched. The present situation is now identical to the situation at the beginning of a presuf shift and is handled in the same way.

The comparison complexity of the above algorithm is determined by the following lemma.

LEMMA 4.15. *If p is a special-case pattern and $|x'_1| < \frac{m}{2}$ for each presuf shift, then the comparison complexity of the algorithm is $n(1 + \frac{2}{m+1})$.*

Proof. We give a charging scheme to account for the comparisons made by the algorithm for handling special-case patterns. The definition of a phase, the charging scheme for Type 1 and Type 2 phases, and the ranges of text characters charged in each phase type remain the same as in Lemma 4.12. Only the charging scheme for Type 3 and Type 4 phases needs to be modified in accordance with the presuf shift handler for special-case patterns.

Consider a presuf shift which initiates a new Type 3 or Type 4 phase. We show that it has an overhead of at most one. The comparison complexity of the algorithm now follows from the fact that $|x'_1| < \frac{m}{2}$ and therefore any two consecutive presuf shifts must occur at least $\frac{m+1}{2}$ characters apart.

Charging scheme for the presuf shift handler. Let q' and q_1 be the leftmost uneliminated pattern instances immediately before and after the presuf shift, respectively. Recall that t_a is the text character aligned with $q'|$.

We show that presuf shifts have overhead at most one for these patterns. Let q_c be the leftmost pattern instance which survives Step 1. All successful comparisons in Step 1 are charged to the text characters compared. These text characters lie to the left of $q_c[j]$, where j is the least index such that $p[j]$ differs from $p[m]$. The lone unsuccessful comparison in Step 1 constitutes the overhead of this shift. Now consider two cases.

1. Suppose q_c survives Step 2. All comparisons made in Steps 2 and 3 are charged to the text characters compared. Thus each text character which lies to the right of t_a and is aligned with or to the left of $q_c|$ is charged at most once. All future comparisons will be charged to text characters to the right of $q_c|$.

2. Suppose q_c does not survive Step 2. Each successful comparison in Step 2 eliminates some pattern instance lying entirely to the right of $q_c[j]$ and is charged to the text character aligned with the left end of that pattern instance. The unsuccessful comparison which eliminates q_c in Step 2 is charged to the text character aligned with $q_c[j]$. Thus each text character lying strictly between t_a and $|q_d$ is charged at most once, where q_d is the leftmost surviving pattern instance at the end of Step 2. All future comparisons will be charged to text characters aligned with or to the right of $|q_d$. □

LEMMA 4.16. *Suppose that p is a special-case pattern and $|x'_1| < \frac{m}{2}$ for all presuf shifts. Then the total time taken by the algorithm is $O(n+m)$, following preprocessing of the pattern, which takes $O(m^2)$ time. The total space used by the algorithm is $O(m)$.*

Proof. The lemma, except for the preprocessing time, is obvious from the above description. Since no extra preprocessing is required for special-case patterns, the lemma follows from Lemma 4.14. □

THEOREM 4.17. *Suppose for all presuf shifts that $|x'_1| < \frac{m}{2}$. Then the total space used by the algorithm is $O(m)$ and the total time taken by the algorithm, after preprocessing, is $O(n+m)$. The preprocessing required by the algorithm takes $O(m^2)$ time.*

Proof. The proof follows from Lemmas 4.14 and 4.16. □

4.5. Data structure details. We prove Lemma 4.13 in this section. The following data structures are used by the algorithm:

1. the array S used in the basic algorithm;
2. an array, indexed by i, storing dif_i, $2 \le i \le m$, used by the presuf shift handler;
3. BV and LBV, the bit vector and its associated list;
4. ET, the elimination tree;
5. Q_x, for each terminal node x of ET, as defined after Step 2 in section 4.2.

Of these, the first three have size $O(m)$ by definition. By Theorem 4.9, ET also has size $O(m)$.

It remains to show how to represent Q_x, for each terminal node x of ET, using $O(m)$ space overall. The following definitions are helpful. Let t_b be the text character aligned with $p_1|$. Let Q refer to the set of pattern instances which overlap p_{k+1}, have left ends to the right of $|p_{k+1}$ and either match or do not overlap t_b.

Before showing how to maintain Q_x, it is helpful to recapitulate some structural properties of ET. ET is a binary tree with each internal node having two children. At each internal node y, a character c_y in p is potentially compared with the text character tc_y. A successful comparison leads to the left child of y while a mismatch leads to the right child. Comparisons are made starting at the root of ET and continuing until a terminal node (a leaf) is reached. A node in ET lies in the right subtree of at most two of its ancestors.

Node x is said to be a *failing descendant* of node y if x is a proper descendant of y and lies in the right subtree of y. A terminal node x can be a failing descendant of at most two nodes in ET. Let $p(x)$ denote the parent of x. For each terminal node x, let $Anc(x)$ be defined as follows. If both children of $p(x)$ are terminal nodes and

$p(x)$ is the right child of $p(p(x))$, then $Anc(x)$ is the set of proper ancestors of $p(x)$. Otherwise, $Anc(x)$ is the set of proper ancestors of x.

$q \in Q$ is said to *occur* at terminal node x of ET if $q \in Q_x$. In section 4.2, we tentatively defined Q_x to be the set of pattern instances in Q which match at all text characters compared successfully at nodes in $Anc(x)$ (actually, the definition was not this precise). Now we refine this definition by letting Q_x satisfy some additional constraints. Informally, q should occur at x if it is consistent with all comparisons made at nodes in $Anc(x)$. This motivates the following characterization of Q_x. Let $Y \subset Anc(x)$ consist of those nodes with respect to which x is a failing descendant. Then Q_x is the maximal subset of Q such that each $q \in Q_x$ satisfies the following properties:

1. $\forall y \in Anc(x) - Y$, the character in q aligned with tc_y, if any, matches the character c_y;
2. $\forall y \in Y$, the character in q aligned with tc_y, if any, is different from c_y.

ET may have $\theta(m)$ terminal nodes. Even though $|Q_x| < m$ for each terminal node x, storing Q_x explicitly for each terminal node x could require $\Omega(m^2)$ space overall. We show how to store the sets Q_x so that $O(m)$ space is used in total and any particular Q_x can be retrieved in $O(m)$ time.

Let $l_1, l_2, \ldots, l_h$, in that order, be the nodes along the leftmost path in ET starting at the root and ending at the terminal node l_h. Define the right subtree of l_i to be the subtree rooted at the right child of l_i. Note that $tc_{l_1}, \ldots, tc_{l_{h-1}}$ form a right-to-left sequence. We show how to maintain Q_x, for all terminal nodes x in the right subtrees of $l_1, \ldots, l_{h-1}$, in $O(m)$ space altogether. Only the terminal node l_h remains and Q_{l_h} can be stored explicitly in $O(m)$ space.

We mark some of the nodes $l_1, \ldots, l_{h-1}$. Node l_i is marked if its right child is neither a terminal node nor the parent of two terminal nodes. Thus node l_i is marked if Phase 2 could make at least two comparisons following a mismatch at tc_{l_i}. Let $l'_1, \ldots, l'_s$, in that order, be the nodes marked.

The following lemmas are helpful.

LEMMA 4.18. *Consider terminal nodes x_1 and x_2 of ET and let their least common ancestor be y. Suppose at most one of the following is true: first, y is the parent of both x_1 and x_2, and second, y is the right child of $p(y)$. If q occurs at x_1 and at x_2, then q does not overlap tc_y.*

Proof. Clearly, $y \in Anc(x_1)$ and $y \in Anc(x_2)$. Suppose q overlaps tc_y. Let c be the character in q aligned with tc_y. Without loss of generality, assume that x_1 is a failing descendant of y. Then x_2 is not a failing descendant of y. By the definition of Q_{x_1}, $c \neq c_y$. By the definition of Q_{x_2}, $c = c_y$, a contradiction. □

COROLLARY 4.19. *Let $i \geq 1$ be the smallest number such that $q \in Q$ does not overlap tc_{l_i}. q can occur at terminal nodes in the right subtrees of at most one of $l_1, \ldots, l_{i-1}$. Further, if q occurs at some terminal node in the subtree rooted at l_i, it cannot occur at terminal nodes in the right subtrees of any of $l_1, \ldots, l_{i-1}$.*

LEMMA 4.20. *Let $i \geq 1$ be the smallest number such that $q \in Q$ does not overlap tc_{l_i}. Suppose q occurs at a terminal node in the subtree rooted at l_i. Then q occurs at all terminal nodes in the right subtrees of each of those nodes among $l_i, \ldots, l_{h-1}$ which are unmarked. Further, q occurs at l_h.*

Proof. Clearly, the characters in q which overlap $tc_{l_1}, \ldots, tc_{l_{i-1}}$ match the characters $c_{l_1}, \ldots, c_{l_{i-1}}$, respectively. Further, q does not overlap $tc_{l_i}, \ldots, tc_{l_{h-1}}$. Therefore, q occurs at l_h. In addition, if a terminal node x is in the right subtree of an unmarked node l_j, $j \geq i$, then either $l_j = p(x)$ or $l_j = p(p(x))$ and $p(x) \notin Anc(x)$. From the

definition of Q_x, q must occur at x. □

LEMMA 4.21. *Consider marked node l'_i, $1 \le i \le s$, and let j be the smallest number such that $tc_{l'_i}$ is to the left of the suffix x_j of p_1. p_{j-1} must be the rightmost pattern instance in its group. In addition, p_j is the leftmost presuf pattern instance to survive a mismatch at $tc_{l'_i}$.*

Proof. Since l'_i is marked, at least three presuf pattern instances must survive a mismatch at $tc_{l'_i}$. Let the leftmost three such pattern instances be p_a, p_b, and p_c (listed in left-to-right order). Let A_w be the group containing p_j. Write x_j as uv^e, where $e \ge 1$, u is a proper suffix of primitive v, and all presufs associated with A_w have the form $uv^{e'}$, $e' \ge 1$.

By Lemma 4.5, successful comparisons within the suffix x_j of p_1 suffice to eliminate all but at most two of the pattern instances in the groups $A_{w+1}, \ldots, A_g$. (At most two pattern instances in $A_{w+1}, \ldots, A_g$ can form a half-done set with p_j.) Therefore, $p_a \in A_w$ and $x_a = uv^{e_a}$, $e_a \ge 1$. Since p_a, p_b, and p_c all survive the mismatch at $tc_{l'_i}$, $\{p_a, p_b, p_c\}$ is a half-done set and therefore $e_a \ge 2$. It follows that $x_b = uv^{e_b}$, $e_b \ge 0$, and $x_c = uv^{e_c}$, $e_c \ge 0$.

Next, suppose p_{j-1} is not the rightmost pattern instance in its group. Then $p_{j-1} \in A_w$ and x_{j-1} has the form uv^{e+1}, $e+1 \ge 2$. We show that p_b would have been eliminated by a comparison to the right of $tc_{l'_i}$, which is a contradiction. Note that p_{j-1} and p_a have a difference point, which is aligned with or to the right of $tc_{l'_i}$ and aligned with p_1. Let $p_{j-1}[d]$ be the rightmost such difference point. Clearly, $p_{j-1}[d]$ is to the left of the suffix x_a of p_1. $p_{j-1}[d + (e_a - e_b)|v|]$ is a difference point of p_{j-1} and p_b which is aligned with p_1. A match at this difference point would have eliminated p_b.

Finally, suppose $p_a \ne p_j$. Then p_j and p_a have a difference point, which is aligned with or to the right of $tc_{l'_i}$ and aligned with p_1. Let $p_j[d]$ be the rightmost such difference point. An argument similar to the one in the previous paragraph shows that p_j and p_b have a difference point to the right of $p_j[d]$ and aligned with p_1; a match at this difference point would have eliminated p_b, which is a contradiction. □

COROLLARY 4.22. *Consider marked nodes l'_{i_1} and l'_{i_2}, $1 \le i_1 < i_2 \le s$. Let x_{i-1} and x_{j-1} be the smallest suffixes (which are also presufs) of p_1 which overlap $tc_{l'_{i_1}}$ and $tc_{l'_{i_2}}$, respectively. Then $i \ne j$.*

Proof. If $i = j$, then by Lemma 4.21, p_j is the leftmost presuf pattern instance to survive the mismatches at both $tc_{l'_{i_1}}$ and $tc_{l'_{i_2}}$. But since a p_j survives a mismatch at $tc_{l'_{i_1}}$, it cannot survive a match at $tc_{l'_{i_1}}$ and therefore it cannot survive a mismatch at $tc_{l'_{i_2}}$. □

LEMMA 4.23. *The size of the presuf corresponding to the rightmost pattern instance in A_j, $1 \le j \le g$, is at most $\frac{m}{(3/2)^j}$.*

Proof. For $j = 1$, the claim is clearly true. Assume that the claim is true for A_{j-1}; i.e.; the size of the presuf corresponding to rightmost pattern instance p_e in A_{j-1} is less than $\frac{m}{(3/2)^{j-1}}$. x_e has either the form uvv or the form uv, where u is a proper suffix of v. In the former case, $x_{e+1} = uv$, and in the latter case, $x_e = x_{e+1} z x_{e+1}$ for some nonempty string z (since uv is not periodic). Thus $|x_{e+1}| < \frac{2|x_e|}{3}$. The claim follows. □

LEMMA 4.24. *The number of pattern instances in Q which overlap t_b and are entirely to the right of $tc_{l'_i}$ is less than $\frac{m}{(3/2)^{s-i+1}}$, for all i, $1 \le i \le s$.*

Proof. From Lemma 4.21 and Corollary 4.22, the rightmost presuf pattern instance p_j such that the suffix x_j of p_1 overlaps $tc_{l'_i}$ must be the rightmost pattern instance in some group $A_{j'}$, $j' \geq s - i + 1$. The lemma follows from Lemma 4.23. □

Consider the right subtrees of $l'_1, \ldots, l'_s$. Note that the comparisons made in each of these subtrees are aimed at eliminating half-done sets whose leftmost pattern instances are in distinct groups. Each of these comparisons is made to the right of $p_1[m]$, as described in Lemma 4.7 and Corollary 4.8.

LEMMA 4.25. *The number of pattern instances in Q which are entirely to the right of t_b and overlap some text character compared in the right subtree of l'_i is at most $\frac{m}{(3/2)^{s-i+1}}$, for all i, $1 \leq i \leq s$.*

Proof. Recall from Lemma 4.7 that a half-done set whose leftmost pattern instance is in group A_j, $j > 1$, is eliminated in Phase 2 of the elimination strategy by making comparisons at text characters which are at most distance $|x_{j'-1}| - |x_{j'}|$ to the right of t_b, where $p_{j'}$ is the leftmost presuf pattern instance in A_j. From Lemma 4.23, $|x_{j'-1}| < \frac{m}{(3/2)^{j-1}}$, and the lemma follows. □

LEMMA 4.26. *Consider a marked node l'_i and the set of terminal nodes in its right subtree. If a pattern instance q occurs at two of these terminal nodes, say w and y, then q occurs at all terminal nodes in the subtree rooted at the least common ancestor z of w and y.*

Proof. By Lemma 4.18, q does not overlap the character tc_z. Since comparisons made in the right subtree of l'_i constitute a right-to-left sequence, q does not overlap $tc_{z'}$, where z' is any descendant of z. The lemma now follows immediately from the definition of the sets Q_x. □

We are now ready to describe the data structure for storing the Q_x's. The following subsets of Q are required: $Z_1, \ldots, Z_{h-1}$, $Y_1, \ldots, Y_{h-1}$ and $W_1, \ldots, W_s$. The Z and the Y sets are used for terminal nodes which lie in the right subtrees of the unmarked nodes among $l_1, \ldots, l_{h-1}$. The W sets are used for terminal nodes which lie in the right subtrees of marked nodes.

The Z sets are defined first. For each i, $1 \leq i \leq h-1$, where l_i is unmarked, define Z_i to be the set of pattern instances in Q which overlap $tc(l_i)$ and occur only at terminal nodes in the right subtree of l_i. Clearly, $\sum_{i=1}^{h-1} Z_i = O(m)$.

The Y sets are defined next. For each i, $1 \leq i \leq h-1$, define Y_i to be the set of pattern instances $q \in Q$ with the following properties.

1. q does not overlap tc_{l_i}.
2. If $i > 1$, q overlaps $tc_{l_{i-1}}$.
3. q occurs at a terminal node in the subtree rooted at l_i.

Clearly, $\sum_{i=1}^{h-1} Y_i = O(m)$. Further, each pair of Y sets is disjoint and Z_i is disjoint from $Y_1, \ldots, Y_i$. The following lemma explains the significance of the Y and Z sets.

LEMMA 4.27. *If terminal node x is in the right subtree of unmarked node l_i, $Q_x = Y_1 \cup Y_2 \cup \cdots \cup Y_i \cup Z_i$.*

Proof. Suppose $q \in Q_x$. If q overlaps tc_{l_i} then by Corollary 4.19, $q \in Z_i$. If q does not overlap tc_{l_i}, then clearly q must be in some Y_j, $j \leq i$.

Next, suppose $q \in Y_j$, $j \leq i$. By Lemma 4.20, $q \in Q_x$. Finally, if $q \in Z_i$, then $q \in Q_x$ since the only internal node (if any) in the right subtree of l_i is not in $Anc(x)$. □

Finally, the W sets are defined. For each i, $1 \leq i \leq s$, W_i consists of those pattern instances which occur at some terminal node in the right subtree of marked node l'_i. Let W'_i denote the set obtained from W_i by removing those pattern instances which

do not overlap any of the text characters compared in the right subtree of l'_i.

LEMMA 4.28. $\sum_{i=1}^{s} |W'_i| = O(m)$.

Proof. Split W'_i into two disjoint subsets, W_i^1 and W_i^2. W_i^1 consists of those pattern instances which overlap $tc_{l'_i}$ and W_i^2 consists of pattern instances which do not overlap $tc_{l'_i}$.

By Lemma 4.18, pattern instances in W_i^1 occur only at terminal nodes in the right subtree of l'_i. Therefore, it suffices to show that $\sum_{i=1}^{s} |W_i^2| = O(m)$. From Lemmas 4.24 and 4.25, it follows that $\sum_{i=1}^{s} |W_i^2| = \sum_{i=1}^{s} (2\frac{m}{(3/2)^{s-i+1}}) = O(m)$. □

Consider some i, $1 \leq i \leq s$. The manner in which W_i is maintained so as to facilitate the recovery of Q_x for each terminal node x in the right subtree of marked node l'_i remains to be shown. Clearly, pattern instances in $W_i - W'_i$ occur at all such nodes x and can be stored implicitly in constant space by just storing the rightmost text position compared in the right subtree of l'_i. For the terminal nodes x in l_i's right subtree, we show how to store the pattern instances in $Q_x \cap W'_i$ using a total of $O(|W'_i|)$ space (summing over all x). The linear-space bound then follows from Lemma 4.28.

At each internal node y in the right subtree T of l'_i, a set Com_y is stored. At each terminal node x in T, a set $Spec_x$ is stored. For each $q \in W'_i$, if q occurs only at terminal node x, then it is added to $Spec_x$. Otherwise, if q occurs at more than one terminal node in T, then q is added to the set Com_y, where y is the least common ancestor of those terminal nodes at which q occurs. Clearly, all Com and $Spec$ sets are disjoint and therefore the total space taken by them is $O(|W'_i|)$. The following lemma shows how Q_x can be retrieved from the Com and $Spec$ sets, for each terminal node x in T.

LEMMA 4.29. *For each terminal node $x \in T$, $Q_x = (W_i - W'_i) \cup Com_{y_1} \cup Com_{y_2} \cup \cdots \cup Com_{y_j} \cup Spec_x$, where $y_1, \ldots, y_j$ are the proper ancestors of x in T.*

Proof. The proof follows immediately from Lemma 4.26. □

To compute Q_x as an ordered list, it suffices to maintain each of the Y, Z, Com, and $Spec$ sets as ordered lists which are then appended together in $O(m)$ time according to either Lemma 4.27 or Lemma 4.29, as the case may be.

This concludes the data structure description. We remark that all of the data structures mentioned at the beginning of this section can be computed using naïve algorithms in $O(m^2)$ time.

5. The transfer function f. Before giving the definition of the function f, we prove a number of preliminary lemmas.

5.1. Preliminary lemmas. These lemmas describe some properties of periodic strings and the distribution of text characters compared in Step 2 (the elimination-tree phase) of the presuf handler described in section 4.2.

Let $V = \{p_1, p_2, \ldots, p_k, p_{k+1}\}$. Consider the set of pattern instances in V which are rightmost in their respective groups. Let p_i be a pattern instance in this set. We introduce a function $h(x_i)$ which is central to the analysis.

DEFINITION. *If $i < k$, then $h(x_i)$ is defined by one of the following three cases:*

1. *x_i is periodic. Then x_{i+1} is also periodic. Let u and v be the head and core, respectively, of x_i. Let w be the core of x_{i+1}. $h(x_i)$ is defined to be the suffix of p_1 of length $|v| + |w|$.*

2. *$x_i = uvu$ is not periodic, where $|u|$ is its s-period. Further, x_{i+1} is periodic with core w. $h(x_i)$ is defined to be the suffix of p_1 of length $|v| + |u| + |w|$.*

3. *$x_i = uvu$ is not periodic, where $|u|$ is its s-period. Further, x_{i+1} is not periodic. $h(x_i)$ is defined to be the suffix of p_1 of length $|u|$.*

If $i = k + 1$, then $h(x_i)$ is defined to be the empty string. Note that $i \neq k$ as p_k and p_{k+1} are both in the same group.

The first two lemmas consider the case when $i < k$ and x_{i+1} is periodic with core w. They show that $h(x_i)$ cannot be periodic with core w.

LEMMA 5.1. *Suppose $x_i = uv^2$, where v is the core of x_i. Further, suppose $x_{i+1} = uv = w'w^{k1}$ is periodic with core w, $|w| < |v|$. Then $h(x_i)$ is not periodic with core w.*

Proof. w is a suffix of v. Since v is primitive, $|v|$ is not a multiple of $|w|$. If $h(x_i)$ were periodic with core w, then the prefix of $h(x_i)$ of size $|w|$ would have the form xy, with x a proper suffix of w and y a proper prefix of w. But this prefix of $h(x_i)$ is a suffix of v and hence is the string w. This implies that w is cyclic and cannot be the core of x_{i+1}, a contradiction. □

LEMMA 5.2. *Suppose $x_i = uvu$ is not periodic, where $|u|$ is the s-period of x_i. Suppose the string $x_{i+1} = u = w'w^{k1}$ is periodic with core w, $|w| < |u|$. Then $h(x_i)$ is not periodic with core w.*

Proof. w is a suffix of u. vu is primitive; otherwise, x_i would be periodic. Suppose $h(x_i)$ is periodic with core w. Then $|vu|$ is not a multiple of $|w|$. Therefore, the prefix of $h(x_i)$ of size $|w|$ is of the form xy, with x a proper suffix and y a proper prefix of w. But this prefix of $h(x_i)$ is a suffix of u and hence is the string w. This implies that w is cyclic and cannot be the core of x_{i+1}, a contradiction. □

The next lemma describes the order in which pattern instances in a half-done set are eliminated in Step 2 of the presuf shift handler.

LEMMA 5.3. *Let $p_{i_1}, \ldots, p_{i_r}$, $r \geq 3$, be pattern instances in V comprising a half-done set. For any l, $3 \leq l \leq r$, if p_{i_1} and p_{i_l} both survive at any instant in Step 2, then $p_{i_1}, \ldots, p_{i_{l-1}}$ also survive at that instant.*

Proof. We show that the lemma is true for any instant in Phase 1 and at the end of Phase 1. For Phase 2, the lemma follows from Corollary 4.8.

Consider the rightmost position e such that $p_{i_1}[e]$ is to the left of t_b (recall that t_b is the text character aligned with $p_1[m]$) and different from the character in p_{i_l} aligned with it. The portions of $p_{i_1}, \ldots, p_{i_l}$ whose left and right ends are aligned with $p_{i_1}[e+1]$ and t_b, respectively, are identical and periodic with core v, where v is the core of x_{i_1}. The portions of $p_{i_1}, \ldots, p_{i_{l-1}}$ whose left and right ends are aligned with $p_{i_1}[e]$ and t_b, respectively, are identical. Therefore, a comparison to the right of $p_{i_1}[e]$ eliminates none or all of $p_{i_1}, \ldots, p_{i_l}$ depending upon whether it succeeds or fails. A comparison at $p_{i_1}[e]$ eliminates either p_{i_l} or all of $p_{i_1}, \ldots, p_{i_{l-1}}$. Thus if p_{i_1} and p_{i_l} survive at any instant in Phase 1 or at the end of Phase 1, then all comparisons made until that instant are to the right of $p_{i_1}[e]$. Each of these comparisons eliminates none or all of $p_{i_1}, \ldots, p_{i_l}$. □

The next lemma establishes that if all comparisons in the suffix x_i of p_1 are successful, then at most two pattern instances to the right of p_i survive.

LEMMA 5.4. *Suppose all comparisons made by S_1 within the suffix x_i of p_1 result in matches. Then at most two instances in V among those lying to the right of p_i survive. Further, if two instances p_y and p_z survive, then $\{x_i, x_y, x_z\}$ is a clone set.*

Proof. First, suppose x_i is periodic. Then by the manner in which groups were defined, x_i has the form uv^2. Let $p_a, p_b \in V$, $a, b > i$. If $\{x_i, x_a, x_b\}$ is not a clone set then by Lemma 4.5, successful comparisons in the suffix x_i of p_1 suffice to eliminate one of p_a, p_b. Thus two or more pattern instances in V to the right of p_i can survive

only if their presufs form a clone set with x_i. But the only candidates are the pattern instances p_y and p_z whose presufs are uv and u, respectively.

Second, suppose x_i is not periodic. Then it is of the form uvu, where u is its s-period. For no two pattern instances p_y and p_z, $y, z > i$, can $\{x_i, x_y, x_z\}$ be a clone set. By Lemma 4.5, one of p_y, p_z, for every such y and z, can be eliminated by a successful comparison made within the suffix x_i of p_1. Thus in this case, at most one pattern instance in V to the right of p_i survives. □

The next two lemmas establish that if all comparisons within $h(x_i)$ are successful, then at most two pattern instances in V to the right of p_i survive.

LEMMA 5.5. *Suppose x_{i+1} is periodic with core w and all comparisons made by S_1 within $h(x_i)$ result in matches. Then at most two instances in V among those to the right of x_i survive.*

Proof. Since the case $i = k + 1$ is vacuous, we assume that $i < k$.

The proof is based on Lemmas 5.1, 5.2, 5.3, and 5.4. Let A_s be the group containing p_i. Let $p_{i+1}, \ldots, p_y$ be the pattern instances in group A_{s+1}. Consider the set V' of pattern instances in V which are to the right of p_i and which survive successful comparisons in the suffix x_y of p_1. By Lemma 5.4, with at most one exception (call it p_o), the pattern instances in V' form a half-done set. By Lemma 5.3, the presufs corresponding to the pattern instances in this half-done set comprise the set $\{w'w^{k2}, \ldots, w'w^{k3+1}, w'w^{k3}\}$, where $k3$ equals 0, 1, or 2. Let $V' = \{p_{i_1}, p_{i_2}, \ldots, p_{i_j}, p_o\}$. We show that successful comparisons in $h(x_i)$ eliminate all but at most one of $\{p_{i_1}, p_{i_2}, \ldots, p_{i_j}\}$.

Note that $\{p_{i_1}, p_{i_2}, \ldots, p_{i_j}\}$ is a half-done set. By Lemmas 5.1 and 5.2, the suffix $h(x_i)$ of p_1 (and of x_i) is not periodic with core w. Let the rightmost suffix of p_1 which is longer than $|x_{i+1}|$ and not periodic with core w begin at $p_1[e]$; $p_1[e]$ lies in $h(x_i)$. Consider the largest h, $1 \le h \le j$, such that p_{i_h} survives all comparisons made to the right of $p_1[e]$. Then by Lemma 5.3, $p_{i_1}, \ldots, p_{i_{h-1}}$ also survive these comparisons while $p_{i_{h+1}}, \ldots, p_{i_j}$ are eliminated. If $h \le 1$, then we are done. Otherwise, as shown in the next paragraph, the characters in $p_{i_1}, \ldots, p_{i_{h-1}}$ aligned with $p_1[e]$ are identical to each other yet different from $p_1[e]$. Hence there will be a comparison involving $p_1[e]$, which by assumption is a match; this leaves only p_{i_h} and p_o uneliminated.

Since the rightmost eligible character is always chosen by the elimination strategy for comparison, the portions of $p_{i_1}, \ldots, p_{i_h}$ aligned with the suffix of p_1 which lies to the right of $p_1[e]$ match that suffix. Suppose for some r, $1 \le r < h$, $a = p_{i_r}[c] \ne p_{i_r}[c + |w|] = b$, where $p_{i_r}[c]$ is aligned with $p_1[e]$. Since $p_{i_{r+1}}$ is $|w|$ units to the right of p_{i_r}, the character in $p_{i_{r+1}}$ aligned with $p_{i_r}[c + |w|] = b$ is an a, a contradiction. Therefore, the characters in $p_{i_1}, \ldots, p_{i_{h-1}}$ aligned with $p_1[e]$ are all equal to the character $p_1[e + |w|]$. However, from the definition of e, $p_1[e + |w|] \ne p_1[e]$. This proves the lemma. □

LEMMA 5.6. *Suppose x_{i+1} is not periodic and all comparisons made by S_1 within the suffix $h(x_i)$ of p_1 result in matches. Then at most two instances in V among those to the right of p_i survive.*

Proof. Since the case $i = k + 1$ is vacuous, we assume that $i < k$.

By the manner in which groups were defined, x_i is not periodic. Since x_{i+1} is not periodic, p_{i+1} is the rightmost instance in its group. Thus x_{i+1} cannot form a clone set with any two of its presufs. By Lemma 5.4, at most one pattern instance to the right of p_{i+1} survives successful comparisons in the suffix $x_{i+1} = h(x_i)$ of p_1. □

The following lemma relates the length of the presufs x_i and x_{i+2} with the suffix $h(x_i)$ of p_1 for $i \le k - 1$.

LEMMA 5.7. $|x_{i+2}| + |h(x_i)| \le |x_i|$.

Proof. First, suppose $x_i = uvu$ is not periodic, where u is its s-period. Then $|x_{i+2}| < |u|$. If x_{i+1} is not periodic, then $h(x_i) = u$ and $|x_{i+2}| + |h(x_i)| < 2|u| < |x_i|$. If x_{i+1} is periodic with core w, then $|h(x_i)| = |u| + |v| + |w|$ and $x_{i+2} = |u| - |w|$. This implies that $|x_{i+2}| + |h(x_i)| = |x_i|$.

Next, suppose x_i is periodic with core v and head u. Then x_{i+1} is also periodic, say with core w. Thus $|h(x_i)| = |v|+|w|$ and $|x_{i+2}| = |v|+|u|-|w|$. Then $|x_{i+2}|+|h(x_i)| = 2|v| + |u| = |x_i|$. □

DEFINITIONS. *Let the term* misfit *refer to any character that differs from the rightmost character of p. If $|x_k| > 1$, let r_i be the number of pattern instances in V which lie to the right of p_i. Otherwise, if $|x_k| = 1$, let r_i be one more than the number of pattern instances in V which lie to the right of p_i and do not belong to the rightmost group. For convenience, we define r_i to be 0 if $|x_k| = 1$ and p_i belongs to the rightmost group.*

We provide some lower bounds on the number of occurrences of misfit characters in the presufs of p and in the cores of periodic presufs.

LEMMA 5.8. *Let $|x_k| > 1$. Let p_j, $j \le k$, be any pattern instance in V. Then x_j contains at least r_j instances of the string x_k and hence r_j misfit characters.*

Proof. Since x_k is the smallest nonnull suffix of p that matches a prefix of p, no nonnull suffix of x_k matches a prefix of x_k. Hence all instances of x_k in any string are disjoint. Since x_k itself contains x_k and $r_k = 1$, the lemma is true for $j = k$. Next, suppose $j < k$ and assume inductively that x_{j+1} contains at least r_{j+1} instances of x_k. Then since x_{j+1} is a proper prefix and a proper suffix of x_j, x_j must contain at least $r_{j+1} + 1 = r_j$ instances of x_k. Since the first character of x_k differs from its last character, x_j has at least r_j misfit characters. □

LEMMA 5.9. *Suppose $|x_k| = 1$. Let p_j be any pattern instance in V. Then x_j has at least r_j misfit characters.*

Proof. If p_j belongs to the rightmost group, then $r_j = 0$ and the lemma holds trivially. Therefore, suppose p_j is not in the rightmost group. Let p_y be the rightmost pattern instance not in the rightmost group. x_y contains at least one misfit character; otherwise, it would be in the rightmost group. Since $r_y = 1$, the lemma is true for $j = y$. Next, assume that $j < y$ and assume inductively that x_{j+1} contains at least r_{j+1} misfit characters. Then since x_{j+1} is a proper prefix and a proper suffix of x_j, x_j must have at least $r_{j+1} + 1 = r_j$ misfit characters. □

LEMMA 5.10. *Let p_j be any instance in V and suppose x_j is periodic with head u and core v, $|v| > 1$. Then v contains a misfit character.*

Proof. If v does not contain a misfit character, then x_j does not contain a misfit character either. This implies that all the characters in x_j are identical. This contradicts the assumption that $|v| > 1$. □

We conclude this section of lemmas with two key lemmas, the *h-suffix mapping lemma* and the *half-done set mapping lemma.* In the h-suffix mapping lemma, a set $R_1(i)$ of text characters is defined for each i, $1 \le i \le k-1$, such that p_i is the rightmost instance in its group. In the half-done set mapping lemma, a set $R_2(O)$ of text characters is defined for a half-done set O consisting of pattern instances from V. These two sets are used as ranges for the f function.

Recall that $V = \{p_1, \ldots, p_k\}$ and t_b is the text character aligned with the rightmost character of p_1.

Let $i \le k-1$ and p_i be the rightmost instance in its group. Let h'_i be the suffix of length $|x_{i+2}|$ of the prefix x_i of p. Let $R_1(i)$ be the set of text characters with which $|p$ is aligned when some misfit character in h'_i is aligned with t_b.

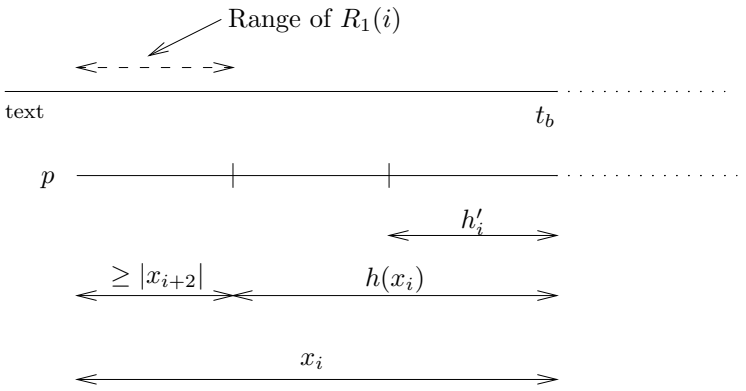


FIG. 5. *The h-suffix mapping lemma.*

LEMMA 5.11 (the h-suffix mapping lemma). $|R_1(i)| \geq r_i - 2$. *All text characters in* $R_1(i)$ *lie strictly to the left of* $|h(x_i)$ *but within the suffix* x_i *of* p_1.

Proof. (See Fig. 5.) By Lemmas 5.8 and 5.9, x_{i+2} and hence h'_i contain at least $r_{i-2} = \max\{0, r_i - 2\}$ misfit characters. Therefore, $|R_1(i)| \geq r_i - 2$. By Lemma 5.7, the left end of any pattern instance in which h'_i overlaps t_b is strictly to the left of $|h(x_i)$ and within the suffix x_i of p_1. □

DEFINITION. *Let* $O \subset V$ *be a half-done set consisting of the pattern instances* $\{p_{h_1}, \ldots, p_{h_j}\}$, $j \geq 3$, $p_{h_1} = p_1$. *Let the head and core of* x_{h_1} *be denoted* u *and* v, *respectively. Let* $v = u'u$. *Suppose* $|v| > 1$. *Further, suppose* p_{h_i} *is* $p_{h_{i-1}}$ *shifted distance* $|v|$ *to the right, for* $1 < i \leq j$. *Let* i_c, $2 \leq c \leq j$, *be the largest index such that* $p_{h_1}[i_c]$ *is different from the character in* p_{h_c} *aligned with it* (*such an index exists by Lemma* 4.7). *Note that* $i_c - i_{c-1} = |v|$, *for all* c, $3 \leq c \leq j$, *and that* $p_{h_1}[i_2] = p_{h_1}[i_3] = \cdots = p_{h_1}[i_j]$. *The text character* t_{i_c} *aligned with* $p_{h_1}[i_c]$ *is called the* characteristic character *of* p_{h_c}.

Let d be the leftmost character in the prefix uu' of p which differs from $p_{h_1}[i_2]$. Define $R_2(t_{i_c})$, $3 \leq c \leq j$, to be the text character with which $|p$ is aligned when d is aligned with t_{i_c}. In addition, define $R_2(O_e)$ to be the set of text characters $R_2(t_{i_c})$, $3 \leq c \leq e \leq j$. For convenience, let $R_2(O)$ denote $R_2(O_j)$.

LEMMA 5.12 (the half-done set mapping lemma). *All text characters in* $R_2(O_j)$ *are distinct.* $R_2(t_{i_c})$ *is aligned with or to the left of* t_{i_c} *and strictly to the right of* $t_{i_{c-1}}$, *for* $3 \leq c \leq j$. *All characters in* $R_2(O_c)$ *are aligned with or to the left of the characteristic character of* p_{h_c}; *in addition, they are strictly to the right of* $|p_{k+1}$, *for* $3 \leq c \leq j$.

Proof. By construction, $R_2(t_{i_c})$ is aligned with or to the left of t_{i_c}, $3 \leq c \leq j$. In addition, $R_2(t_{i_c})$ is at most distance $|v| - 1$ to the left of t_{i_c}. Since $i_c - i_{c-1} = |v|$, $R_2(t_{i_c})$ is strictly to the right of $t_{i_{c-1}}$.

The only part of the lemma still unproven is the claim that all characters in $R_2(O_j)$ are strictly to the right of $|p_{k+1}$. Note that t_{i_3} is distance $|v|$ to the right of t_{i_2}, $R_2(t_{i_3})$ is at most distance $|v| - 1$ to the left of t_{i_3}, and all characters in $R_2(O_j)$ are aligned with or to the right of $R_2(t_{i_3})$. Since t_{i_2} is aligned with or to the right of $|p_{k+1}$, the lemma follows. □

Note that p_{h_c} is eliminated by the time a comparison is made strictly to the left of its characteristic character, for $3 \leq c \leq j$. Further, if a successful comparison

eliminates p_{h_c}, then this comparison must involve its characteristic character.

5.2. The transfer function f. Let C be the set of text characters involved in comparisons in Steps 1 and 2 of the presuf shift handler of section 4.2. For each character $t_c \in C$, with at most two exceptions, we define $f(t_c)$ to be a text character t_d satisfying the following properties.

1. t_d is to the right of $|p_{k+1}$.
2. t_d either coincides with t_c or lies to the left of t_c.
3. The pattern instance whose left end is aligned with t_d is eliminated as a result of comparisons in Steps 1 and 2 of the presuf shift handler.
4. For every distinct $t_{c_1}, t_{c_2} \in C$, $f(t_{c_1}) \neq f(t_{c_2})$.

Furthermore, the mismatches, if any, are always included among the exceptions. We refer to the above properties as Properties 1, 2, 3 and 4, respectively.

Since patterns with $g = 1$ and $|x_k| = 1$ are special-case patterns, we assume that $g > 1$ if $|x_k| = 1$. Further, if $p_1[m]$ does not match the text, then Steps 1 and 2 of the presuf shift handler together make at most one comparison. Therefore, we also assume that $p_1[m]$ matches the text. Let p_l be the rightmost pattern instance in A_1. Let p_r be the rightmost pattern instance in V outside A_g, if any.

We split the sequence C' of comparisons made in Steps 1 and 2 of the presuf shift handler into three disjoint classes as follows.

1. Class 1 consists of the comparison in Step 1. In addition, if $|x_k| = 1$, then Class 1 contains the comparisons which comprise the smallest prefix of C' having the following property: either the last comparison in this prefix is unsuccessful or following that comparison, exactly one pattern instance in A_g survives.

2. Class 2 consists of the comparisons in C' which follow all Class 1 comparisons and are made in the suffix $h(x_l)$ of p_1.

3. Class 3 consists of comparisons in C' which follow all Class 2 comparisons.

Note that if Class 1 contains an unsuccessful comparison, then Class 2 is empty because no further comparisons are made in the suffix $h(x_l)$ of p_1. Thus Classes 1 and 2 together have at most one unsuccessful comparison. The only other possibly unsuccessful comparison is the last comparison in Class 3. We do not define an f value for the last comparison in Class 3. In addition, one other comparison may not receive an f value. If Classes 1 and 2 contain an unsuccessful comparison, then this comparison does not receive an f value. If all comparisons in Classes 1 and 2 are successful, then one successful comparison in one of the three classes may not receive an f value. All other comparisons receive f values. Thus f values are never defined for mismatches and at most two comparisons in C' do not receive f values.

We define f values for each class in turn. f values for Class 2 comparisons are always defined using the set $R_1(l)$. These f values are aligned with the suffix x_l of p_1 and to the left of $h(x_l)$. f values for Class 3 comparisons are defined in one of three ways. If all comparisons in Classes 1 and 2 are successful, then these f values are to the left of the suffix x_l of p_1. If Class 2 contains a mismatch, then these f values are defined using the set $R_1(l)$. If Class 2 is empty and Class 1 contains a mismatch, then these f values are aligned with or to the right of the suffix x_{r+1} of p_1. f values for Classes 2 and 3 are easily seen to be distinct. f values for Class 1 comparisons are aligned either with the suffix x_r of p_1 or with the suffix x_{r-1} of p_1; in Lemma 5.17, we show that these f values do not clash with the f values for Classes 2 and 3.

Classes 2 *and* 3. We consider three cases.

Case 1. Class 2 contains an unsuccessful comparison.

Classes 2 and 3 together contain at most $r_l - 1$ comparisons in addition to this unsuccessful comparison. To see this, note that $r_l - 1$ comparisons in addition to the comparisons in Class 1 suffice to eliminate all but one of the pattern instances in V to the right of p_l. Further, excluding the unsuccessful Class 2 comparison and the last comparison in Class 3, all other comparisons in Classes 2 and 3 are successful. f is defined to map the text characters involved in these $r_l - 2$ successful comparisons to the text characters in $R_1(l)$ in some arbitrary order. By the h-suffix mapping lemma and the fact that all Class 3 comparisons are to the right of $p_1[m]$ in this case, all the text characters in $R_1(l)$ lie to the left of all the text characters involved in Class 2 and Class 3 comparisons. Clearly, Properties 2, 3, and 4 are true for these f values. Property 1 follows from the fact that $|x_l| \leq |x_1| < \frac{m}{2}$, and hence $|p_{k+1}$ is to the left of the suffix x_l of p_1.

Case 2. All comparisons in Classes 1 and 2 are successful.

There are at most r_l comparisons in Class 2, all of which are successful. f is defined to map the text characters involved in $r_l - 2$ of these r_l comparisons to the text characters in $R_1(l)$ in some arbitrary order. As in Case 1, Properties 1, 2, 3, and 4 are satisfied by these f values. This leaves at most s Class 2 comparisons for some $s \leq 2$.

Next, we define f values for Class 3 comparisons and s Class 2 comparisons. These f values will be defined for all comparisons in Class 3 plus the s comparisons in Class 2, with at most two exceptions. These f values will be to the left of the suffix x_l of p_1 and thus clearly distinct from f values for Class 2 comparisons.

Following Class 2 comparisons, at most $\min\{r_l, 2\} - s$ of the pattern instances to the right of p_l survive along with pattern instances in A_1. Let O' denote the following set of $\min\{r_l, 2\}$ pattern instances: those pattern instances to the right of p_l which survive Class 1 and Class 2 comparisons and those pattern instances which are eliminated by one of the s Class 2 comparisons under consideration. Let O refer to the largest half-done set consisting of pattern instances in A_1 and O'. Redefine O' by removing pattern instances in it which are also in O. Considering comparisons which eliminate pattern instances in O and O' is equivalent to considering Class 3 comparisons plus the s Class 2 comparisons.

Let $O = \{p_{h_1}, \ldots, p_{h_e}\}$. If $l = 1$, then the number of comparisons in Class 3 plus s is at most $2 - s + s = 2$. In this case, we do not define f values for the comparisons in Class 3 and the s comparisons in Class 2. Therefore, suppose that $l > 1$. Each successful comparison which eliminates a pattern instance in O involves the characteristic character of the pattern instance eliminated. Let v and u be the core and head, respectively, of x_{h_1} and let $v = u'u$. $|v| > 1$ because either $|x_k| > 1$ or $|x_k| = 1$ and $g > 1$. By Lemma 5.10, v contains a misfit character.

First, consider successful comparisons which eliminate pattern instances in O. If $|O| \leq 2$, there is at most one such comparison in Class 3 and we do not define an f value for it. Therefore, suppose $|O| > 2$. There are two subcases depending on the location of the characteristic character t_{i_e} of p_{h_e}.

Subcase 2a. Either O' is not empty and t_{i_e} is strictly to the left of the left end of the suffix $x_{h_{e-1}}$ of p_1 or O' is empty and t_{i_e} is strictly to the left of the left end of the suffix $x_{h_{e-2}}$ of p_1.

For $3 \leq c \leq e$, if a successful comparison is made at t_{i_c}, $f(t_{i_c})$ is defined to be $R_2(t_{i_c})$. By the half-done set mapping lemma, these f values are strictly to the left of the suffix $x_{h_{e-1}}$ of p_1 if O' is not empty and strictly to the left of the suffix $x_{h_{e-2}}$ of p_1 if O' is empty. A simple case analysis (O' equals 0, 1, 2) shows that these f values

are strictly to the left of the left end of the suffix x_l of p_1, as claimed. Properties 1, 2, and 4 for these f values follow easily from the half-done set mapping lemma while Property 3 follows from the definition of the set $R_2(O)$. At most one successful comparison eliminating pattern instances in O does not have an f value: the one eliminating p_{h_2}.

Subcase 2b. Either O' is not empty and t_{i_e} is aligned with some character in the suffix $x_{h_{e-1}}$ of p_1 or O' is empty and the characteristic character of p_{h_e} is aligned with some character in the suffix $x_{h_{e-2}}$ of p_1.

In the first case, t_{i_c}, the characteristic character of p_{h_c}, is aligned with some character in the suffix $x_{h_{c-1}}$ of p_1, for $2 \leq c \leq e$. Consider the set R'_2 of $e-2$ text characters with which $|p$ is aligned when the rightmost misfit character in the prefix x_{h_c} of p, $1 \leq c \leq e-2$, is aligned with t_b. Since v contains a misfit character, the cth leftmost text character in R'_2 is aligned with some character in the suffix x_{h_c} of p_1 and is strictly to the left of the left end of the suffix $x_{h_{c+1}}$ of p_1. A successful comparison at t_{i_c}, $3 \leq c \leq e$, is mapped by f to the $(c-2)$nd leftmost character in R'_2. Clearly, all characters in R'_2 are distinct and $f(t_{i_c})$ is strictly to the left of t_{i_c}. All characters in R'_2 are aligned with some character in the suffix x_1 of p_1. Thus Properties 1, 2, 3, and 4 are satisfied by these f values. These f values are strictly to the left of the left end of the suffix $x_{h_{e-1}}$ of p_1. Since O' is not empty, $h_{e-1} \leq l$. Therefore, these f values are strictly to the left of the left end of the suffix x_l of p_1. Again, the only successful comparison without an f value, if any, is the one eliminating p_{h_2}.

In the second case, t_{i_c}, $4 \leq c \leq e$, is aligned with some character in the suffix $x_{h_{c-2}}$ of p_1. f values are not defined for the two leftmost comparisons under consideration. The remaining comparisons involve text characters aligned with some character in the suffix x_{h_2} of p_1. A successful comparison at t_{i_c}, $4 \leq c \leq e$, is mapped by f to the $(c-3)$rd leftmost character in R'_2. Clearly, $f(t_{i_c})$ is strictly to the left of t_{i_c}. As in the first case, Properties 1, 2, 3, and 4 are satisfied by these f values. All of these f values are to the left of the left end of the suffix $x_{h_{e-2}}$ of p_1. Since $h_{e-2} \leq l$ for this case, these f values are strictly to the left of the left end of the suffix x_l of p_1. The only successful comparisons without f values, if any, are those eliminating p_{h_3} and p_{h_2}. This ends Subcase 2b.

Before we define f values for comparisons which eliminate pattern instances in O', we need a lemma which will be used later when Class 3 comparisons are defined. This lemma can be verified easily from the above description.

LEMMA 5.13. *If $O \subset A_1$ and $|O'| \leq 1$, then the f values defined in Subcases* 2a *and* 2b *are to the left of the suffix x_{l-1} of p_1.*

Next, consider successful comparisons which eliminate pattern instances in O'. If $|O'| < 2$ or no successful comparison eliminates a pattern instance in O', then no further f values are defined. Therefore, suppose $|O'| = 2$ and a successful comparison is made to eliminate one of the pattern instances in O'. By Lemma 4.5, this comparison involves a text character t_c which is aligned with some character in the suffix x_{h_e} of p_1. $f(t_c)$ is defined to be the text character with which $|p$ is aligned when the rightmost misfit character in the prefix $x_{h_{e-1}}$ of p_1 is aligned with t_b. Since v contains a misfit character, $f(t_c)$ is aligned with some character in the suffix $x_{h_{e-1}}$ of p_1 and is strictly to the left of the suffix x_{h_e} of p_1. $f(t_c)$ is thus to the right of and distinct from all f values defined previously for comparisons which eliminate pattern instances in O. Further, $f(t_c)$ is to the left of t_c because t_c is aligned with the suffix x_{h_e} of p_1. Since $|O'| = 2$, $l = h_e$, and therefore $f(t_c)$ is strictly to the left of the suffix x_l of p_1, as claimed. Now Properties 1, 2, 3, and 4 are easily seen to be true for all Class 2

and 3 comparisons.

LEMMA 5.14. *For Case* 2, f *values have been defined for all but two of the comparisons in Classes* 2 *and* 3. *Further, the omitted comparisons include mismatches, if any.*

Proof. We just need to show that at most two of the comparisons among those which eliminate pattern instances in O and O' do not receive f values, for the mismatches never receive f values.

If $|O| < 2$, then there are at most two comparisons which eliminate pattern instances in O and O'. If O' is empty, then f values are defined for all but the last two comparisons which eliminate pattern instances in O. Therefore, suppose O' is not empty and $|O| \geq 2$. The only possible successful comparisons for which an f value might not be defined are those which eliminate p_{h_2} or one of the pattern instances in O'. There are two cases.

First, suppose $|O'| = 1$. Let $O' = \{p_z\}$. The only possible successful comparisons for which an f value is not defined are those which eliminate p_{h_2} or p_z. We show that if one of these successful comparisons actually occurs, then there can be at most one mismatch, and if both these successful comparisons occur, then there are no mismatches. (Recall that all comparisons in Classes 1 and 2 are successful.) Suppose p_{h_2} is eliminated by a successful comparison. p_{h_1} must be alive immediately before this comparison and $p_{h_3} \ldots p_{h_e}$ must have been eliminated prior to this comparison. This implies that no mismatch could have occurred before this comparison and only the pattern instances p_{h_1} and p_z survive this comparison. Therefore, if p_{h_2} is eliminated by a successful comparison, then there is at most one unsuccessful comparison, and if both p_{h_2} and p_z are eliminated by successful comparisons, then there are no unsuccessful comparisons. Next, suppose p_z is eliminated by a successful comparison but no successful comparison eliminates p_{h_2}. Each of the other comparisons in Class 3 eliminates some pattern instance in O and the first such unsuccessful comparison eliminates all but one of the instances in O. Therefore, there is at most one unsuccessful comparison in this case.

Second, suppose $|O| \geq 2$ and $|O'| = 2$. If one of the pattern instances in O' is eliminated by a successful Class 2 or 3 comparison, then an f value is defined for this comparison. From this point onwards, $|O| \geq 2$ and $|O'| = 1$. Therefore, the argument in the previous paragraph applies. On the other hand, if no successful Class 2 or 3 comparison eliminates a pattern instance in O', then the first comparison in Class 3 must be unsuccessful. This comparison leaves at most two pattern instances uneliminated and thus there are at most two comparisons which eliminate pattern instances in O and O'. □

Case 3. Class 1 contains an unsuccessful comparison.

In this case, $|x_k| = 1$ and Class 2 is empty as mentioned before. We define f values for all but the last of the comparisons in Class 3. These f values are aligned with or to the right of the suffix x_{r+1} of p_1. All Class 3 comparisons are made to the right of $p_1[m]$ in this case. Each such successful comparison matches an instance of the character x_k in p_{r+1} against a text character t_c; t_c is aligned with a non-x_k character in some pattern instance p_s, $s > r$. f is defined to map a text character t_c matched successfully by a Class 3 comparison to the text character with which $|p$ is aligned when the leftmost non-x_k character in p is aligned with t_c. Clearly, these f values are aligned with or to the right of the suffix x_{r+1} of p_1 and Properties 1, 2, 3, and 4 are satisfied by these f values.

This finishes the description of the f function for Classes 2 and 3. The following lemma is obvious from the above description.

LEMMA 5.15. *f values for Class 2 and 3 comparisons belong to one of the following sets of text characters:*

(i) *the set $R_1(l)$;*
(ii) *the set of text characters to the left of the suffix x_l of p_1 and to the right of $|p_{k+1}$;*
(iii) *the set of text characters aligned with or to the right of the suffix x_{r+1} of p_1.*

Further, an f value can be in set (i) *only if $r_l - 2 > 0$ and in set* (iii) *only if Class* 1 *contains an unsuccessful comparison.*

Class 1. We consider two cases, $|x_k| > 1$ and $|x_k| = 1$.

Case 1. $|x_k| > 1$.

The only comparison in Class 1 matches t_b with $p_1[m]$. $f(t_b)$ is defined to be t_b. This mapping satisfies Property 3 because the leftmost character in p is a misfit character in this case. If $g = 1$, then all other comparisons in C' are made to the left of t_b, and therefore all other f values are to the left of t_b. If $g > 1$, then all other f values are either to the left of the suffix x_l of p_1 or to the left of the suffix $h(x_l)$ of p_1. Since $|h(x_l)| \geq 1$ if $g > 1$, these f values are to the left of t_b. Therefore, Property 4 is satisfied by all f values. Properties 1 and 2 are obvious for $f(t_b)$.

Case 2. $|x_k| = 1$.

$g > 1$ by assumption. f values are defined for all comparisons in Class 1 unless either Class 1 contains an unsuccessful comparison or $r = l = 1$. If Class 1 contains an unsuccessful comparison or if $r = l = 1$, then one comparison in Class 1 does not receive an f value. However, in these cases, there is at most one comparison in Classes 2 and 3 for which an f value was not defined earlier. To see this, note that if the last Class 1 comparison is successful and $r = l = 1$, then Classes 2 and 3 together can have at most one comparison, and if the last Class 1 comparison is unsuccessful, then Class 2 is empty and Case 3 must hold for Class 3 comparisons.

All f values defined for Class 1 comparisons will be to the left of the suffix x_{r+1} of p_1 and aligned with either the suffix x_{r-1} of p_1 or the suffix x_r of p_1. Clearly, if $p_l \neq p_r, p_{r-1}$, then these f values are distinct from all f values defined earlier for Class 2 and 3 comparisons. If $p_l = p_{r-1}$, then $r_l - 2 = 0$ and all f values for Class 2 and 3 comparisons are either to the left of the suffix x_{r-1} of p_1 or aligned with or to the right of the suffix x_{r+1} of p_1. Therefore, f values for comparisons in Class 1 are distinct from f values for comparisons in Classes 2 and 3 in this case. If $p_l = p_r$, then $r_l - 2 < 0$ and, by Lemma 5.15, all f values for Class 2 and 3 comparisons are either to the left of the suffix x_r of p_1 or aligned with or to the right of the suffix x_{r+1} of p_1. In this case, we show that if an f value for some Class 1 comparison is aligned with the suffix x_{r-1} of p_1 and to the left of the suffix x_r of p_1, then all f values for Class 2 and Class 3 comparisons are to the left of the suffix x_{r-1} of p_1. Thus all f values are distinct.

The following lemma describes the distribution of Class 1 comparisons.

LEMMA 5.16. *Let d be the rightmost misfit character in p_1. Each Class* 1 *comparison involves a text character which is aligned with or to the right of d.*

Proof. Suppose two pattern instances $p_{j_1}, p_{j_2} \in A_g$, $j_1 < j_2$, are left uneliminated by comparisons made at or to the right of d. Let c_{j_1} and c_{j_2} be the portions of p_{j_1} and p_{j_2}, respectively, which overlap the suffix z of p_1 starting at d. Then $c_{j_1} = c_{j_2} = z$. Consider the characters in p_{j_1} and p_{j_2} aligned with the $(j_2 - j_1)$th character to the right of d in p_1. Clearly, the first of these matches x_k while the second is a misfit

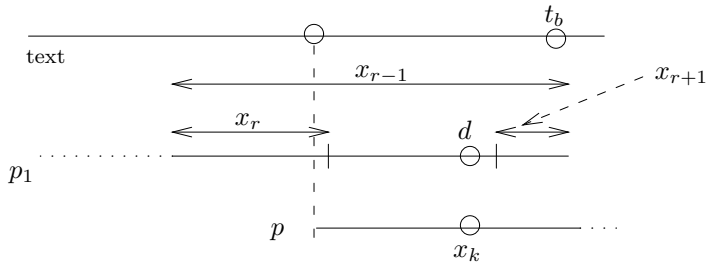


FIG. 6. *The set* R_3, $r > 1$.

character. This is a contradiction. □

Note that x_r contains a misfit character. Further, its suffix and prefix x_{r+1} are disjoint. Each contains at least $k - r$ instances of x_k. We define a set R_3 of text characters which serves as the range of f values for Class 1 comparisons. The definition has the following property. All characters in R_3 are to the left of the suffix x_{r+1} of p_1. If all successful Class 1 comparisons are made to the right of d or if $r = 1$, then all characters in R_3 are aligned with the suffix x_r of p_1. If a successful Class 1 comparison is made at d and $r > 1$, then characters in R_3 are aligned with the suffix x_{r-1} of p_1.

First, suppose all successful Class 1 comparisons are made to the right of d. Each successful comparison matches an occurrence of x_k to the right of d against the text. R_3 is defined to be the set of text characters with which $|p$ is aligned when d', the leftmost misfit character in p, is aligned with one of the text characters matched by a Class 1 comparison. Clearly, all characters in R_3 are aligned with the suffix x_r of p_1. f is defined to map the text characters compared by Class 1 comparisons to the text characters in R_3 in some arbitrary order. Properties 2, 3, and 4 readily follow for these f values. Property 1 follows from the fact that $|x_r| \leq |x_1| < \frac{m}{2}$.

Next, suppose a successful Class 1 comparison is made at d. In this case, all comparisons in Class 1 are successful and there are at most $k + 1 - r$ comparisons in Class 1. R_3 is defined differently depending upon whether $r = 1$ or $r > 1$.

First, suppose $r = 1$. R_3 is defined to contain the $k - r$ text characters with which $|p$ is aligned when one of $k - r$ instances of x_k to the left of d' (recall that d' is the leftmost misfit character in p) is aligned with d. The text characters in R_3 are clearly aligned with the suffix x_r of p_1. f is defined to map up to $k - r$ of the text characters compared by Class 1 comparisons to the $k - r$ text characters in R_3 in some arbitrary order. All of these f values are distinct and are aligned with or to the left of d. Properties 2, 3, and 4 immediately follow for these f values. Property 1 follows from the fact that $|x_r| \leq |x_1| < \frac{m}{2}$.

Next, suppose $r > 1$. See Fig. 6. When p is placed with $|p$ aligned with the left end of the suffix x_{r-1} of p_1, there exist at least $2(k - r) \geq k - r + 1$ instances of x_k to the left of d in p. R_3 is defined to be the set of $2(k - r)$ text characters with which $|p$ is aligned when one of these $2(k - r)$ instances of x_k is aligned with d. Clearly, characters in R_3 are aligned with the suffix x_{r-1} of p_1. f is defined to map the text characters compared by Class 1 comparisons to some $k - r + 1$ of the $2(k - r)$ text characters in R_3 in some arbitrary order. Properties 2 and 3 readily follow for these f values. Property 1 follows from the fact that $|x_{r-1}| \leq |x_1| < \frac{m}{2}$. The distinctness of these f values from the f values for Classes 2 and 3 follows from the following lemma.

LEMMA 5.17. *If $r > 1$, $p_l = p_r$, and a successful Class 1 comparison is made at d, then f values for Class 2 and 3 comparisons are to the left of the suffix x_{r-1} of p_1.*

Proof. At most one pattern instance $p_{r'} \in A_g$ survives a successful comparison at d. Further, Class 1 does not contain an unsuccessful comparison. Since $r_l = 1$, Class 2 contains at most one comparison. If this comparison is unsuccessful, then only $p_{r'}$ survives; no f values are defined for Class 2 or 3 comparisons in this case. If the sole Class 2 comparison is a successful one, then Case 2 must hold for all Class 2 and 3 comparisons. Since $r_l = 1$, $r_l - 2 < 0$ and, by Lemma 5.15, no f values are defined using the set $R_1(l)$. Therefore, all f values for Class 2 and 3 comparisons in this case are defined as in Subcases 2a and 2b. Refer to these subcases. Note that, in this case, O consists of the pattern instances in A_1 and $O' = \{p_{r'}\}$. From Lemma 5.13, all f values for Class 2 and 3 comparisons are to the left of the suffix x_{r-1} of p_1 in this case. □

This concludes the definition of the f function.

6. Presuf shifts with $|x'_1| \geq \frac{m}{2}$. This case can occur only for periodic patterns. Therefore, assume that p is periodic and has the form $u_p v_p^{i_p}$, where v_p and u_p are the core and head of p, respectively, and $i_p \geq 2$.

Recall that the lower bound of $\frac{m+1}{2}$ on the distance between consecutive presuf shifts was crucial in deriving the comparison complexity for the case where $|x'_1| < \frac{m}{2}$. This lower bound does not hold if $|x'_1| \geq \frac{m}{2}$. Consecutive presuf shifts can occur distance $|v_p| \ll \frac{m+1}{2}$ apart. Even a single mismatch per presuf shift leads to a large comparison complexity. Since the problem in this case lies only in the frequency of occurrence of presuf shifts, we use the same basic algorithm, changing only the presuf shift handler. The new presuf shift handler ensures that either two consecutive presuf shifts are at least distance $\frac{m+1}{2}$ apart or no mismatch occurs between consecutive presuf shifts. In fact, we show the following stronger claim about the performance of the presuf shift handler. A presuf shift has overhead 0 if the next presuf shift occurs a distance less than $\frac{m+1}{2}$ ahead, overhead 1 if the next presuf shift occurs a distance less than $\frac{3(m+1)}{4}$ ahead, and overhead at most 2 otherwise. A comparison complexity of $n(1 + \frac{8}{3(m+1)})$ comparisons follows.

6.1. The presuf shift handler for $|x'_1| \geq \frac{m}{2}$. Before describing the presuf shift handler, we recall some definitions and assumptions made in section 4. Let t_A refer to the portion of the text with which the prefix x'_1 of p is aligned following the shift. We assume that prefix x'_1 of p matches t_A following a presuf shift and that the variable t_{last} has been appropriately set to prevent this assumption from leading to an incorrect inference. Let t_a refer to the rightmost character in t_A.

As for the case where $|x'_1| < \frac{m}{2}$, the presuf shift handler considers all presuf pattern instances, i.e., those pattern instances in which a prefix (possibly null) of p matches some suffix of t_A. In a presuf pattern instance, the prefix matching a suffix of t_A is a presuf of p and is called the presuf corresponding to this presuf pattern instance. Presuf pattern instances are of two types. The first type consists of those presuf pattern instances whose corresponding presufs have the form $u_p v_p^l$, $1 \leq l \leq i_p - 1$. The second type consists of those presuf pattern instances whose corresponding presufs are less than $|v_p|$ in length. We identify a presuf pattern instance p'_α of the second type as follows. If $|u_p| > 0$, then p'_α is the presuf pattern instance corresponding to the presuf u_p. If $|u_p| = 0$, then p'_α is the presuf pattern instance corresponding to the null presuf; i.e., $|p'_\alpha$ is to the immediate right of t_a. The following observation

enables us to work with only presuf pattern instances of the second type while making comparisons within a text window γ of length $|v_p|$ to the right of t_a.

LEMMA 6.1. *A presuf pattern instance of the first type matches all text characters in the window γ if and only if p'_α matches all characters in that window.*

Proof. The portion of p'_α which overlaps γ is identical to v_p, as is the corresponding portion of any presuf pattern instance of the first type. □

If p'_α is eliminated by comparisons in γ, then so are all presuf pattern instances of the first type. This forces the next presuf shift to occur at least distance $m - |v_p| \geq \frac{m}{2} + 1$ to the right. If p'_α is not eliminated by comparisons in γ, then presuf pattern instances of the first type also survive, and therefore the next presuf shift can occur as little as distance $|v_p|$ to the right. In this case, it is important to ensure that no mismatches are made in γ.

The presuf shift handler has five steps and works broadly as follows. As in the presuf shift handler of section 4.2, the first two steps identify a presuf pattern instance p'_e with the following property: all presuf pattern instances that survive the first two steps are presuf overlaps of p'_e. This is accomplished by making comparisons in a manner similar to the presuf shift handler of section 4.2, but with a single difference. This difference is aimed at ensuring that the first mismatch eliminates p'_α. Steps 3, 4, and 5 are, however, identical to the corresponding steps of the earlier presuf shift handler.

Steps 1 and 2 proceed as follows to determine p'_e. They consider only presuf pattern instances of the second type and eliminate all but one of these. The survivor determines p'_e; i.e., if the survivor is p'_α, then p'_e is the leftmost presuf pattern instance, and otherwise p'_e is the survivor itself. We show that in order to eliminate among presuf pattern instances of the second kind, it suffices to consider suitable prefixes of these presuf pattern instances. We need the following definitions in order to describe Steps 1 and 2 in detail. Consider the leftmost presuf pattern instance of the second type and let β be the length of the corresponding presuf. Suppose there are $k'' + 1$ presuf pattern instances of the second type. We define $p''_1, \ldots, p''_{k''}, p''_{k''+1}$ such that p''_j, $1 \leq j \leq k'' + 1$, is the prefix of length $m'' = \beta + |v_p|$ of the jth leftmost presuf pattern instance of the second type. Let x''_j, $1 \leq j \leq k'' + 1$, be the presuf corresponding to the jth leftmost presuf pattern instance of the second type. We call x''_j the presuf corresponding to p''_j. Let p''_α refer to the length m'' prefix of p'_α and p'' refer to the length m'' prefix of p. The following lemma shows that in order to eliminate all but one of the presuf pattern instances of the second kind, it suffices to consider only $p''_1, \ldots, p''_{k''}, p''_{k''+1}$.

LEMMA 6.2. *At most one of $p''_1, \ldots, p''_{k''}, p''_{k''+1}$ matches γ.*

Proof. Let x and y be the portions overlapping γ in some two of $p''_1, \ldots, p''_{k''+1}$. Then x and y are different cyclic shifts of v_p. If $x = y$, then v_p is cyclic, a contradiction. □

Let $V'' = \{p''_1, \ldots, p''_{k''+1}\}$. Note that Lemmas 4.3–4.7 continue to hold if p_j, x_j, V, p, and m are replaced by p''_j, x''_j, V'', p'', and m'', respectively, for $1 \leq j \leq k'' + 1$. Henceforth, these substitutions are implicit in all references to these lemmas. The elements in V'' are divided into groups $A''_1, \ldots, A''_{g''}$ in accordance with Lemma 4.3.

Remark. The presuf shift handler being described does not work for patterns for which $|x''_{k''}| = 1$ and $g'' = 1$. Presuf shifts for these exception patterns are handled separately in section 6.5.

With this background, we describe the five steps of the presuf shift handler.

Step 1. The characters in $p''_1, \ldots, p''_{k''}$ aligned with $p''_1[m'']$, the rightmost character

in p''_1, are identical. If the character in $p''_{k''+1}$ aligned with $p''_1[m]$ is also identical to it, then $p''_1[m]$ is compared with the aligned text character. A mismatch eliminates all of $p''_1, \ldots, p''_{k''}, p''_{k''+1}$ and the basic algorithm is restarted with $|p$ placed immediately to the right of $|p_{k''+1}$. A match leads to Step 2.

Step 2. All but one of $p''_1, \ldots, p''_{k''+1}$ are eliminated in this step by making up to k'' comparisons, at most two of which are unsuccessful. Further, p''_α is eliminated by the first unsuccessful comparison. As in Step 2 of section 4.2, there are two phases.

Phase 1 is identical to Phase 1 of section 4.2; i.e., at every step the rightmost character c in p''_1 having the following property is compared with the aligned text character: the character aligned with c in at least one of the surviving elements in V'' is different from c. By Lemma 4.6, the outcome of Phase 1 is a half-done set O.

LEMMA 6.3. *$p''_\alpha \in O$ if and only if all of the comparisons in Phase* 1 *are successful.*

Proof. All comparisons in Phase 1 are made in the suffix x''_1 of p_1 because, by Lemma 4.5, successful comparisons in that suffix leave a half-done set uneliminated. x''_1 is a presuf of p''_1 and therefore a suffix of v_p. By the manner in which p'_α is defined, the portion of p''_α that overlaps γ is identical to the string v_p. Therefore, a mismatch in Phase 1 eliminates both p''_1 and p''_α while a match in Phase 1 eliminates neither. The lemma follows. □

If Phase 1 ends with a mismatch or if $p''_\alpha = p''_1$, then Phase 2 is identical to Phase 2 of section 4.2; i.e., all but one of the elements in O are eliminated by making comparisons according to a right-to-left sequence. Note that in both cases, the first mismatch eliminates p''_α. Otherwise, if $p''_\alpha \neq p''_1$ and all comparisons in Phase 1 are successful, then we modify Phase 2 as follows so as to ensure that the first mismatch eliminates p''_α.

Phase 2 proceeds exactly as Phase 2 of Step 2 in section 4.2 until p''_α becomes the rightmost element in O. Any mismatch in this process terminates Phase 2 and eliminates p''_α and all elements in O to the left of p''_α. If no mismatch occurs in this process, then let the surviving elements in O be $\{p''_{h_1}, \ldots, p''_{h_e}\}$, where $p''_{h_1} = p''_1$ and $p''_{h_e} = p''_\alpha$. These elements are eliminated using a left-to-right sequence of comparisons instead of the right-to-left sequence used in Step 2 of section 4.2. This left-to-right sequence ensures that a mismatch eliminates p''_α. Let d_e be the leftmost character in p''_{h_e} such that p''_{h_e} differs from the aligned character in $p''_{h_{e-1}}$. By Lemma 6.2, d_e is aligned with or to the left of $p''_1[m'']$ and to the right of t_a. If $e = 2$, then a comparison at d_e terminates Phase 2 with a mismatch eliminating p''_{h_e} and a match eliminating p''_{h_1}. Suppose $e > 2$. Then x''_{h_1} is periodic with core, say, v. Let d_j, $2 \leq j \leq e-1$, be the character in p''_{h_e} which is distance $(e-j)|v|$ to the left of d_e. The characters $d_2, \ldots, d_e$ are compared with the aligned text characters in sequence until either a mismatch occurs or the sequence is exhausted. The following lemma shows that at most one element of O survives these comparisons.

LEMMA 6.4. *A mismatch at d_j leaves only $p_{i_{j-1}}$ uneliminated. A match at d_j eliminates $p_{i_{j-1}}$.*

Proof. If $e = 2$, then the lemma is clearly true. Suppose $e > 2$. From the definition of d_e, it follows that the prefix of p''_{h_e} ending at d_e is periodic with core of size $|v|$ while the prefix of $p''_{h_{e-1}}$ is not; therefore, $d_2 = d_3 = \cdots = d_e \neq d_{e+1}$, where d_{e+1} is the character which is distance $|v|$ to the right of d_e. It follows that the characters in $p''_{h_j}, \ldots, p''_{h_e}$ aligned with d_j are identical to each other but different from the character in $p''_{h_{j-1}}$ aligned with d_j. Therefore, a match at d_j eliminates $p''_{h_{j-1}}$. A mismatch at d_j eliminates $p''_{h_j}, \ldots, p''_{h_e}$ and the preceding successful comparisons at

$d_2, \ldots, d_{j-1}$ eliminate $p''_{h_1}, \ldots, p''_{h_{j-2}}$. The lemma follows. □

This completes Step 2. At most k'' comparisons are made in this step, at most two of which result in mismatches. Further, p''_α survives only if there are no mismatches. The sequence of comparisons made in Step 2 can be represented by a tree ET'', akin to the tree ET of section 4.2. The only difference between ET'' and ET is that the sequence corresponding to a portion of Phase 2 may now be a left-to-right sequence if Phase 1 does not end in a mismatch and $p''_\alpha \neq p''_1$. We conclude Step 2 with the following lemma.

LEMMA 6.5. *All but at most one of $p''_1, \ldots, p''_{k''}, p''_{k''+1}$ can be eliminated by making up to k'' comparisons using the $O(k'')$-sized binary comparison tree ET''. At most two of these comparisons result in mismatches. If p''_α survives, then no comparisons result in mismatches. Moreover, the sequence of comparisons made by the elimination strategy consists of two sequences: a right-to-left sequence followed by either another right-to-left sequence or a left-to-right sequence.*

We describe Steps 3, 4, and 5 next. Let p''_e be the only element of V'' to survive Steps 1 and 2. If $p''_e = p''_\alpha$, then define p'_e to be the leftmost presuf pattern instance. If $p''_e \neq p''_\alpha$, then let p'_e be the presuf pattern instance of which p''_e is a prefix; i.e., p'_e and p''_e have their left ends aligned. Clearly, p'_e is the leftmost presuf pattern instance to survive Steps 1 and 2. Let Q denote the set of pattern instances which overlap p'_e and have their left end to the right of $|p''_{k''+1}$. In the elimination process, some elements of Q may also have been eliminated from being potential matches. They need not be reconsidered. To this end, a subset Q_x of Q consisting of pattern instances consistent with comparisons in Steps 1 and 2 is associated with each terminal node x in ET''. The maintenance of Q_x is similar to the description in section 4.5 and is described in section 6.4. Suppose that the elimination process terminates at terminal node x. Let $Q' = \{p'_e\} \cup Q_x$. Steps 3, 4, and 5 are now identical to the corresponding steps in section 4.2.

6.2. Comparison complexity. In order to determine the comparison complexity, we need to define a transfer function f'' akin to the transfer function f defined in section 5. We state the following lemma describing the properties of f''. The proof of this lemma is deferred to section 6.3.

LEMMA 6.6. *Let C be the set of text characters involved in comparisons in Steps 1 and 2 of the presuf shift handler of section 6.1. For each character $t_c \in C$, with at most two exceptions, there exists a text character $f''(t_c) = t_d$ satisfying the following properties:*

1. *t_d is to the right of $|p''_{k''+1}$.*
2. *t_d either coincides with t_c or lies to the left of t_c.*
3. *The pattern instance whose left end is aligned with t_d is eliminated as a result of comparisons in Steps 1 and 2 of the presuf shift handler.*
4. *For every distinct $t_{c_1}, t_{c_2} \in C$, $f(t_{c_1}) \neq f(t_{c_2})$.*

Furthermore, mismatches, if any, are always included among the exceptions.

The following lemma determines the comparison complexity of the algorithm.

LEMMA 6.7. *If p is not a special-case pattern, then the comparison complexity of the algorithm is bounded by $n(1 + \frac{8}{3(m+1)})$.*

Proof. A presuf shift occurs either with $|x'_1| < \frac{m}{2}$ or with $|x'_1| \geq \frac{m}{2}$. In the former case, it was shown in Lemma 4.12 that a presuf shift can have overhead at most two and that an overhead of two implies that the next presuf shift occurs at least distance $\frac{3(m+1)}{4}$ to the right. Further, $\frac{m+1}{2}$ is a lower bound on the distance between two consecutive presuf shifts in this case. We show similar properties for presuf shifts

with $|x'_1| \geq \frac{m}{2}$. Specifically, we show that a presuf shift can have overhead at most two. Further, we show that an overhead of one forces the next presuf shift to occur at least distance $\frac{m+1}{2}$ to the right and an overhead of two forces the next presuf shift to occur at least distance $\frac{3(m+1)}{4}$ to the right. We show the above by giving a charging scheme for the presuf shift handler of section 6.1. The comparison complexity of the algorithm now follows.

Charging scheme. As in Lemma 4.12, the run of the algorithm is divided into phases; a phase can be of one of four types. The ranges of the text characters charged in each phase type remain exactly the same as in Lemma 4.12. The charging scheme for Type 1 and Type 2 phases also remains exactly the same. Only the charging scheme for Type 3 and Type 4 phases is modified in accordance with the presuf shift handler of section 6.1.

We consider a single phase, which could be a Type 3 or a Type 4 phase. We assume that this phase begins with a presuf shift with $|x'_1| \geq \frac{m}{2}$. Let q_1 and q_2 refer to the leftmost surviving pattern instances at the beginning and end of that phase, respectively. Note that q_1 is a presuf overlap of the pattern instance q', the leftmost uneliminated pattern instance prior to the presuf shift which initiated this phase. Specifically, the prefix x'_1 of q_1 is aligned with the suffix x'_1 of q' (recall that on a presuf shift, we assume that the prefix x'_1 of q_1 matches the text). Recall that t_a is the text character aligned with $q'|$.

Consider the comparisons made by the current use of the presuf shift handler of section 6.1. If a mismatch occurs in Step 1, the current phase ends immediately and the basic algorithm is resumed. The presuf shift in this case has overhead one and the next presuf shift occurs at least distance $m + 1$ to the right. Next, suppose that the comparison in Step 1 is successful. Let p'_e be the presuf pattern instance to survive the elimination using tree ET'' in Step 2. After the presuf shift handler finishes, one of three scenarios ensues. We consider each in turn.

1. All pattern instances overlapping p'_e are eliminated apart from its presuf overlaps, and p'_e or at least a suffix of p'_e is matched. This is a Type 4 phase. We consider two cases, depending upon whether p'_e is a presuf pattern instance of the first or the second type.

First, suppose p'_e is of the first type; i.e., it is the leftmost presuf pattern instance. Then no mismatches are made in Steps 1, 2, 3, 4, or 5. All comparisons made by the presuf shift handler are charged to the text characters compared. The bit vector BV ensures that each of these comparisons involves a different text character. Thus each text character which lies to the right of t_a and is aligned with or to the left of $p'_e|$ is charged at most once. In this case, the overhead of this presuf shift is zero.

Next, suppose p'_e is of the second type. Then p''_α is eliminated in Step 2. All comparisons in Steps 1, 3, 4, and 5 and all but at most two comparisons in Step 2 are successful. Each successful comparison is charged to the text character compared. The bit vector BV ensures that each of these comparisons involves a different text character. Thus each text character which lies to the right of t_a and is aligned with or to the left of $p'_e|$ is charged at most once. At most two comparisons in Step 2 are unsuccessful, so this shift has overhead at most two. If there are two mismatches in Step 2, then we claim that p''_1 is eliminated; in addition, if x''_1 is periodic, with core v and head u, say, then all elements in V'' whose associated presufs have the form uv^o, $o \geq 1$, are also eliminated. (This can be shown in a manner similar to the corresponding proof in Lemma 4.12.) Let p''_e be the m''-length prefix of p'_e and let x''_e be the presuf associated with p''_e. From the above, it follows that $x''_1 = x''_e z x''_e$, for

some nonempty string z. Since $|x''_1| < |v_p|$, $p''_e = x''_1 w x''_1$ for some nonempty string w. Therefore, $|x''_e| \leq \frac{m''-3}{4} \leq \frac{m-3}{4}$. This guarantees that the next presuf shift occurs at least distance $\frac{3(m+1)}{4}$ to the right. If there is just one mismatch in Step 2, then since $|x''_1| < |v_p| \leq \frac{m}{2}$, the next presuf shift occurs at least distance $\frac{m+1}{2}$ to the right.

2. p'_e is eliminated. In addition, there is some pattern instance q_c overlapping p'_e, such that all pattern instances overlapping q_c are eliminated apart from its presuf overlaps; further, q_c or at least a suffix of q_c is matched. This is also a Type 4 phase.

Each comparison in Steps 1 and 2 with a text character to the left of $|q_c$ for which function f'' is defined is charged to the text character specified by the function f'', called its f'' value; f'' values are distinct by definition. Comparisons in Step 3 fall into one of three categories:

1. comparisons which eliminate pattern instances whose left ends lie to the right of $|p''_{k''+1}$ and to the left of $|q_c$;
2. comparisons which eliminate pattern instances whose left ends lie to the right of $|q_c$;
3. the comparison which eliminates p'_e.

Each comparison in the first category is charged to the text character aligned with the left end of the pattern instance eliminated. By the definition of the function f'', these text characters do not occur in the range of f'' values. Comparisons in the second category, along with the comparisons made in Steps 4 and 5 and those successful comparisons in Steps 1 and 2 that involve text characters overlapping q_c, are charged to the text characters compared. BV ensures that each of these comparisons involves a distinct text character. Thus each text character which lies to the right of $|p''_{k''+1}$ and is aligned with or to the left of $q_c|$ is charged at most once. The comparison that eliminates p'_e is charged to the text character aligned with $|p''_{k''+1}$. Since all f'' values lie to the right of $|p''_{k''+1}$ and all pattern instances eliminated by comparisons in the first category have left ends to the right of $|p''_{k''+1}$, this text character is charged exactly once. The two comparisons in Step 2 lacking f'' values constitute the overhead of this presuf shift. Since p'_e is eliminated, the next presuf shift occurs at least distance $m+1$ to the right of the current presuf shift.

3. p'_e is eliminated as are all pattern instances overlapping p'_e. This is a Type 3 phase.

Let q_d denote the leftmost surviving pattern instance. All comparisons in Steps 1 and 2 for which function f'' is defined are charged to their f'' values. f'' values are distinct by definition. Excluding the comparison which eliminates p'_e, each comparison in Steps 3 and 4 eliminates some pattern instance whose left end lies to the right of $|p''_{k''+1}$ and to the left of $|q_d$. Each such comparison is charged to the text character aligned with the left end of the pattern instance eliminated. These text characters cannot occur in the range of the function f'' and hence are charged only once. Thus each text character which lies to the right of $|p''_{k''+1}$ and to the left of $|q_d$ is charged at most once. The comparison that eliminates p'_e is charged to the text character aligned with $|p''_{k''+1}$. The two comparisons in Step 2 lacking f'' values constitute the overhead of this presuf shift. Since p'_e is eliminated, the next presuf shift occurs at least distance $m+1$ to the right of the current presuf shift. □

6.3. The transfer function f''. In this section, we prove Lemma 6.6. The definition of the function f'' is similar to that of the function f in section 5. This is hardly surprising since the elimination procedure ET'' is similar to the elimination process ET, the only difference between the two being that the former switches to a left-to-right comparison sequence in some cases.

First, note that each of the definitions and lemmas in section 5.1 continue to hold if p''_j, x''_j, V'', p'', A'', g'', k'', and m'' replace p_j, x_j, V, p, A, g, k, and m, respectively, for $1 \leq j \leq k'' + 1$.

Since patterns with $g'' = 1$ and $|x''_k| = 1$ are special-case patterns, we assume that $g'' > 1$ if $|x''_k| = 1$. If $p''_1[m'']$ does not match the text, then Steps 1 and 2 of the presuf shift handler make at most one comparison. Therefore, we also assume that $p''_1[m]$ matches the text. Let p''_l be the rightmost element in A''_1. Let p''_r be the rightmost element in V'' outside $A''_{g''}$, if such a pattern instance exists.

As in section 5.2, we split the sequence C' of comparisons made in Steps 1 and 2 of the presuf shift handler into three classes as follows.

1. Class 1 consists of the comparison in Step 1. In addition, if $|x''_k| = 1$, then Class 1 contains the comparisons which comprise the smallest prefix of C' having the following property: either the last comparison in that prefix is unsuccessful or following that comparison, exactly one pattern instance in $A''_{g''}$ survives.

2. Class 2 consists of the comparisons in C' which follow all Class 1 comparisons and are made in the suffix $h(x''_l)$ of p''_1.

3. Class 3 consists of comparisons in C' which follow all Class 2 comparisons.

f'' values are defined by considering 3 cases.

Case 1. Suppose Phase 1 of Step 2 terminates with a mismatch or $p''_\alpha = p''_1$. Then ET'' eliminates among elements in V'' exactly as ET eliminates among the elements of V. Therefore, f'' values for comparisons are defined exactly as in section 5.2 with p''_j, x''_j, V'', p'', A'', g'', k', and m'' replacing p_j, x_j, V, p, A, g, k, and m, respectively, for $1 \leq j \leq k'' + 1$.

Case 2. Suppose $p''_\alpha = p''_2$ or the half-done set left uneliminated by Phase 1 has at most two elements. The only difference between the way ET'' eliminates among the elements in V'' and ET eliminates among elements in V is in the last comparison of Step 2. Note that in section 5.2, the last comparison in Step 2 is not given an f value. Therefore, f'' values for comparisons in this case are again defined exactly as in section 5.2 with p''_j, x''_j, V'', p'', A'', g'', k'', and m'' replacing p_j, x_j, V, p, A, g, k, and m, respectively, for $1 \leq j \leq k'' + 1$.

Case 3. Suppose all comparisons in Phase 1 are successful, $p''_\alpha \neq p''_1, p''_2$, and the half-done set which survives Phase 1 has at least three elements. The only difference between the way ET'' eliminates among the elements in V'' and ET eliminates among the elements in V is in the portion of Phase 2 that makes comparisons according to a left-to-right sequence. As we will show in Lemma 6.11, this left-to-right sequence involves only text characters to the left of the suffix x''_1 of p''_1. Consequently, Class 1 and Class 2 comparisons are not affected by this sequence.

f'' values for comparisons in Class 1 are defined exactly as in section 5.2 with p''_j, x''_j, V'', p'', A'', g'', k'', and m'' replacing p_j, x_j, V, p, A, g, k, and m, respectively, for $1 \leq j \leq k'' + 1$. Consider Class 2 comparisons next. At most one element of V'' will survive a mismatch in Class 2, if any, because Phase 1 has no mismatches. Therefore, if a mismatch occurs in Class 2, then Class 3 is empty and f'' values for Class 2 comparisons are defined exactly as in section 5.2 with the appropriate substitutions mentioned above. Otherwise, if all comparisons in Class 2 are successful, then f'' values for all but some s, $s \leq 2$, of the comparisons in Class 2 are defined in the same manner. It remains to define f'' values for Class 3 comparisons and s Class 2 comparisons when all comparisons in Class 2 are successful. This involves modifying only Case 2 of the definition of f values for Class 2 and Class 3 comparisons in section 5.2. We define f'' values for all but two of these comparisons. The range of these f''

values is the same as the range of the f values defined for this subcase, i.e., to the left of the suffix x''_l of p''_1 and to the right of $|p''_{k+1}$.

Following Class 1 and 2 comparisons, at most $\min\{r_l, 2\} - s$ of the elements of V'' to the right of p''_l survive along with the elements in A''_1. Let O' refer to the set of $\min\{r_l, 2\}$ elements in V'' which includes elements which survive comparisons in $h(x''_l)$ and elements which are eliminated by one of the s Class 2 comparisons under consideration. Let O refer to the largest half-done set consisting of elements in A''_1 and O'. Redefine O' by removing pattern instances in it which are also in O. Considering comparisons which eliminate pattern instances in O and O' is equivalent to considering Class 3 comparisons plus s of the Class 2 comparisons. Let $O = \{p''_{h_1}, \ldots, p''_{h_e}\}$. Let v and u be the core and head, respectively, of x''_{h_1} and let $v = u'u$. $|v| > 1$ because either $|x''_k| > 1$ or $|x''_k| = 1$ and $g' > 1$. By Lemma 5.10, v contains a misfit character. If $l = 1$, then the number of comparisons in Class 3 plus s is at most $2 - s + s = 2$. In this case, we do not define an f'' value for the comparisons in Class 3 and the s comparisons in Class 2. Therefore, suppose that $l > 1$.

The comparisons given by tree ET'' in this case form two sequences; the first sequence which includes Phase 1 and part of Phase 2 is a right-to-left sequence and the second sequence is a left-to-right sequence. The following lemmas show some properties which are necessary for defining f''.

LEMMA 6.8. *The portion of p''_α which overlaps the suffix x''_i, $1 \le i \le \alpha$, of p''_1 matches x''_i.*

Proof. x''_i is a suffix of v_p. The length $|v_p|$ substring of p''_α which is to the immediate right of t_a is identical to v_p. □

LEMMA 6.9. *$p''_\alpha \in O$ and $|O| \ge 3$.*

Proof. Since all comparisons in Phase 1 are successful, p''_1 survives Phase 1. By Lemma 6.3, p''_α also survives. If $p''_\alpha \notin O$, then p''_1 and p''_α do not form a half-done set with any other element in V''. Therefore, the cardinality of the half-done set which survives Phase 1 would be at most 2, which is a contradiction. Thus $p''_\alpha \in O$. p''_2 must form a half-done set along with p''_1 and p''_α; otherwise, no other element in V'' forms a half-done set with p''_1 and p''_α and, consequently, at most two elements in V'' would survive the successful Phase 1 comparisons. By Lemma 6.8, p''_1 and p''_α survive successful comparisons in $h(x''_l)$, and then by Lemma 5.3, p''_2 also survives these comparisons. Therefore, $p''_2 \in O$ also. Since $p''_\alpha \ne p''_1, p''_2$, the lemma follows. □

COROLLARY 6.10. *The half-done set which survives Phase* 1 *must be a subset of O.*

Proof. Both p''_1 and p''_α survive successful comparisons in Phase 1 and both are elements of O. The only elements in V'' which can form a half-done set with p''_1 and p''_α are those in O. □

LEMMA 6.11. *The leftmost character compared by ET'' in the first (right-to-left) sequence is at least distance $|v|$ to the right of the rightmost character compared in the second (left-to-right) sequence. The rightmost character compared in the latter sequence is to the left of the suffix x''_1 of p''_1.*

Proof. Let d'' be the rightmost position in p''_α such that $p''_\alpha[d'']$ is aligned with some character in p''_1 and $p''_\alpha[d''] \ne p''_\alpha[d'' + |v|]$. Such an index exists by Lemma 4.7. All comparisons in the second sequence are aligned with or to the left of $p''_\alpha[d'']$. All characters in the first sequence compared in Phase 2 are aligned with or to the right of $p''_\alpha[d'' + |v|]$. All characters compared in Phase 1 involve characters in the suffix x''_1 of p''_1. By Lemma 6.8, $p''_\alpha[d'']$ is to the left of the suffix x''_1 of p''_1. The lemma

follows. □

COROLLARY 6.12. *Successful comparisons which eliminate elements of O are made at least distance $|v|$ apart.*

LEMMA 6.13. *The portion of p''_{h_e} that overlaps the suffix $x''_{h_{e-2}}$ of p''_1 matches that suffix.*

Proof. Since $p''_\alpha \in O$ and $p''_\alpha \neq p''_1, p''_2$, it follows from Lemma 6.8 that $(uu')^2$ is a suffix of $p''[1 \ldots m'' - |x''_1|]$. Therefore, the portion of p''_{h_e} that overlaps the suffix $x''_{h_{e-2}}$ of p''_1 matches that suffix. □

COROLLARY 6.14. *All successful comparisons which eliminate an element of O are made to the left of the suffix $x''_{h_{e-2}}$ of p''_1.*

We now define the f'' function for this case.

First, consider comparisons which eliminate elements of O'. From Corollary 6.10, it follows that all elements of O' must be eliminated by Phase 1 comparisons. These comparisons have to be successful because all comparisons in Phase 1 are successful. If $|O'| = 2$, then, by Lemma 4.5, the first such comparison is made in the suffix x''_{h_e} of p''_1. If $|O'| = 2$ or $|O'| = 1$, then, by Lemma 6.13, the portion of p''_{h_e} which overlaps the suffix $x''_{h_{e-1}}$ of p''_1 matches that suffix and therefore, by Lemma 4.5, the last comparison which eliminates an element of O' is made in the suffix $x''_{h_{e-1}}$ of p''_1. Consider the text characters t_c and t'_c with which $|p''$ is aligned when the rightmost misfit characters in the prefixes $x''_{h_{e-1}}$ and $x''_{h_{e-2}}$, respectively, of p'' are aligned with t_b. Since v is a suffix of $x''_{h_{e-1}}$ and $x''_{h_{e-2}}$ and since v contains a misfit character, t_c is aligned with the suffix $x''_{h_{e-1}}$ of p''_1 and to the left of the suffix x''_{h_e} of p''_1 while t'_c is aligned with the suffix $x''_{h_{e-2}}$ of p''_1 and to the left of the suffix $x''_{h_{e-1}}$ of p''_1. If $|O'| = 2$, then f'' is defined to map the text characters involved in comparisons which eliminate elements of O' to the text characters t_c and t'_c. If $|O'| = 1$, then f'' is defined to map the text character involved in the comparison which eliminates the only element of O' to the text character t'_c. A simple case analysis ($p''_l = p''_{h_e}, p''_{h_{e-1}}, p''_{h_{e-2}}$) shows that these f'' values are to the left of p''_l, as claimed. The two f'' values are clearly distinct and to the left of their respective text characters. Further, they are aligned with the suffix x''_1 of p''_1. Since $|x''_1| < \frac{m''}{2}$, these f'' values are to the right of $|p''_{k''+1}$.

Next, consider comparisons which eliminate elements of O, excluding the leftmost and the last such comparison. The remaining comparisons must be successful. f is defined to map the text character t_c involved in such a comparison to the text character with which $|p''$ is aligned when the leftmost character in p'' which differs from t_c is aligned with t_c. Clearly, $f''(t_c)$ is aligned with or to the left of t_c. Since uu' contains at least two characters, $f''(t_c)$ is at most distance $|v| - 1$ to the left of t_c. It follows from Corollary 6.12 that $f''(t_c)$ is distinct from the f'' values for all other text characters involved in successful comparisons which eliminate elements of O. By Corollary 6.14, these f'' values are to the left of f'' values for successful comparisons which eliminate elements of O' and therefore to the left of p''_l. Only the leftmost text character involved in a comparison which eliminates an element of O is within distance $|v|$ of t_a; the rest are at least distance $|v| + 1$ to the right of t_a. Therefore, these f'' values are to the right of $|p''_{k''+1}$.

This concludes the definition of the transfer function f''.

6.4. Data-structure details. It remains to describe the maintenance of the sets Q_x for each terminal node x of tree ET''. These sets can be maintained exactly as described in section 4.5 but with the following difference: the sequence of comparisons corresponding to Phase 2 in Step 2 of the elimination strategy using ET'' is a left-to-

right sequence if all comparisons in Phase 1 are successful.

As in section 4.5, let $l_1, \ldots, l_h$ be the nodes, in order of appearance, on the leftmost path from the root of ET''. Consider the largest i such that $tc_{l_i}, \ldots, tc_{l_{h-1}}$ (recall from section 4.5 that tc_x is the text character compared at node x of ET'') is a left-to-right sequence. For all terminal nodes in ET'' which are not in the subtree rooted at l_i, the data structure is maintained exactly as in section 4.5. Q_{l_h} can be stored explicitly. It remains to describe the data structure for terminal nodes in the right subtrees of $l_i, \ldots, l_{h-1}$.

Note that if a mismatch occurs at tc_{l_j}, $i \leq j \leq h-1$, at most two elements in V'' survive. Therefore, for each terminal node x in the right subtree of l_j, either $p(x)$ or $p(p(x))$ equals l_j, where $p(x)$ is the parent of x. From the definition of the sets Q_x in section 4.5, it follows that for terminal nodes x and y in the right subtree of l_j, $Q_x = Q_y$. The following lemma is crucial.

LEMMA 6.15. *Let terminal node x_1 be in the right subtree of l_{j_1} and terminal node x_2 be in the right subtree of l_{j_2}, $i \leq j_1, j_2 \leq h-1$, $j_2 > j_1$. If $q \in Q_{x_1}$ and $q \in Q_{x_2}$, then q occurs at all terminal nodes in the right subtrees of $l_i, \ldots, l_{j_1}$.*

Proof. Clearly, q cannot overlap $tc_{l_{j_1}}$. Since $tc_{l_i}, \ldots, tc_{l_{h-1}}$ form a left-to-right sequence, q cannot overlap $tc_i, \ldots, tc_{j_1}$. Further, since q occurs at some terminal node in the subtree rooted at l_i, characters in q which overlap $tc_{l_1}, \ldots, tc_{l_{i-1}}$ match the characters $c_1, \ldots, c_{h-1}$, respectively. The lemma follows from the definition of the sets Q_x. □

COROLLARY 6.16. *Suppose q occurs at some terminal node in the subtree T rooted at l_i. Further, suppose j is the largest number, if any, such that $i \leq j \leq h-1$ and q does not overlap tc_{l_j}. Then q occurs at all terminal nodes in the right subtrees of $l_i, \ldots, l_j$.*

Corollary 6.16 immediately gives a linear-space scheme for storing the sets Q_x for terminal nodes x in the subtree T rooted at l_i. Two sets Com_j and $Spec_j$ are maintained at each node l_j, $i \leq j \leq h-1$. A pattern instance q is added to Com_j if it occurs at some terminal node in T and overlaps $tc_{l_{j+1}}$ but not tc_{l_j}. A pattern instance q is added to $Spec_j$ if it overlaps tc_{l_j} and occurs at a terminal node in the right subtree of l_j. Each q can be added to at most one Com set and one $Spec$ set; thus, the total space used is linear. Q_x is readily seen to equal $Com_j \cup Com_{j+1} \cup \cdots \cup Com_{h-1} \cup Spec_j$. Note that each pair of Com sets is disjoint and Com_k is disjoint from $Spec_j$, for each $j \leq k \leq h-1$. In order to obtain Q_x as a sorted list, it suffices to maintain each of the Com and $Spec$ sets as ordered lists which are then appended together. Thus obtaining any particular Q_x takes $O(m)$ time. Q_{l_h} is stored explicitly and hence can be obtained as a list in constant time.

6.5. Presuf shift handler for special-case patterns. We describe the presuf shift handler for patterns for which $|x''_k| = 1$ and $g'' = 1$. This presuf shift handler leads to an overhead of at most two per presuf shift. We show that if a presuf shift has overhead two, then the next presuf shift must occur distance at least $\frac{3(m+1)}{4}$ to the right, and if a presuf shift has overhead one, then the next presuf shift must occur distance at least $\frac{m+1}{2}$ to the right. A comparison complexity of $n(1+\frac{8}{3(m+1)})$ follows.

Let $b = x''_k$. p contains at least two different characters. Therefore, v_p and p'' both contain at least two different characters. Let $p''[j]$ and $p''[j']$ be, respectively, the leftmost and rightmost characters in p'' which differ from b. Let t_c be the text character to the immediate right of t_a.

We consider two cases, namely $|x''_1| < \frac{|v_p|}{2}$ and $|x''_1| \geq \frac{|v_p|}{2}$. The former case has the advantage that if all presuf pattern instances of the first type (recall that presuf

pattern instances were classified into two types in section 6) are eliminated, then the next presuf shift occurs distance at least $\frac{3(m+1)}{4}$ to the right. The absence of this property in the latter case makes it more complicated.

Case 1. $|x''_1| < \frac{|v_p|}{2}$.

Step 1. Step 1 locates the leftmost non-b text character t_d to the right of t_a. Following Step 1, either the basic algorithm is resumed or p'_e, the leftmost surviving pattern instance, is determined and Step 2 follows. This is done as follows. Text characters to the right of t_a and to the left of $p''_\alpha[j]$ are compared from left to right with the character b. A mismatch in this process terminates Step 1. If no mismatch occurs, then $p''_\alpha[j]$ is compared with the aligned text character. A match terminates Step 1. In case of a mismatch, text characters aligned with or to the right of $p''_\alpha[j]$ are compared from left to right with the character b. Step 1 then terminates when a mismatch occurs or when the right end of the text is reached.

One of the following situations now holds:

1. t_d is to the left of $p''_1[j]$. $p''_1, \ldots, p''_{k+1}$ are eliminated and the basic algorithm is resumed with $|p$ placed to the right of the text character that mismatched.

2. t_d is aligned with $p''_i[j]$, $i \neq \alpha$. p'_e is the pattern instance whose left end is aligned with $|p''_i$.

3. t_d is aligned with $p''_\alpha[j]$ and $t_d = p''_\alpha[j]$. p'_e is defined to be the leftmost presuf pattern instance.

4. t_d is aligned with $p''_\alpha[j]$ but $t_d \neq p''_\alpha[j]$. The basic algorithm is resumed with $|p$ immediately to the right of t_e.

5. t_d exists but does not satisfy any of the above cases. p'_e is the pattern instance such that $p'_e[j]$ is aligned with t_d.

6. t_d does not exist. There are no further occurrences of the pattern in the text and the algorithm terminates.

Steps 2 *and* 3. Let q_c denote p'_e. Then Steps 2 and 3 are identical to the corresponding steps in the presuf shift handler for special-case patterns described in section 4.4.

Note that at most two mismatches are made in Step 1 and the first mismatch eliminates p''_α.

LEMMA 6.17. *If p is a special-case pattern and $|x''_1| < \frac{|v_p|}{2}$, then the comparison complexity of the algorithm is $n(1 + \frac{8}{3(m+1)})$.*

Proof. We give charging strategies to show that a presuf shift can have overhead at most two. Further, we show that an overhead of one forces the next presuf shift to occur at least distance $\frac{m+1}{2}$ to the right and an overhead of two forces the next presuf shift to occur at least distance $\frac{3(m+1)}{4}$ to the right. The lemma follows.

As in Lemma 4.12, the run of the algorithm is divided into phases; a phase can be of one of four types. The range of text characters charged in each type of phase remains exactly the same as in Lemma 4.12. The charging scheme for Type 1 and Type 2 phases also remains exactly the same. Only the charging scheme for Type 3 and Type 4 phases is modified in accordance with the presuf shift handlers described above.

We consider a single phase, which could be a Type 3 or a Type 4 phase. We assume that this phase begins with a presuf shift with $|x'_1| \geq \frac{m}{2}$.

The charging scheme. Let q_c be the leftmost pattern instance which survives Step 1. Note that q_c is the leftmost presuf pattern instance if and only if no mismatches occur in Step 1. All successful comparisons in Step 1 are charged to the text characters compared. These text characters lie to the left of $q_c[j]$ if q_c is not the leftmost presuf

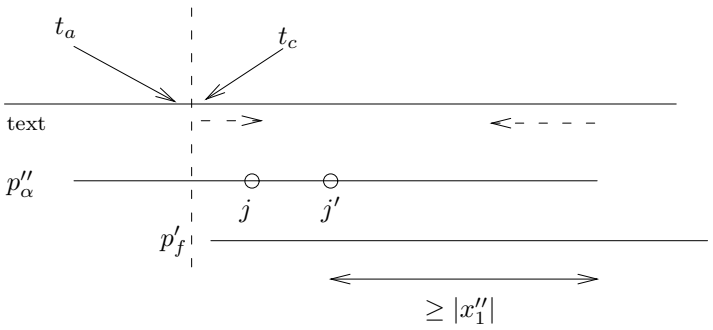


FIG. 7. *Step* 1 *of Case* 2.

pattern instance and are aligned with or to the left of $q_c[j]$ otherwise. If unsuccessful comparisons occur in Step 1, then these comparisons constitute the overhead of this shift. Otherwise, if all comparisons in Step 1 are successful, the only possible comparison which constitutes the overhead of this shift is the comparison in Step 2 which eliminates q_c. Thus the overhead is at most two. Since the first mismatch in Steps 1 and 2 eliminates all presuf pattern instances of the first type and since $|x''_1| < \frac{|v_p|}{2}$, either the overhead is zero or the next presuf shift occurs distance at least $\frac{3(m+1)}{4}$ to the right.

Now consider two cases.

1. Suppose q_c survives Step 2. All comparisons made in Steps 2 and 3 are charged to the text characters compared. Thus each text character which lies to the right of t_a and is aligned with or to the left of $q_c|$ is charged at most once over Steps 1, 2, and 3. All future comparisons will be charged to text characters to the right of $q_c|$.

2. Suppose q_c does not survive Step 2. Each successful comparison in Step 2 eliminates some pattern instance lying entirely to the right of $q_c[j]$ and is charged to the text character aligned with the left end of that pattern instance. The unsuccessful comparison which eliminates q_c in Step 2 is charged to the text character aligned with $q_c[j]$ if q_c is not the leftmost presuf pattern instance. Thus each text character lying between t_a and $|q_d$ is charged at most once, where q_d is the leftmost surviving pattern instance at the end of Step 2. All future comparisons will be charged to text characters aligned with or to the right of $|q_d$. □

Case 2. $|x''_1| \geq \frac{|v_p|}{2}$.

There are five steps in the presuf shift handler for this case. At most five mismatches are made in these steps. We show that three of these mismatches can be charged to unmatched text characters; consequently, the overhead of the current presuf shift is at most two. Further, the first mismatch in Step 1 eliminates p''_α and the second mismatch eliminates all of the presuf pattern instances.

Step 1. Step 1 eliminates all but at most one of $p''_1, \ldots, p''_{k+1}$ as follows. See Fig. 7. The following sequence of text characters is compared with the aligned characters in p''_α: t_b, followed by the text characters strictly between t_b and $p''_\alpha[j']$ considered right to left, followed by the text characters strictly between t_a and $p''_\alpha[j]$ considered left to right. Step 1 terminates when the first mismatch occurs or when this sequence is exhausted.

Let p'_e be the leftmost surviving presuf pattern instance following Step 1. Consider the pattern instance p'_f, $|p'_f$ aligned with the text character to the immediate right of

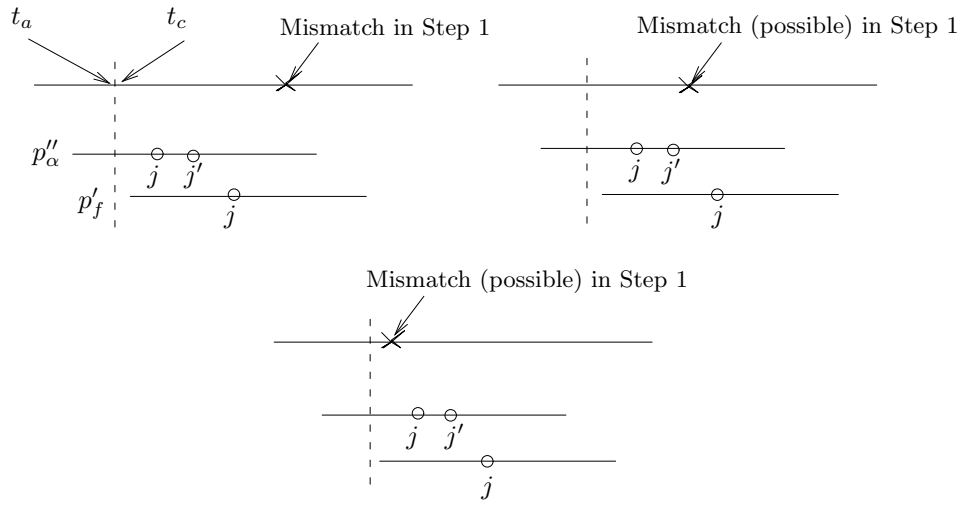


FIG. 8. *Possible outcomes of Step* 1.

t_c. Let t_e be the text character at which the mismatch occurred, if any. Note that since $j \geq |x''_1| + 1$ and $|x''_1| \geq \frac{|v_p|}{2}$, by Lemma 6.8, $p'_f[j]$ must be to the right of $p''_\alpha[j']$. The outcome of Step 1 depends upon which of the following two cases occurs (see Fig. 8).

Case 1.1. $p'_f[j]$ is aligned with or to the left of t_e (first diagram in Fig. 8). Clearly, $p''_\alpha[j']$ is to the left of t_e. We show that a transfer function similar to the function f'' of section 6.1 (see Lemma 6.6) can be used to account for the comparisons made in Step 1. In this case, the rest of the steps are identical to Steps 3, 4, and 5 of the presuf shift handler of section 6.1.

Case 1.2. Either there is no mismatch in Step 1 or $p'_f[j]$ is to the right of t_e (second and third diagrams in Fig. 8). The leftmost surviving pattern instance with left end to the right of t_c has its jth character to the right of t_b; we show this claim in the next paragraph. Step 2 follows in this case.

Recall that $p'_f[j]$ is to the right of $p''_\alpha[j']$. The mismatch, if any, in Step 1 occurs to the left of $p'_f[j]$. Therefore, all text characters aligned with or to the right of $p'_f[j]$ and to the left of (and including) t_b are identical to b. The claim follows.

Step 2. If p'_e does not extend to the right of t_b, then no comparisons are made in this step (this happens if and only if p'_e is the leftmost presuf pattern instance). Otherwise, Step 2 attempts to extend the match of p'_e. Characters in p'_e to the right of t_b (if any) are compared from left to right until a mismatch occurs or a non-b character is matched against the text. To see that p'_e will have a non-b character to the right of t_b if it extends to the right of t_b, note that the distance between t_b and t_c equals $|v_p| - 1$ and that p'_e has at least two non-b characters distance $|v_p|$ apart, neither of which can be to the left of t_c.

The successful comparisons in this step will be charged to the text characters compared. Clearly, all of these text characters are to the right of the text characters compared in Step 1.

Step 3. A pattern instance p'_g with the following properties is determined in this step.

1. p'_g is the leftmost surviving pattern instance.

2. All surviving pattern instances which overlap $p'_g[i]$ are presuf overlaps of p'_g, where i is defined as follows. If $p'_g \neq p'_e$, $i = j$. If $p'_g = p'_e$ and p'_e is the leftmost presuf pattern instance, then $p'_g[i]$ is the character aligned with t_b. Otherwise, if $p'_g = p'_e$ and p'_e is not the leftmost presuf pattern instance, then $p'_g[i]$ is the leftmost non-b character in p'_g which is to the right of t_b.

All text characters compared successfully in this step will be distinct from all text characters compared successfully in Steps 1 and 2. There are two cases depending upon the outcome of Step 2.

Case 2.1. p'_e is eliminated in Step 2. At most two mismatches could have occurred in Steps 1 and 2. Note that p'_e cannot be the leftmost presuf pattern instance in this case. The leftmost surviving pattern instance must have its left end to the right of t_c. As shown in Step 1, its jth character must be to the right of t_b. There are two subcases.

Case 2.1a. Step 2 terminates in a mismatch at a non-b character t_h in p'_e. Then, starting at t_h, a left-to-right pass is made in which each text character is compared with b. This pass ends when a mismatch occurs or when the right end of the text is reached. In the latter case, there are no further occurrences of the pattern and the algorithm terminates. In the former case, p'_g is defined to be the pattern instance in which $p'_g[j]$ is aligned with the text character t_x at which the mismatch occurs. Since all text characters strictly between t_b and t_x are identical to b, p'_g is the leftmost surviving pattern instance and all pattern instances to the right of p'_g which overlap $p'_g[j]$ are eliminated. Note that the number of mismatches made in Steps 1–3 is at most three in this case.

Case 2.1b. Step 2 terminates in a mismatch at a character t_h in p'_e which is a b. p'_g is defined to be the pattern instance in which $p'_g[j]$ is aligned with the text character at which the mismatch occurs. As in the previous case, p'_g is the leftmost surviving pattern instance and all pattern instances to the right of p'_g which overlap $p'_g[j]$ are eliminated. The number of mismatches made in Steps 1–3 is at most two in this case.

Case 2.2. p'_e survives Step 2. At most one mismatch could have occurred so far. There are two subcases.

Case 2.2a. p'_e is not the leftmost presuf pattern instance; i.e., it extends to the right of t_b. Let t_x be the rightmost text character matched in Step 2. t_x must be a non-b character. Consider the pattern instance p'_h, where $p'_h[j]$ is aligned with t_x.

Clearly, all pattern instances to the right of p'_h which overlap t_x are eliminated since each has a b aligned with t_x. We claim that all pattern instances strictly between p'_e and p'_h have also been eliminated. This is shown as follows. All pattern instances to the right of p'_e which overlap t_c have been eliminated in Step 1. Recall from Step 1 that the leftmost surviving pattern instance after Step 1 with left end to the right of t_c has its jth character to the right of t_b. Since all text characters to the right of t_b and up to but not including t_x are identical to b, the claim follows.

If p'_e and p'_h lack a difference point or if $p'_h[j]$ does not match t_x, then $p'_g = p'_e$. Otherwise, if p'_e and p'_h have a difference point, the character in p'_e at that difference point is compared with the aligned text character and one of p'_e and p'_h is eliminated; the difference point itself is to the right of $p'_h[j]$. Let p'_g denote the survivor. Clearly, p'_g is the leftmost surviving pattern instance in both cases. Further, all pattern instances to the right of p'_g which overlap t_x (note that t_x is aligned with $p_g[i]$) have either been eliminated or are presuf overlaps of p'_g.

At most two mismatches are made in Steps 1–3 in Case 2.2a.

Case 2.2b. Second, suppose p'_e is the leftmost presuf pattern instance. Recall that $p'_e[m]$ is aligned with t_b. In this case, no comparisons are made in Step 2. Consider the pattern instance p'_h, where $p'_h[j]$ is to the immediate right of t_b. Recall from Step 1 that p'_h is the leftmost surviving pattern instance with left end to the right of t_c. The only surviving pattern instances to the right of p'_e which overlap t_c are presuf overlaps of p'_e. Therefore, p'_h is the leftmost surviving pattern instance, barring p'_e and its presuf overlaps. In addition, note that any pattern instance which overlaps p'_e but not $p''_\alpha[j']$ and has its jth character to the right of t_b is a presuf overlap of p'_e.

If $|p'_h$ is to the right of $p''_\alpha[j']$, then p'_h is a presuf overlap of p'_e as are all pattern instances to the right of p'_h which overlap p'_e. Step 5 follows with $p'_g = p'_e$ in this case.

Otherwise, if p'_h overlaps $p''_\alpha[j']$ then p'_h is not a presuf overlap of p'_e. The character $p''_\alpha[j']$ is then compared with the text. A match eliminates all pattern instances which overlap $p''_\alpha[j']$ but are not presuf overlaps of p'_e. (This can be seen from the following two facts: (a) all pattern instances with left end to the right of t_c which survive Step 1 have their jth character to the right of $p''_\alpha[j']$, and (b) all surviving pattern instances which overlap t_c are presuf overlaps of p'_e.) Clearly, all surviving pattern instances which overlap t_b are presuf overlaps of p'_e. In this case, Step 5 follows with $p'_g = p'_e$. Otherwise, if a mismatch occurs at $p''_\alpha[j']$, p'_e is eliminated as are all its presuf overlaps which overlap t_c. Text characters to the right of t_b are now compared from left to right with the character b until either a mismatch occurs or the right end of the text is reached. In the former case, Step 4 follows with p'_g denoting the pattern instance such that $p'_g[j]$ is aligned with the text character at which the mismatch occurs. Clearly, p'_g is the leftmost surviving pattern instance and all pattern instances which overlap $p'_g[j]$ have been eliminated. In the latter case (i.e., the right end of the text is reached), the algorithm terminates as there are no further occurrences of the pattern.

At most two mismatches are made in Steps 1–3 in Case 2.2b.

Step 4. In this step, either all surviving pattern instances which overlap p'_g are eliminated (except for presuf overlaps) or p'_g is eliminated. In the latter case, the basic algorithm is resumed with the leftmost surviving pattern instance. In the former case, Step 5 follows. All comparisons in this step are to the right of all text characters matched in the previous steps. In addition, the left end of each pattern instance eliminated in this step is also to the right of any text character matched in one of the previous steps.

In Step 4, difference-point comparisons are used. This step has a number of iterations. In each iteration, a different pattern instance overlapping p'_g but strictly to the right of $p'_g[i]$ is considered. If it is a presuf overlap of p'_g, then nothing is done. Otherwise, if it is not a presuf overlap of p'_g, the character in p'_g at the difference point of the two pattern instances is considered. If the text character aligned with this character has not been successfully compared earlier (this is ascertained using a bit vector), the two characters are compared. Step 4 ends when a mismatch occurs or when all pattern instances overlapping p'_g (excluding presuf overlaps) are eliminated.

If no mismatch occurs in Step 4, then all comparisons in this step will be charged to the text characters compared; otherwise, they will be charged to left ends of the pattern instances eliminated. In both cases, the text characters charged are to the right of all text characters matched in previous steps.

Step 5. This step attempts to complete the match of p'_g. Characters in p'_g which have not yet been matched are compared with the aligned text characters from right to left until a mismatch occurs or all of its characters are matched. In either case, another presuf shift follows. All comparisons in this step will be charged to the text

characters compared.

LEMMA 6.18. *If p is a special-case pattern with $|x_1''| \geq \frac{|v_p|}{2}$, then the comparison complexity of the algorithm is $n(1 + \frac{8}{3(m+1)})$.*

Proof. We give charging strategies to show that a presuf shift can have overhead at most two. Further, we show that an overhead of one forces the next presuf shift to occur at least distance $\frac{m+1}{2}$ to the right and an overhead of two forces the next presuf shift to occur at least distance $\frac{3(m+1)}{4}$ to the right. The lemma follows.

As in Lemma 4.12, the run of the algorithm is divided into phases; a phase can be of one of four types. The range of text characters charged in each type of phase remains exactly the same as in Lemma 4.12. The charging scheme for Type 1 and Type 2 phases also remains exactly the same. Only the charging scheme for Type 3 and Type 4 phases is modified in accordance with the presuf shift handler described above.

We consider a single phase, which could be a Type 3 or a Type 4 phase. We assume that this phase begins with a presuf shift with $|x_1'| \geq \frac{m}{2}$.

The charging scheme. We consider two cases.

Case A. Suppose a mismatch occurs in Step 1 at a text character t_e to the right of $p_\alpha''[j']$ and $p_f'[j]$ is aligned with or to the left of t_e (i.e., Case 1.1 in Step 1 holds).

Each successful comparison in Step 1 matches the character b against the text. The charging scheme for this case is identical to the charging scheme in Lemma 6.7 with the function f'' defined as follows. f'' is defined to map each text character compared successfully in Step 1 to the text character which is distance $j-1$ to its left. This definition of f'' is easily verified to satisfy all four required properties of f'' stated in Lemma 6.6. Further, only the last Step 1 comparison can possibly be unsuccessful and might not receive an f'' value. From the above charging scheme, it follows that the overhead of the current presuf shift is at most one.

Case B. Suppose all comparisons in Step 1 are successful or $p_f'[j]$ is to the right of t_e, the character at which the mismatch in Step 1 occurs (i.e., Case 1.2 in Step 1 holds).

Recall from Step 1 that the leftmost surviving pattern instance completely to the right of t_c must have its jth character to the right of t_b. There are three subcases to consider.

Subcase B1. Suppose p_e' (the leftmost of the presuf pattern instances to survive Step 1) survives Steps 2, 3, and 4.

All successful comparisons in Steps 1, 2, 3, and 4 and all comparisons in Step 5 are charged to the text characters compared. All of these comparisons involve distinct text characters, and thus each text character which is to the right of t_a and aligned with p_e' is charged at most once. Further, at most one mismatch is made in Step 1 and no mismatches are made in Steps 2, 3, and 4 (otherwise, p_e' would be eliminated). In addition, a mismatch in Step 1 eliminates p_α'', thus forcing the next presuf shift to occur at least distance $\frac{m+1}{2}$ to the right. Thus the current presuf shift has overhead at most one and an overhead of one forces the next presuf shift to occur distance at least $\frac{m+1}{2}$ to the right.

Subcase B2. Suppose p_e' is eliminated in one of Steps 2, 3, and 4; further, suppose p_e' is the leftmost presuf pattern instance.

In this case, the next presuf shift occurs distance at least $m+1$ to the right. We show an overhead of at most two for this case.

All comparisons in Step 1 are successful and are charged to the text characters compared. No comparisons are made in Step 2. p_e' must be eliminated in Step 3 since

Step 5 follows directly from Step 3 otherwise (see Case 2.2b in Step 3). All successful comparisons in Step 3 are charged to the text characters compared. At most two mismatches are made in Step 3 and these constitute the overhead of this shift. All text characters matched in Steps 1–3 are to the left of $p'_g[j]$. If p'_g survives Step 4, then all comparisons in Step 4 are charged to the text characters compared. If p'_g does not survive Step 4, then the comparison which eliminates p'_g is charged to the text character aligned with $p'_g[j]$ and all other comparisons in Step 4 are charged to the left ends of the respective pattern instances eliminated. (From the definition of p'_g in Step 3, note that the left ends of these pattern instances are to the right of $p'_g[j]$.) Thus all text characters charged in Step 4 are distinct and are aligned with or to the right of $p'_g[j]$. All comparisons in Step 5 are charged to the text characters compared. These text characters are distinct from all text characters matched in the previous steps. Therefore, if p'_g survives Step 4, then each text character to the right of t_a and aligned with or to the left of $p'_g|$ is charged at most once. Otherwise, if p'_g is eliminated in Step 4 and p'_l is the leftmost surviving pattern instance following Step 4, each text character strictly between t_a and $|p'_l$ is charged at most once.

Subcase B3. Suppose p'_e is eliminated in one of Steps 2, 3, and 4 and p'_e is not the leftmost presuf pattern instance.

In this case, the next presuf shift occurs distance at least $m+1$ to the right. We show an overhead of at most two for this case.

All successful comparisons in Steps 1 and 2 and all comparisons in Step 5 are charged to the text characters compared. If p'_e does not survive Step 2 (Case 2.1 of Step 3), then all successful comparisons in Step 3 are charged to the text characters compared. Otherwise, if p'_e survives Step 2 (Case 2.2a of Step 3), there is at most one comparison in Step 3 and it is accounted for later. If p'_g survives Step 4, then all successful comparisons in Step 4 are charged to the text characters compared. In this case, each text character to the right of t_a and aligned with or to the left of $p'_g|$ is charged at most once. Otherwise, if p'_g is eliminated in Step 4, each successful comparison in Step 4 (except the one which eliminates p'_g) is charged to the text character aligned with the left end of the pattern instance eliminated by this comparison; this text character is to the right of $p'_g[j]$. In this case, each text character strictly between t_a and $|p'_l$ is charged at most once, where p'_l is the leftmost surviving pattern instance after p'_g is eliminated.

At most four comparisons have not yet been accounted for. These include the mismatch in Step 1, the mismatch in Step 4 (which eliminates p'_g), and either the mismatches in Steps 2 and 3 or the only comparison in Step 3, depending on whether or not p'_e survives Step 2. Note that if mismatches occur in all of Steps 2, 3, and 4, then the text character aligned with $p'_g[j]$ is not charged for any comparison. In addition, we show that the text character aligned with $p''_\alpha[j]$ is also not charged for any comparison. An overhead of two for the current presuf shift follows immediately.

Clearly, $p''_\alpha[j]$ is not compared in Step 1. All comparisons in Steps 2, 3, and 4 are made to the right of t_b and hence to the right of $p''_\alpha[j]$. Further, $p'_g[i]$ (i as defined in Step 3) is aligned with or to the right of t_b. Therefore, the text character aligned with $p''_\alpha[j]$ is not charged for any of the comparisons made in Steps 1–4. Consider Step 5 next. If p'_e survives Steps 2 and 3 and is eliminated in Step 4, then the presuf shift handler terminates after Step 4 and the basic algorithm is resumed. Therefore, suppose that p'_e is eliminated in Step 2 or Step 3. From the definition of p'_g in Step 3, $p'_g \neq p'_e$ and therefore $i = j$. To show that all comparisons in Step 5 are made to the right of $p''_\alpha[j]$, it suffices to show that $|p'_g$ is to the right of $p''_\alpha[j]$.

We show this by considering two cases. First, suppose $p''_\alpha \neq p''_1$. Then, by Lemma 6.8, it follows that the character in p'' which is to the immediate left of its suffix x''_1 is a b. Since x''_1 is the longest presuf of p'', it follows that $j = |x''_1| + 1$. By Lemma 6.8, $p''_\alpha[j']$ and hence $p''_\alpha[j]$ are to the left of the suffix x''_1 of p''_1. Since $|p'_g[j]$ is to the right of t_b, $|p'_g$ is aligned with or to the right of the suffix x''_1 of p''_1. The claim follows for this case.

Next, suppose $p''_\alpha = p''_1$. A mismatch occurs in Step 1 since p'_e is not the leftmost presuf pattern instance. Further, this mismatch occurs at some text character t_e to the right of $p''_\alpha[j]$ since no comparisons are made to the left of $p''_\alpha[j]$ in Step 1 in this case. Each comparison in Step 1 compares a text character with b. Since $p'_g[j]$ must be to the right of t_b, either $|p'_g$ is to the right of t_e or a b in p'_g overlaps t_e. However, p'_g will not survive in the latter case. The claim follows.

The lemma follows. □

Finally, we state the following theorem; the proof is similar to the proof of Theorem 4.17.

THEOREM 6.19. *There is a string-matching algorithm with a comparison complexity of* $n(1+\frac{8}{3(m+1)})$ *comparisons which uses* $O(m)$ *space and takes* $O(n+m)$ *time following preprocessing of the pattern*; *the preprocessing time is* $O(m^2)$.

Acknowledgment. We thank Dany Breslauer for a number of comments and suggestions and in particular for the observation that p_{k+1} could be viewed as a presuf pattern instance. This contributed to an improvement of our upper bound from $n + \frac{3(n-m)}{m+1}$ to $n + \frac{8(n-m)}{3(m+1)}$.

REFERENCES

[AC89] A. APOSTOLICO AND M. CROCHEMORE, *Optimal canonization of all substrings of a string*, Technical Report TR 89-75, Laboratoire Informatique, Théorique, et Programmation, Université Paris 7, Paris, 1989.

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

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

[B94] D. BRESLAUER, *Saving comparisons in the Crochemore–Perrin string matching algorithm*, Theoret. Comput. Sci., 158 (1996), pp. 177–192.

[BCT93] D. BRESLAUER, L. COLUSSI, AND L. TONIOLO, *Tight comparison bounds for the string prefix-matching problem*, Inform. Process. Lett., 47 (1993), pp. 51–57.

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

[Co91] R. COLE, *Tight Bounds on the complexity of the Boyer–Moore algorithm*, SIAM J. Comput., 23 (1994), pp. 1075–1091.

[CHPZ92] R. COLE, R. HARIHARAN, M. PATERSON, AND U. ZWICK, *Tighter lower bounds on the exact complexity of string matching*, SIAM J. Comput., 24 (1995), pp. 30–45.

[CCG92] M. CROCHEMORE, A. CZUMAJ, L. GASINIEC, S. JAROMINEK, T. LECROQ, W. PLANDOWSKI, AND W. RYTTER, *Speeding up two string-matching algorithms*, Algorithmica, 5 (1994), pp. 247–267.

[Col91] L. COLUSSI, *Correctness and efficiency of pattern matching algorithms*, Inform. and Comput., 5 (1991), pp. 225–251.

[CGG90] L. COLUSSI, Z. GALIL, AND R. GIANCARLO, *On the exact complexity of string matching*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 135–143.

[CP89] M. CROCHEMORE AND D. PERRIN, *Two-way pattern matching*, Technical Report, Laboratoire Informatique, Théorique, et Programmation, Université Paris 7, Paris, 1989.

[FW65] N. FINE AND H. WILF, *Uniqueness theorem for periodic functions*, Proc. Amer. Math. Soc., 16 (1965), pp. 109–114.

[GG91] Z. Galil and R. Giancarlo, *On the exact complexity of string matching: Lower bounds*, SIAM J. Comput., 6 (1991), pp. 1008–1020.

[GG92] Z. Galil and R. Giancarlo, *On the exact complexity of string matching: Upper bounds*, SIAM J. Comput., 3 (1993), pp. 407–437.

[GS80] Z. Galil and J. Seiferas, *Saving space in fast string-matching*, SIAM J. Comput., 2 (1980), pp. 417–438.

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

[Ha93] C. Hancart *On Simon's string searching algorithm*, Inform. Process. Lett., 47 (1993), pp. 95–99.

[KMP77] D. E. Knuth, J. Morris, and V. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1973), pp. 323–350.

[Lo82] M. Lothaire, *Combinatorics on Words*, Encyclopaedia of Mathematics and Its Applications 17, Addison–Wesley, Reading, MA, 1982.

[LS62] R. Lyndon and M. Schutzenberger, *The equation $a^m = b^n c^p$ is a free group*, Michigan J. Math., 9 (1962), pp. 289–298.

[Vi85] U. Vishkin, *Optimal parallel pattern matching in strings*, Inform. and Control, 67 (1985), pp. 91–113.

[ZP92] U. Zwick and M. Paterson, *Lower bounds for string matching in the sequential comparison model*, manuscript, 1992.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 857–869, June 1997

014

THE k-STEINER RATIO IN GRAPHS*

AL BORCHERS† AND DING-ZHU DU†

Abstract. A Steiner minimum tree (SMT) is the shortest-length tree in a metric space interconnecting a set of points, called the regular points, possibly using additional vertices. A k-size Steiner minimum tree (kSMT) is one that can be split into components where all regular points are leaves and all components have at most k leaves. The k-Steiner ratio, ρ_k, is the infimum of the ratios SMT/kSMT over all finite sets of regular points in all possible metric spaces, where the distances are given by a complete graph. Previously, only ρ_2 and ρ_3 were known exactly in graphs, and some bounds were known for other values of k. In this paper, we determine ρ_k exactly for all k. From this we prove a better approximation ratio for the Steiner tree problem in graphs.

Key words. Steiner trees, Steiner ratio, approximation algorithms, graph algorithms, graph theory

AMS subject classifications. 68R10, 05C05, 05C85, 90C27

PII. S0097539795281086

1. Introduction.

Given a set of points in a metric space, the *Steiner minimum tree* on the point set is the shortest network interconnecting all points in the set. Those points in the set are called *regular points*, and vertices other than regular points are called *Steiner points*. Computing the Steiner minimum tree in various metric spaces is NP-hard [12, 7, 8, 10, 6].

A tree interconnecting a regular point set is called a *Steiner tree* if every leaf is a regular point. However, a regular point in a Steiner tree may not be a leaf. A Steiner tree is *full* if every regular point is a leaf. When a regular point is not a leaf, the tree can be decomposed into several smaller trees at this point. In this way, every Steiner tree can be decomposed into smaller trees in each of which every regular point is a leaf. These smaller trees are called *full components* of the tree. The *size* of a full component is the number of regular points in the full component. For brevity, we will refer to a full component simply as a component.

A k-size Steiner tree is a Steiner tree with all components of size at most k. The k-size Steiner minimum tree is the shortest one among all k-size Steiner trees. The 2-size Steiner minimum tree is also called the minimum spanning tree. The k-Steiner ratio in a metric space E is defined by

$$\rho_k(E) = \inf_{P \subset E} \frac{L_S(P)}{L_{kS}(P)},$$

where $L_S(P)$ is the length of the Steiner minimum tree for the finite set of points P and $L_{kS}(P)$ is the length of the k-size Steiner minimum tree for P. The 2-Steiner ratio is simply called the Steiner ratio. It is known that in the rectilinear plane L_1, $\rho_2(L_1) = 2/3$ [11], and in the Euclidean plane L_2, $\rho_2(L_2) = \sqrt{3}/2$ [9, 3].

Every weighted graph can be seen as a metric space with the distance between two vertices equal to the minimum total length of a path connecting them. Then

$$\rho_k = \inf_G \rho_k(G) = \inf_E \rho_k(E).$$

* Received by the editors January 31, 1995; accepted for publication (in revised form) July 19, 1995. This research was supported in part by National Science Foundation grant CCR-9208913.
http://www.siam.org/journals/sicomp/26-3/28108.html

† Department of Computer Science, University of Minnesota, Minneapolis, MN 55455 (borchers @cs.umn.edu, dzd@cs.umn.edu).

That is, the k-Steiner ratio in graphs is the same as the k-Steiner ratio over all metric spaces. It is a well-known fact that $\rho_2 = 1/2$ [9, 14].

The k-Steiner ratio is important because of Steiner-tree-approximation algorithms. It was a long-standing open problem [2] whether there exists a polynomial-time approximation for the Steiner minimum tree in each metric space with a performance ratio smaller than the inverse of the Steiner ratio. (The performance ratio of an approximation algorithm is the largest lower bound for the ratio of lengths between the approximate solution and the Steiner minimum tree for the same set of points.) Zelikovsky [15] made the first breakthrough. He found a polynomial-time 11/6 approximation for the Steiner minimum tree in graphs, which beat the inverse of the Steiner ratio in graphs, $\rho_2^{-1} = 2$. By extending Zelikovsky's idea, Berman and Ramaiyer [1] gave a polynomial-time 92/72 approximation for the Steiner minimum tree in the rectilinear plane, and Du, Zhang, and Feng [4] gave a general solution to the open problem. They showed that in any metric space, there exists a polynomial-time approximation with performance ratio better than the inverse of the Steiner ratio provided that for any set of a fixed number of points, the Steiner minimum tree is polynomial-time computable. A main part of these works was to establish the lower bound for the k-Steiner ratio. A better lower bound will give a better performance ratio for their approximations. Zelikovsky [15] showed that $\rho_3 \geq 3/5$. Du [5] showed that $\rho_3 \leq 3/5$. Therefore, $\rho_3 = 3/5$. Berman and Ramaiyer [1] proved that $\rho_3(L_1) = 4/5$ and for $k \geq 4$, $\rho_k(L_1) \geq (2k-2)/(2k-1)$. Du, Zhang, and Feng [4] showed that $\rho_k \geq \lfloor \log_2 k \rfloor / (1 + \lfloor \log_2 k \rfloor)$.

In this paper, we determine the k-Steiner ratio in graphs exactly. For $k = 2^r + s$, where $0 \leq s < 2^r$, the k-Steiner ratio is

$$\rho_k = \frac{r2^r + s}{(r+1)2^r + s}.$$

This value agrees with the lower bound determined by Du, Zhang, and Feng [4] when k is a power of 2; however, they gave no upper bound for ρ_k. In section 2, we prove that this value is an upper bound on ρ_k, and in section 3, we prove that it is also a lower bound.

Using these values for ρ_k and Berman and Ramaiyer's algorithm, we get a $1.734\ldots$ approximation to the Steiner tree problem in graphs, improving on their $1.746\ldots$ approximation.

When we talk about a Steiner tree in a graph, the edges of the Steiner tree are actually shortest paths between vertices of the graph. We could think of the graph vertices along such a path as degree-2 Steiner points, though normally we do not. The Steiner points are vertices of the graph, but a vertex can be used more than once as a Steiner point in different components of the same Steiner tree.

We call a binary tree where every internal vertex has exactly two children a *regular binary tree*. A *complete binary tree* is a regular binary tree where all leaves have the same depth. A binary tree that is *complete except perhaps at the bottom level* is a regular binary tree where all leaves have depth t or $t+1$ for some t.

2. Upper bound for ρ_k. To prove the upper bound for the k-Steiner ratio, we consider the following particular metric space M_n based on a weighted tree B_n. Let B_n be a complete binary tree with n levels of edges and a final bottom level where each internal vertex has only one child. The edges at level $1 \leq i \leq n$ have length 2^{n-i} and the edges at the bottom level, level $n+1$, have length 1. See Figure 1.

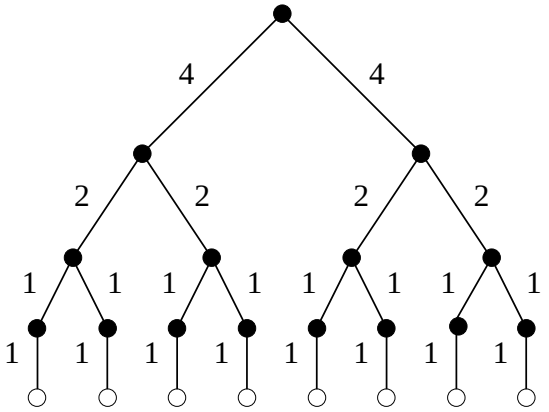


FIG. 1. *The weighted tree B_3 for the metric space M_3.*

The points of M_n are the vertices of B_n; the distance between two points is the length of the shortest path between them in B_n. It is easy to see this forms a metric space. We can think of edges between two points of M_n as paths between those points in B_n.

Let the leaves of B_n be the set P_n of regular points. We will calculate the limit of the ratio $L_S(P_n)/L_{kS}(P_n)$ as $n \to \infty$. This will give us an upper bound for ρ_k.

LEMMA 2.1. *The length of the Steiner minimum tree for P_n is the total length of B_n; that is, $L_S(P_n) = (n+1)2^n$.*

Proof. The tree B_n interconnects P_n, and it is clear that every edge of B_n must be used to interconnect all of P_n. Therefore, this is the shortest tree interconnecting P_n.

In B_n the sum of the lengths of the edges at any fixed level is 2^n—this is true at the bottom two levels, and each level above has half as many edges each with twice the length. Since there are $n+1$ levels, we get $L_S(P_n) = (n+1)2^n$. □

LEMMA 2.2. *There is a k-size Steiner minimum tree for P_n where each Steiner point has degree exactly 3.*

Proof. The regular points of a component in any k-size Steiner minimum tree can be interconnected by a binary tree embedded in B_n, and as in Lemma 2.1, this must be a shortest tree interconnecting these points. Steiner points of degree 2 can be removed and the two adjacent edges replaced by a single edge (which is a path in B_n) between the two adjacent vertices. Therefore, we can assume all Steiner points in this component tree have degree 3. These components make up the desired k-size minimum tree. □

We think of a component in a k-size Steiner minimum tree for P_n as a regular binary tree with at most k leaves embedded in B_n, where the edges are paths in B_n. The root of this component tree is the highest-level degree-2 Steiner point; we will refer to it as the *component root*.

LEMMA 2.3. *There is a k-size Steiner minimum tree for P_n where each Steiner point has degree 3 and where no point of B_n is used more than once as a Steiner point or component root of any component.*

Proof. Consider a k-size Steiner minimum tree T as described in Lemma 2.2. Let v be a vertex at the highest level of B_n that is used as a root or Steiner point of two components, A and B, in T. Let u and w be the children of v. The edges $\overline{vu}$ and $\overline{vw}$

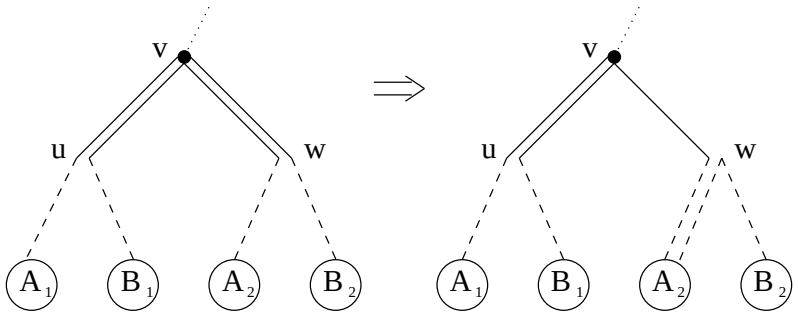


FIG. 2. *Components A and B share a root or Steiner point.*

are used in both components A and B. Let A_1 and B_1 be the parts of components A and B, respectively, that lie below u, and let A_2 and B_2 be the parts that lie below w. See Figure 2.

Components A and B must be connected by some path in T, but because this is a tree, the path cannot go through both B_1 and B_2. Say it does not go through B_2.

Remove the edge $\overline{vw}$ from component B, make B_2 a separate component, and connect B_2 to component A by a path from w to a regular point in A_2 (a leaf in B_n). See Figure 2.

Note that in B_n the length of edge $\overline{vw}$ is the same as the length of the path from w to a leaf—this follows because the length of the edges doubles at each level up the tree. Therefore, this change does not increase the length of the Steiner tree; it does not increase the size of any component; and it does not add any Steiner points, so all Steiner points are still of degree 3. Vertex w has become the root of a new component, so we may have added points that are used as roots or Steiner points of more than one component; however, we have only added them below v.

If we repeat this process, by induction, we can successively remove the Steiner points or component roots that overlap at vertices of B_n from level to level going down the tree until there are no such points remaining. □

LEMMA 2.4. *There is a k-size Steiner minimum tree for P_n where each Steiner point has degree 3 and where every internal vertex of B_n, except at the lowest level, is used exactly once as a Steiner point or component root of some component.*

Proof. We know from Lemma 2.3 that there is a k-size Steiner minimum tree T where every internal vertex of B_n is used at most once. The lowest-level internal vertices cannot be component roots or Steiner points because they have only one child. We will show that $2^n - 1$ Steiner points and roots are needed to interconnect all 2^n points in P_n, and since there are only $2^n - 1$ internal vertices of B_n not at the lowest level, every point must be used exactly once.

Remove the component roots and Steiner points one by one from T. When a component root is removed, split the tree into two pieces at the root. When a Steiner point is removed, split the tree into two pieces by disconnecting one of the two children of the Steiner point. After all points are removed, the 2^n regular points must be completely disconnected. Since removing one point adds only one new piece, we must have removed $2^n - 1$ points to create the 2^n disconnected pieces. □

THEOREM 2.5 (upper bound for ρ_k). *For any k, with $k = 2^r + s$, where $0 \le s <$*

2^r, *we have*

$$\rho_k \le \frac{r2^r + s}{(r+1)2^r + s}. \tag{1}$$

Proof. Let $T_{k,n}$ be a k-size Steiner minimum tree on P_n that satisfies Lemma 2.4.

We can think of a component C in $T_{k,n}$ as a regular binary tree; the internal vertices of this tree are the component root and the Steiner points. Below each of these internal vertices are two edges of B_n; all Steiner points have two edges below them by Lemma 2.2, and the component root must have two edges below it in a minimum tree. Call these edges of B_n the *peak edges* of C, and denote them by $\mathcal{P}_C$. Call the rest of the edges used by C the *connecting edges* of C, and denote them by $\mathcal{C}_C$. Denote all of the peak edges of $T_{k,n}$ by $\mathcal{P}_{k,n}$ and all the connecting edges by $\mathcal{C}_{k,n}$. When we refer to a *peak*, we mean two peak edges and their common vertex.

We want to show that

$$\sum_{e\in\mathcal{C}_C} \text{length}(e) \ge \left(\frac{2^r}{r2^r + s}\right) \sum_{e\in\mathcal{P}_C} \text{length}(e). \tag{2}$$

If this is true, then summing over all components of $T_{k,n}$ gives

$$\sum_{e\in\mathcal{C}_{k,n}} \text{length}(e) \ge \left(\frac{2^r}{r2^r + s}\right) \sum_{e\in\mathcal{P}_{k,n}} \text{length}(e). \tag{3}$$

Assume that we have inequality (3). By Lemma 2.4 the Steiner points and component roots of $T_{k,n}$ cover all of the internal vertices of B_n exactly once, except at the lowest level. Thus the peak edges of $T_{k,n}$ cover all of the edges of B_n exactly once, except for the edges at the lowest level. The sum of the lengths of the edges at the lowest level is 2^n, and the sum of the lengths of all edges of B_n is $L_S(P_n)$ by Lemma 2.1. Thus

$$L_S(P_n) = 2^n + \sum_{e\in\mathcal{P}_{k,n}} \text{length}(e).$$

Then by inequality (3),

$$\begin{aligned} L_{kS}(P_n) &= \sum_{e\in\mathcal{P}_{k,n}} \text{length}(e) + \sum_{e\in\mathcal{C}_{k,n}} \text{length}(e) \\ &\ge \left(1 + \frac{2^r}{r2^r + s}\right) \sum_{e\in\mathcal{P}_{k,n}} \text{length}(e) \\ &= \frac{(r+1)2^r + s}{r2^r + s}(L_S(P_n) - 2^n). \end{aligned}$$

Then

$$\begin{aligned} \rho_k &\le \frac{L_S(P_n)}{L_{kS}(P_n)} \\ &\le \frac{r2^r + s}{(r+1)2^r + s} + \frac{2^n}{L_{kS}(P_n)} \\ &\le \frac{r2^r + s}{(r+1)2^r + s} + \frac{1}{(n+1)} \end{aligned}$$

since ρ_k is the infimum of $L_S(P)/L_{kS}(P)$ and since $L_{kS}(P_n) \geq L_S(P_n) = (n+1)2^n$. Letting $n \to \infty$ gives the upper bound for ρ_k, inequality (1).

To complete the proof of Theorem 2.5, we must establish inequality (2). This inequality depends on $k = 2^r + s$, but the component C can be k-size or smaller. We will prove it for a full k-size component in a k-size tree for all k. It is easy to see that for $k' \leq k$ and $k' = 2^{r'} + s'$, where $0 \leq s' < 2^{r'}$, we have

$$\frac{2^{r'}}{r'2^{r'} + s'} \geq \frac{2^r}{r2^r + s}.$$

From this it follows that inequality (2) is true for any component of size less than k in a k-size Steiner tree.

Note that we always assume that Steiner points have degree 3, as we can by Lemma 2.2, so we can always think of a component as a regular binary tree.

We will prove inequality (2) for k-size components by induction on the sum of the depths of the peaks from the root of the component. We compute the depths of the peaks by counting edges in B_n, not edges in M_n; and we count each edge the same, ignoring the lengths of the edges. The sum of the peak depths is minimum when the peaks form a complete binary tree, except perhaps at the lowest level, so we will treat this case first.

In a k-size component C whose peaks form a complete binary tree, except perhaps at the lowest level, we have 2^r peak edges at the second to lowest level and $2s$ peak edges at the lowest level. Say a peak edge at the lowest level has length w. Then the peak edges at the lowest level have total length $2sw$ and at the r higher levels each level has total length $2^{r+1}w$. Thus we get

$$\sum_{e \in \mathcal{P}_C} \text{length}(e) = 2w(r2^r + s).$$

The connecting edges of C form paths from the lowest-level peak edges down to regular points at the leaves of B_n. By the construction of B_n, these paths have the same length as the lowest-level peak edge where they originate. There are $2s$ such paths of length w at the lowest level and $2^r - s$ such paths of length $2w$ one level higher. Thus

$$\sum_{e \in \mathcal{C}_C} \text{length}(e) = 2w \cdot 2^r.$$

In this case, inequality (2) is satisfied; in fact, it is an equality.

Next, we will show that the ratio of the length of the connecting edges to the length of the peak edges decreases as the sum of the depths of the peaks decreases. In fact, as we change the shape of the component by moving a peak up, the length of the connecting edges remains constant, while the length of the peak edges increases. When the peaks cannot move up any further, the sum of the peak depths is minimum, and as we just showed, inequality (2) holds. Hence it will hold for any shape component. Here are the details.

There are two cases in which the sum of the peak depths might not be minimum: either a peak has a nonpeak edge above it or else there are two bottom-level peak edges whose depths differ by more than 1. We assume that inequality (2) has been established for all possible components where the sum of the peak depths is at most N and that the sum of the peak depths of component C is $N + 1$, so it is not minimum.

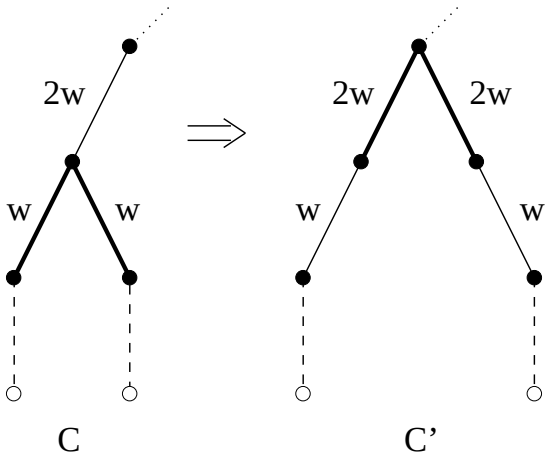


FIG. 3. *C has a peak with a nonpeak edge above it.*

Case 1. C has a peak with a nonpeak edge above it.

Suppose the peak edges with the nonpeak edge above have length w. Consider the component C' obtained by moving the peak with the nonpeak edge above it up one level; see Figure 3. This decreases the sum of the peak depths by 1, so our induction hypothesis applies to C'.

The sum of the lengths of the connecting edges of C and C' are the same—the connecting edge of length $2w$ above the peak in C has been replaced by two connecting edges of length w in C'. The sum of the lengths of the peak edges of C is less than that in C'—the two peak edges of length w in C have been moved up so they each have length $2w$ in C'. Combining this with the induction hypothesis on C', we get

$$\begin{aligned}\sum_{e\in\mathcal{C}_C}\text{length}(e) &= \sum_{e\in\mathcal{C}_{C'}}\text{length}(e)\\ &\geq \left(\frac{2^r}{r2^r+s}\right)\sum_{e\in\mathcal{P}_{C'}}\text{length}(e)\\ &> \left(\frac{2^r}{r2^r+s}\right)\sum_{e\in\mathcal{P}_C}\text{length}(e). \qquad (4)\end{aligned}$$

Case 2. C has two bottom-level peak edges whose depths differ by more than 1.

Consider the lowest and highest bottom-level peak edges in C. The lowest peak must have two paths to two leaves below it; the highest such peak must have at least one path to a leaf below it. Say the highest peak edge is of length $2a$ and the lowest peak edge is of length b; the paths below these peak edges are also of lengths $2a$ and b, respectively, by the construction of B_n.

Consider the component C' constructed from C by moving the lowest peak just below the highest bottom-level peak edge. See Figure 4. This decreases the sum of the peak depths by at least 1, so our induction hypothesis applies to C'.

The sum of the lengths of the connecting edges of C and C' are again the same—the connecting path of length $2a$ below the high peak in C has been replaced by two connecting paths of length a in C', and the two connecting paths of length b below the lowest peak in C have been replaced by one connecting path of length $2b$ in C'.

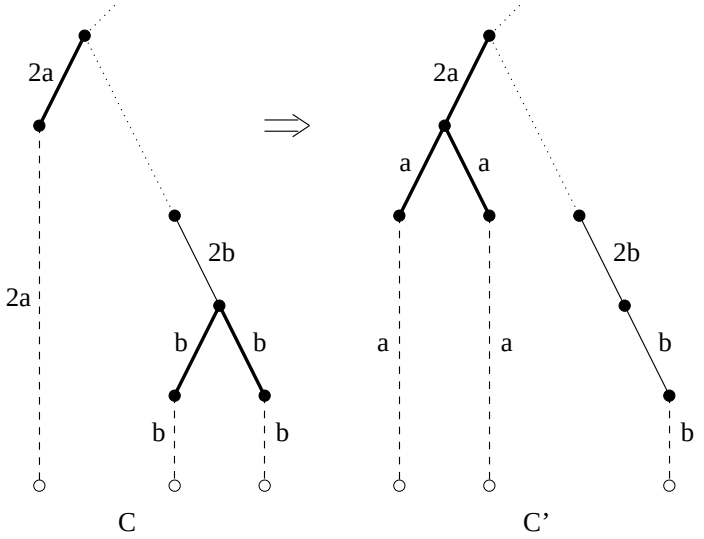


FIG. 4. *C has two bottom-level peak edges whose depths differ by more than* 1.

The sum of the lengths of the peak edges of C is less than that in C'—the two lowest peak edges of length b in C have been moved up, so they have length a in C'. Since the lowest and highest bottom-level peaks differ in depth by at least 2, we know that indeed $b < a$.

Combining this with the induction hypothesis on C', we again get inequality (4).

These two cases complete the induction step, and so we have proved inequality (2) and completed the proof of Theorem 2.5. □

3. Lower bound for ρ_k. The proof of the lower bound will follow the same general outline as the lower bound proof in [4]. We first convert a Steiner tree into a weighted regular binary tree. Then by labeling the vertices of this binary tree, we construct $r2^r + s$ different k-size Steiner trees and show that one of these trees has small enough length to give us the lower bound.

We will use the following lemma from that paper.

LEMMA 3.1. *For any regular binary tree, there exists a one-to-one mapping f from internal vertices to leaves, such that*

(a) *for any internal vertex u, $f(u)$ is a descendant of u;*

(b) *all tree paths $p(u)$ from u to $f(u)$ are edge disjoint.*

Proof. First, we add the following additional requirement:

(c) There is a leaf v so that the path from the root to v is edge disjoint from all other paths $p(u)$.

We prove by induction on the height of the tree that all three conditions can be met. When the tree has height 0, this is trivially true.

Let T be a tree of height $d \geq 1$. T has two subtrees, T_1 and T_2, each of height at most $d-1$ and rooted at the two children of the root of T. By induction, there are functions f_1 and f_2 on the internal vertices of T_1 and T_2 satisfying (a) and (b). There are also vertices v_1 and v_2 in T_1 and T_2, respectively, satisfying (c). Define f as the union of f_1 and f_2 and define $f(\text{root of } T) = v_1$. Clearly, f satisfies (a) and (b) and v_2 satisfies (c) for T. This completes the induction. □

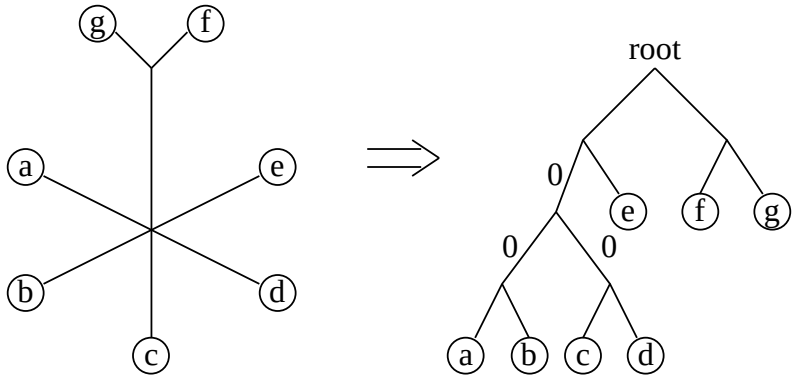


FIG. 5. *Constructing a regular binary tree from a Steiner tree.*

THEOREM 3.2 (lower bound for ρ_k). *For any k, with $k = 2^r+s$, where $0 \le s < 2^r$, we have*

$$\rho_k \ge \frac{r2^r + s}{(r+1)2^r + s}. \tag{5}$$

Proof. We want to prove that for any metric space and any set of points P in that space,

$$\frac{L_S(P)}{L_{kS}(P)} \ge \frac{r2^r + s}{(r+1)2^r + s}. \tag{6}$$

Equation (5) follows immediately because ρ_k is the infimum of these ratios.

We do this by induction on n, the number of points in P. If $n \le k$, then the inequality is true since $L_S(P)/L_{kS}(P) = 1$. For $n > k$, consider the Steiner minimum tree T on P. If T is not a full Steiner tree, then we can split it at a regular point into two smaller Steiner trees, each with fewer than n regular points. Say the regular points of these trees are given by the sets P_1 and P_2. Then we have $L_S(P) = L_S(P_1)+L_S(P_2)$ and $L_{kS}(P) \le L_{kS}(P_1) + L_S(P_2)$, and so

$$\frac{L_S(P)}{L_{kS}(P)} \ge \frac{L_S(P_1) + L_S(P_2)}{L_{kS}(P_1) + L_{kS}(P_2)} \ge \min\left\{\frac{L_S(P_1)}{L_{kS}(P_1)}, \frac{L_S(P_2)}{L_{kS}(P_2)}\right\} \ge \frac{r2^r + s}{(r+1)2^r + s}$$

by our induction hypothesis. Therefore, we only have to consider the case where T is a full Steiner tree, that is, where all regular points are leaves.

By adding zero-length edges and Steiner points, we first modify T to be a tree where every Steiner point has degree exactly 3. Then we choose a root in the middle of an edge to convert T into a weighted regular binary tree; call it T'. The weight of each edge is the length of the edge in the metric space. See Figure 5.

We label all internal vertices of T' with sets of size exactly 2^r chosen from the numbers $\{1, 2, \ldots, r2^r + s\}$. The labeling of a vertex is determined inductively by the labeling of the r vertices above it on a path to the root, its r immediate ancestors.

To begin this labeling, we label the vertex on the first level (the root) with the set $\{1, 2, \ldots, 2^r\}$; we label the two vertices on the second level with the set $\{2^r + 1, 2^r + 2, \ldots, 2 \cdot 2^r\}$; and in general, we label all vertices on the ith level, for $1 \le i \le r$, with the set $\{(i-1)2^r + 1, (i-1)2^r + 2, \ldots, i2^r\}$.

The inductive labeling will maintain the following property, which the labeling of the first r levels clearly satisfies.

Disjointness property. The label sets of up to r consecutive vertices on a path up the tree are disjoint.

Assume that the first i levels have been labeled, $i \geq r$, and that the disjointness property holds up to level i. We label the vertices at level $i+1$ by the following two rules.

Rule 1. Let v be a vertex at level $i+1-r$ with label set $S_v = \{\ell_1, \ell_2, \ldots, \ell_{2^r}\}$. Label the jth descendant of v on level $i+1$ with the set $S_j = \{\ell_j, \ell_{j+1}, \ldots, \ell_{j+2^r-s-1}\}$, where we reduce the subscripts (mod 2^r) so that they are in the range 1 to 2^r.

Vertex v has at most 2^r descendents on level $i+1$, so we need at most the sets $S_1, S_2, \ldots, S_{2^r}$. The sets S_j each have $2^r - s$ elements, and each label ℓ_k from S_v appears in at most $2^r - s$ of the sets, namely $\ell_k \in S_k, S_{k-1}, \ldots, S_{k-2^r+s+1}$, where again we reduce the subscripts (mod 2^r).

To complete the labeling at the $(i+1)$st level, we must add s numbers to each labeling set.

Rule 2. For a vertex at level $i+1$, add to its label set those s labels that are not in the label sets of any of its immediate r ancestors.

By our disjointness property, the r immediate ancestors of a vertex at level $i+1$ are labeled by r disjoint sets of size 2^r, so there must be exactly s numbers from $\{1, 2, \ldots, r2^r + s\}$ unused. Also, the labels added by Rule 2 will be different from the labels added by Rule 1, so all vertices at level $i+1$ are given exactly 2^r labels by the two rules.

The disjointness property now holds up to level $i+1$, since a vertex u at level $i+1$ has labels taken from its rth ancestor's label set, which by the disjointness property at level i are unused by u's $r-1$ immediate ancestors, and s other labels, which are also unused by its $r-1$ immediate ancestors.

By induction, we can label the entire tree. See Figure 6 for an example of this labeling process when $k = 5$.

Now we use this labeling to give us $r2^r + s$ k-size Steiner trees. Each label $\ell = 1, 2, \ldots, r2^r + s$ determines the k-size Steiner tree T_ℓ. Each node labeled ℓ becomes the component root of a component in T_ℓ. In addition, we always have a component in T_ℓ whose component root is the root of T', even if the root of T' is not labeled by ℓ. A component that begins at vertex v, the component root, then connects by paths in T' to the first vertices below v that are also labeled by ℓ; call these vertices the *intermediate leaves* of the component. From the intermediate leaf u, the component then follows the path $p(u)$ to the tree leaf $f(u)$, as given by Lemma 3.1. If there are no vertices labeled ℓ on a path below v, then the component extends along that path all the way down to a tree leaf. See Figure 7.

First, we verify that T_ℓ is in fact a tree that spans P. (Remember that the points of P are the leaves of T'.) This follows by induction on the height of the tree T'. It is trivially true when the height is 0. Look at the component at the top of T_ℓ, the component whose component root is the root of T'; then look at a subtree below an intermediate leaf of this top component. By induction, T_ℓ restricted to this subtree is a tree spanning those points in P that are leaves of the subtree. The top component, which is itself a tree, then joins one tree leaf from each of the subtrees below its intermediate leaves (and any tree leaves of T' that are not below any intermediate leaves) into one large tree.

Now look at a component in tree T_ℓ that has component root v. We verify that

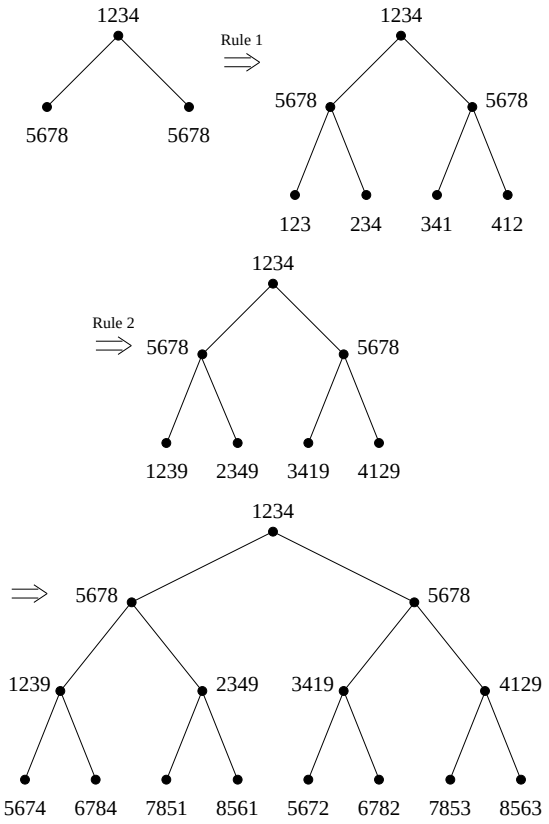


FIG. 6. *Labeling a binary tree when* $k = 5$.

these components are always of size at most k. If a component stops at a tree leaf before reaching an intermediate leaf, its size will be smaller, so we only look at the maximum-sized components that have all possible intermediate leaves.

Suppose v is not labeled by ℓ, so v must be the root of the tree T' and $\ell \geq 2^r + 1$. By the initial labeling, $2^r + 1, 2^r + 2, \ldots, r2^r$ appear on all vertices of one of the $r - 1$ levels below the root, and by Rule 2, the remaining s labels appear on all vertices of the rth level below the root. Thus the intermediate leaves are all at the rth level or above, and so it is of size at most $2^r \leq k$.

Therefore, we can assume that v is itself labeled by ℓ. By Rule 1 of the labeling process, we know that s of the descendants of v r levels below are not labeled by ℓ, and the remaining $2^r - s$ descendants are labeled by ℓ.

Now look at a vertex w that is r levels below the component root v and that is not labeled by ℓ. Since ℓ does label v, by the disjointness property, it cannot label any of the $r - 1$ vertices on the path from v to w. Since ℓ does not label w, by Rule 2 of the labeling process, the children of w must be labeled by ℓ and they will be intermediate leaves.

Thus the component rooted at v has $2s$ intermediate leaves $r + 1$ levels below v and $2^r - s$ intermediate leaves r levels below. Therefore, there are $2^r + s = k$ intermediate leaves, and thus the component is of size k. Hence all components are of size at most k, and T_ℓ is a k-size Steiner tree on the set P of regular points.

In T_ℓ, let L_ℓ be the sum of the lengths of the paths $p(u)$ from intermediate leaves

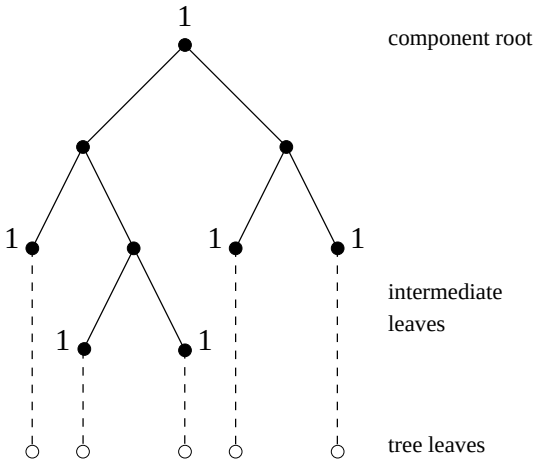


FIG. 7. *A component in T_1 from the labeling of Figure* 6.

u in T_ℓ to tree leaves. Consider the sum $L_1 + L_2 + \cdots + L_{r2^r+s}$. Since each internal vertex u in T' is an intermediate leaf in exactly 2^r of the k-size Steiner trees, namely T_ℓ for each ℓ in the label set of u, the length of the path $p(u)$ will be counted exactly 2^r times in the sum. Since these paths are disjoint for different intermediate leaves, the sum of all of the paths in all of the terms of the sum will be at most 2^r times the length of the entire tree T', which is the length of the Steiner minimum tree. Thus

$$L_1 + L_2 + \cdots + L_{r2^r+s} \leq 2^r L_S(P).$$

Then there must be a d so that

$$L_d \leq \frac{2^r}{r2^r + s} L_S(P).$$

Now the entire length of T_d is the length of the components from the component roots to the intermediate leaves plus the length from the intermediate leaves to the tree leaves. The components from the component roots to the intermediate leaves cover T' exactly once, so this part has length $L_S(P)$. The other part has length L_d. Therefore,

$$\text{length}(T_d) = L_S(P) + L_d \leq \left(1 + \frac{2^r}{r2^r + s}\right) L_S(P).$$

Since $L_{kS}(P) \leq \text{length}(T_d)$, we get

$$\frac{L_S(P)}{L_{kS}(P)} \geq \frac{L_S(P)}{\text{length}(T_d)} \geq \frac{1}{(1 + \frac{2^r}{r2^r+s})} = \frac{r2^r + s}{(r+1)2^r + s}.$$

This is equation (6), and so Theorem 3.2 is proved. □

4. Discussion. Berman and Ramaiyer [1] showed that their polynomial-time approximation algorithm has performance ratio

$$\rho_2^{-1} - \frac{\rho_2^{-1} - \rho_3^{-1}}{2} - \frac{\rho_3^{-1} - \rho_4^{-1}}{3} - \cdots - \frac{\rho_{k-1}^{-1} - \rho_k^{-1}}{k-1}.$$

Our result determined all ρ_k. It follows that there exists a polynomial-time approximation for the Steiner minimum tree in graphs with performance ratio r for any r larger than

$$2 - \frac{2 - \frac{5}{3}}{2} - \frac{\frac{5}{3} - \frac{3}{2}}{3} - \frac{\frac{3}{2} - \frac{13}{9}}{4} - \cdots = 1.734\ldots.$$

Recently, Zelikovsky [16] gave a polynomial-time approximation for the Steiner minimum tree in graphs with performance ratio $\rho_k^{-1}(1 - \log \rho_2 + \log \rho_k)$ which approaches $(1 - \log \rho_2) = 1.693\ldots$ as $k \to \infty$. This approximation is better than Berman and Ramaiyer's for k sufficiently large; however, for smaller k, Berman and Ramaiyer's algorithm still has a better performance ratio. Karpinski and Zelikovsky [13] have improved the algorithm further to give a $1.644\ldots$ approximation.

The interesting open question is whether there exists a polynomial-time approximation for the Steiner minimum tree in graphs with a performance ratio which beats this value.

REFERENCES

[1] P. Berman and V. Ramaiyer, *Improved approximation algorithms for the Steiner tree problem*, J. Algorithms, 17 (1994), pp. 381–408.

[2] M. Bern and P. Plassmann, *The Steiner problem with edge lengths* 1 *and* 2, Inform. Process. Lett., 32 (1989), pp. 171–176.

[3] D.-Z. Du and F. K. Hwang, *A proof of Gilbert and Pollak's conjecture on the Steiner ratio*, Algorithmica, 7 (1992), pp. 121–135.

[4] D.-Z. Du, Y.-J. Zhang, and Q. Feng, *On better heuristic for Euclidean Steiner minimum trees*, Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 431–439.

[5] D.-Z. Du, *On component size bounded Steiner trees*, Discrete Appl. Math., 60 (1995), pp. 131–140.

[6] L. R. Foulds and R. L. Graham, *The Steiner problem in phylogeny is NP-complete*, Adv. Appl. Math., 3 (1982), pp. 43–49.

[7] M. R. Garey, R. L. Graham, and D. S. Johnson, *The complexity of computing Steiner minimal trees*, SIAM J. Appl. Math., 32 (1977), pp. 835–859.

[8] M. R. Garey and D. S. Johnson, *The rectilinear Steiner tree problem is NP-complete*, SIAM J. Appl. Math., 32 (1977), pp. 826–834.

[9] E. N. Gilbert and H. O. Pollak, *Steiner minimal trees*, SIAM J. Appl. Math., 16 (1968), pp. 1–29.

[10] R. L. Graham and F. K. Hwang, *Remarks on Steiner minimal trees*, Bull. Inst. Math. Acad. Sinica, 4 (1976), pp. 177–182.

[11] F. K. Hwang, *On Steiner minimal trees with rectilinear distance*, SIAM J. Appl. Math., 30 (1976), pp. 104–114.

[12] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computation, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[13] M. Karpinski and A. Z. Zelikovsky, *A new approach to approximation of Steiner trees*, manuscript.

[14] H. Takahashi and A. Matsuyama, *An approximate solution for the Steiner problem in graphs*, Math. Japon., 24 (1980), pp. 573–577.

[15] A. Z. Zelikovsky, *The* 11/6*-approximation algorithm for the Steiner problem on networks*, Algorithmica, 9 (1992), pp. 463–470.

[16] A. Z. Zelikovsky, *Better approximation bounds for the network and Euclidean Steiner tree problems*, manuscript.

SIAM J. COMPUT.
Vol. 26, No. 3, pp. 870–872, June 1997

015

A NOTE ON "AN ON-LINE SCHEDULING HEURISTIC WITH BETTER WORST CASE RATIO THAN GRAHAM'S LIST SCHEDULING"*

R. CHANDRASEKARAN†, BO CHEN‡, GÁBOR GALAMBOS§, P. R. NARAYANAN†, ANDRÉ VAN VLIET¶, AND GERHARD J. WOEGINGER‖

Key words. combinatorial problems, scheduling, worst-case bounds, on-line algorithms

AMS subject classifications. 90B35, 90C27

PII. S0097539793258775

1. Introduction. This is an erratum for the paper by Galambos and Woeginger [2]. The proofs of two basic results in [2], Lemmas 4.3 and 4.5, are incorrect. The flaw in the proof of Lemma 4.3 can be easily fixed, but a rather different approach is needed in order to prove Lemma 4.5.

2. Corrections.

2.1. Correction to Lemma 4.3. The assertion "$L_1 > \alpha L_m \geq \alpha C^*$" on line 15 of page 352 is incorrect and should be replaced by "$L_1 > \alpha \min\{C^*, L_m\}$." Inequality (8) should be strengthened to

$$L_1 + x \leq \beta \min\{C^*, L_m\}.$$

Then (9) can be replaced by

$$\begin{aligned} L_2 + x &\leq (mC^* - L_1 - x)/(m-1) + x \\ &= mC^*/(m-1) - L_1 + (L_1 + x)(m-2)/(m-1) \\ &< mC^*/(m-1) + (\beta(m-2)/(m-1) - \alpha) \min\{C^*, L_m\} \\ &\leq C^*(m + (m-2)\beta - (m-1)\alpha)/(m-1). \end{aligned}$$

2.2. Correction to Lemma 4.5. In the proof, the claim "Moreover by the definition of M_j, no other machine received a job between the primed and the unprimed moment" is incorrect. However, Lemma 4.5 follows directly as a corollary of the following result.

LEMMA 4.5′. *Assume the algorithm enters Step* 4. *Denote by a_i the job assigned last to machine M_i. Then there exists a $k \geq 2$ such that*

$$\text{(a)} \quad L_j - a_j \leq L_1 \quad \forall j \geq 2,\ j \neq k,$$
$$\text{(b)} \quad L_k - a_k \leq \beta L_1.$$

* Received by the editors November 23, 1993; accepted for publication (in revised form) November 21, 1995.
http://www.siam.org/journals/sicomp/26-3/25877.html
† University of Texas at Dallas, P.O. Box 830688, Richardson, TX 75083-0688 (chandra@utdallas.edu, pr@pros.prosx.com).
‡ Warwick Business School, University of Warwick, Coventry CV4 7AL, United Kingdom (bchen@warwick.ac.uk).
§ Department of Applied Computer Sciences, József Attila University, H-6720 Szeged, Hungary (h762gal@ella.hu).
¶ ORTEC Consultants B. V., Groningenweg 6-33, 2803 PV Gouda, The Netherlands (andre@ortec.nl).
‖ Institut für Mathematik, Technische Üniversität Graz, A-8010 Graz, Austria (gwoegi@figids01.tugraz.ac.at).

Proof. Choose $k \geq 2$ such that M_k is the machine that received its last job a_k earlier than all other machines M_j received their last job a_j ($j \geq 2$). Since $\sim (L_1 + x, L_2, \ldots, L_m)$ and $L_1 \leq \alpha L_m$, we have $L_1 \leq \alpha\beta L_k$ and $L_1 \leq \alpha\beta L_j$.

Proof of (a). Since a_j is the last job on M_j, it cannot have been placed in Step 4.1. We consider the cases where a_j was placed in Step 2, 3, or 4.2. Let us denote by L'_i the workload of machine M_i, $1 \leq i \leq m$, just before machine M_j receives its last job a_j.

Suppose that M_j received its last job in Step 4.2. Before M_j received a_j, the current schedule at that time satisfied $\sim (L'_1, L'_2, \ldots, L'_m)$. Thus

$$L_1 \geq L'_1 \geq \frac{1}{\beta} L'_k = \frac{1}{\beta} L_k \geq \frac{1}{\alpha\beta^2} L_1.$$

This contradicts our initial assumption that $\alpha \leq \frac{1}{3}$ and $\beta \leq \frac{5}{4}$. Therefore, we conclude that a_j was not placed in Step 4.2.

Suppose that M_j received its last job in Step 3. This means that L'_j was the second smallest workload at that time. Denote the minimum workload at that moment by $L'_{\min}$. If we had assigned job a_j to the machine of minimum workload, then the schedule would have become similar. This implies that $L'_k \sim L'_{\min} + a_j$ and $L'_k \sim L'_j$. Since $L_j \sim L_k$, we can write $L_j = xL_k$, where $\frac{1}{\beta} \leq x \leq \beta$. Then

$$L_j = xL_k = xL'_k \leq x\beta L'_j = x\beta(L_j - a_j).$$

On the other hand, we have

$$\frac{1}{\beta} L_j - xa_j = x\left(\frac{1}{\beta} L'_k - a_j\right) \leq xL'_{\min} \leq xL_1 \leq x\alpha\beta L_k = \alpha\beta L_j.$$

Combining the above two relations gives us

$$\left(\frac{1}{\beta} - \alpha\beta\right) L_j \leq xa_j \leq \left(x - \frac{1}{\beta}\right) L_j \quad \Rightarrow \quad \frac{1}{\beta} - \alpha\beta \leq x - \frac{1}{\beta} \leq \beta - \frac{1}{\beta},$$

which contradicts the initial assumptions that $\beta + 1 > \beta^3(\alpha + 1)$ and $\beta \geq 1$. We then conclude that a_j was not placed in Step 3.

The only possibility that remains is that M_j received a_j in Step 2. This means that L'_j was the smallest workload at that moment. Hence $L_j - a_j = L'_j \leq L'_1 \leq L_1$.

Proof of (b). We again need to consider the three cases where a_k was placed in Step 2, 3, or 4.2. If a_k was placed in Step 2, then machine M_k had the smallest workload at that time. Thus $L_k - a_k \leq L_1$. If a_k was placed in Step 3, then from conclusion (a), we know that all other machines had workloads less than or equal to L_1 at that time. Since M_k had the second smallest workload, we conclude that $L_k - a_k \leq L_1$. The last possibility that remains is that a_k was placed in Step 4.2. Since the schedule was similar just before a_k was placed, we conclude that $L_k - a_k \leq \beta L_1$. □

3. Remarks. As we have seen, Lemma 4.5 has been strengthened to Lemma 4.5′. A claim stronger than Lemma 4.3 that allows us to give the exact worst-case bound of RLS has also been established. Further investigations of RLS yield an improved algorithm, RLS2. Interested readers are referred to [1] for details.

REFERENCES

[1] B. Chen and A. van Vliet, *On the on-line scheduling algorithm RLS*, Report 9325/A, Econometric Institute, Erasmus University, Rotterdam, The Netherlands, 1993.

[2] G. Galambos and G. J. Woeginger, *An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling*, SIAM J. Comput., 22 (1993), pp. 349–355.

[3] P. R. Narayanan, *Performance analysis of on-line algorithms under various scheduling criteria*, Ph.D. thesis, University of Texas at Dallas, Dallas, TX, 1992.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 873–933, August 1997

001

AN OPTIMAL PROBABILISTIC PROTOCOL FOR SYNCHRONOUS BYZANTINE AGREEMENT*

PESECH FELDMAN† AND SILVIO MICALI‡

Abstract. Broadcasting guarantees the recipient of a message that everyone else has received the same message. This guarantee no longer exists in a setting in which all communication is person-to-person and some of the people involved are untrustworthy: though he may claim to send the same message to everyone, an untrustworthy sender may send different messages to different people. In such a setting, *Byzantine agreement* offers the "best alternative" to broadcasting. Thus far, however, reaching Byzantine agreement has required either many rounds of communication (i.e., messages had to be sent back and forth a number of times that grew with the size of the network) or the help of some external trusted party.

In this paper, for the standard communication model of synchronous networks in which each pair of processors is connected by a private communication line, we exhibit a protocol that, in probabilistic polynomial time and without relying on any external trusted party, reaches Byzantine agreement in an expected constant number of rounds and in the worst natural fault model. In fact, our protocol successfully tolerates that up to 1/3 of the processors in the network may deviate from their prescribed instructions in an arbitrary way, cooperate with each other, and perform arbitrarily long computations.

Our protocol effectively demonstrates the power of randomization and zero-knowledge computation against errors. Indeed, it proves that "privacy" (a fundamental ingredient of one of our primitives), even when is not a desired goal in itself (as for the Byzantine agreement problem), can be a crucial tool for achieving correctness.

Our protocol also introduces three new primitives—graded broadcast, graded verifiable secret sharing, and oblivious common coin—that are of independent interest, and may be effectively used in more practical protocols than ours.

Key words. broadcasting, Byzantine agreement, fault-tolerant computation, randomization

AMS subject classifications. 68Q22, 68R05, 68M15, 94A60, 94A99, 94B99

PII. S0097539790187084

1. The problem.

A motivating scenario. We are in Byzantium, the night before a great battle. The Byzantine army, led by a commander in chief, consists of n legions, each one separately encamped with its own general. The empire is declining: up to 1/3 of the generals—including the commander in chief—may be traitors. No radios (sic!) are available: all communication is via messengers on horseback. To make things worse, the loyal generals do not know who the traitors are. During the night each general receives a messenger with the order of the commander for the next day: either "attack" or "retreat." If all the good generals attack, they will be victorious; if they all retreat, they will be safe; but if some of them attack and some retreat they will be defeated. Since a treasonous commander in chief may give different orders to different generals, it is not a good idea for the loyal ones to directly execute his orders. Asking the opinion of other generals may be quite misleading too: traitors may represent their orders differently to different generals, they may not send any information to someone,

* Received by the editors August 30, 1990; accepted for publication (in revised form) July 31, 1995. An earlier version of this work was presented at the 1988 ACM Symposium on the Theory of Computing (STOC).

http://www.siam.org/journals/sicomp/26-4/18708.html

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Current address: OHR SOMAYACH, 22 Shimon Hatzedik, Jerusalem, Israel.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (silvio@theory.lcs.mit.edu). The research of this author was supported in part by NSF grants DCR-84-13577 and CCR-9121466, ARO grant DAALO3-86-K-0171, and ONR grant NOOO14-92-J-1799.

and they may claim to have received nothing from someone else. On the other hand, should the honest generals always—say—attack (independently of the received orders and of any discussion), they would not follow any meaningful strategy. What they need is a way to exchange messages so as to always reach a common decision while respecting the chief's order, should he happen to be honest. They need *Byzantine agreement*.

Byzantine agreement. As insightfully defined by Pease, Shostak, and Lamport [32], Byzantine agreement essentially consists of providing "the best alternative" to broadcasting when all communication is person-to-person (as in an ordinary telephone network) and some of the people involved are untrustworthy. In order to briefly describe what this alternative is, we must first sketch its classic underlying communication model, the most convenient and simplest one in which the need for Byzantine agreement arises.

Modernizing the motivating scenario a bit, generals are processors of a computer network. Every two processors in the network are joined by a separate communication line, but no way exists to broadcast messages. (Thus, though a processor can directly send a given message to all other processors, each recipient has no way to know whether everyone else has received the same message.) The network otherwise has some positive features. Each processor in it has a distinct identity and knows the identities of the processors on the other end of its lines. The network is *synchronous*, that is, messages are reliably delivered in a sufficiently timely fashion: there is a common clock, messages are sent at each clock tick (say, on the hour) and are guaranteed to be delivered by the next tick (though not necessarily simultaneously). Each communication line is *private*, that is, no one can alter, inject, or read messages traveling along it. Indeed, the only way for an adversary to disturb the communication of two good processors is by corrupting one of them. We will refer to such a network as a *standard network* since it is the one generally adopted for discussing the problem of Byzantine agreement.[1]

Now assume that each of the processors of a standard network has an initial value. Then, speaking informally, a Byzantine agreement protocol should guarantee that for *any* set of initial values, the following two properties hold:

1. *Consensus*: All honest (i.e., following the protocol) processors adopt a common value.
2. *Validity*: If all honest processors start with the same value, then they adopt that value.[2]

Byzantine faults. Having briefly discussed our communication model, we must now mention our fault model. Processors become faulty when they deviate from their prescribed programs. "Crashing" (i.e., ceasing all activities) is a *benign* way for a processor in a network to become faulty. The faulty behavior considered in this paper is instead much more adversarial: faulty processors may deviate from their prescribed programs in an arbitrary fashion, perform an arbitrary amount of computation, and

[1] Standard networks are advantageous to consider in that they allow one to focus on the novel characteristics of Byzantine agreement without being distracted by legitimate but "orthogonal" concerns. We wish to stress, however, that, while the absence of broadcasting is crucial for the problem of Byzantine agreement to be meaningful, we shall see in section 9.1 that most of the fine details of the adopted communication model can be significantly relaxed without affecting our result.

[2] Notice that we have stated Byzantine agreement a bit more generally than in the motivating scenario; namely, the processors are not given their initial values by a distinguished member of their group but have their own individual sources for these values. Consider, for instance, the case of party bosses who, before an election, call each other on the phone to select a common candidate to back: even though their initial choices do not arise from the suggestion of a distinguished boss, they still need Byzantine agreement.

even coordinate their disrupting actions. Modeling software and hardware faults as malicious behavior may not be unreasonable when dealing with the interaction of very complex programs, but it is actually *very* meaningful, and even mandatory, if there are people behind their computers. Indeed, whenever we wish to emphasize the possibility of human control—and thus that of malicious behavior—we do employ the term "player" instead of processor.

The goal of the faulty players is to disrupt either the consensus or the validity requirement or simply to delay reaching Byzantine agreement for as long as possible (as when, say, they prefer the status quo to any of the two alternatives being voted on). Here is an example of what malicious players may do against a simple-minded protocol.

Assume that the honest generals of the motivating scenario try to reach agreement as follows: they send their initial orders to each other and then execute the most "popular" order. Then the dishonest generals can easily cause disagreement. To make our example more dramatic, let us suppose that 2/3 of the generals are loyal, that half of the loyal ones start with the value "attack," and that the other loyal half start with "retreat."[3] In this situation, the traitors simply tell every loyal general in the first half that their initial value is "attack" and every loyal one in the second half that their value is "retreat." Consensus is then disrupted in a most dramatic way: half of the loyal generals will attack and the other half will retreat. Indeed, reaching Byzantine agreement is a tricky business.[4]

The significance of Byzantine agreement. Byzantine agreement is widely considered the standard bearer in the field of fault-tolerant distributed computation. While it is indisputable that this problem has attracted an enormous amount of attention, we are skeptical about its relevance in the context of errors *naturally* occurring in a distributed computation. In our opinion, Byzantine agreement is relevant to the field of *secure computation protocols*, which includes problems such as electronic elections, electronic negotiations, or electronic bids.

Secure protocols (see [30] for a satisfactory and general definition) is a new and exciting branch of mathematics that has experienced impressive growth in recent years. A problem in this field consists of enabling a group of mutually distrustful parties to achieve, in a interaction in which some of the players do not follow the rules of the game, the same results that are obtainable by exchanging messages in a prescribed manner when there is total and honest collaboration. Indeed, it is thanks to insights from the field of secure protocols that we have succeeded in finding our optimal probabilistic solution to the synchronous Byzantine agreement problem.

Byzantine agreement plays an important role in secure protocol theory; essentially, it dispenses with the need to hold a meeting when, because of the presence of adversaries among us, it is useful to establish in a public manner who said what or what was decided upon. In the simplest secure protocol or in the most complex one,

[3] These initial values are not at all unlikely if they represent (as in our motivating scenario) the individual version of an alleged unique message sent by a dishonest party. In any case, consensus and validity are very strong requirements: they should hold for any initial values!

[4] As we shall mention in the next section, at least t rounds of communications are needed to reach Byzantine agreement whenever (1) t parties are dishonest and (2) the honest ones follow a deterministic protocol. This fact immediately yields an alternative way to dismiss the simple-minded strategy discussed above: it is deterministic and can be implemented in *two* rounds, no matter how many players there are and no matter how many of them can be faulty. The same fact allows one to dismiss a good deal of other simple-minded strategies as well. As we shall see, it is only through a careful use of randomization that a strong majority of honest players may reach Byzantine agreement very fast.

the honest players cannot possibly make any progress without keeping a meaningful and consistent view of the world. This is what Byzantine agreement gives us.

The quality of a Byzantine agreement protocol. Several aspects are relevant in determining the quality of a Byzantine agreement protocol. As for most protocols, the amount of local computation and the total number of message bits exchanged continue to be important. But in this archetypal problem in distributed adversarial computation, two are the most relevant (and most investigated) aspects: the *round complexity* and the *fault model*.

The round complexity measures the amount of interaction that a protocol requires.[5] Since, at each clock tick, a player may send messages to more than one processor (and their recipients will receive them by the next tick), the round complexity of a protocol naturally consists of the total number of *rounds* (i.e., clock pulses and thus "waves" of messages) necessary for completing the protocol.

The fault model specifies what can go wrong (while still being tolerable somehow) in executing a protocol, namely the following: How many processors can become faulty? How much can they deviate from their prescribed programs? How long can faulty processors compute to pursue their disruptive goals?

In this light, the goal of a Byzantine agreement protocol naturally consists of simultaneously "decreasing" the round complexity while "increasing" the fault model.

Our solution. We present a probabilistic-polynomial-time protocol that reaches Byzantine agreement in an expected constant number of rounds (thus minimizing the round complexity) while tolerating the maximum possible number of faulty players and letting them exhibit a most malicious behavior.

2. Previous solutions and ours.

2.1. The worst natural fault model. Though several weaker models for Byzantine agreement can be considered (see the excellent surveys of Fischer [19] and Chor and Dwork [11] for a more comprehensive history of this subject), in this paper, we concentrate on a most adversarial setting. Speaking informally for now, the *worst (natural) fault model* is characterized by the following three conditions:

1. the good players are bound to polynomial-time computation;
2. a constant fraction of the total number of players may become faulty; and
3. the faulty players can deviate from their prescribed instructions in any arbitrary way, perform arbitrarily long computations, and perfectly coordinate their actions.

The worst fault model is not only the most difficult one to handle but also, in our opinion, the most meaningful one to consider. Condition 1 essentially expresses that for a Byzantine agreement protocol to be useful, the computational effort required by the honest processors should be reasonable. Condition 2 properly captures our intuition about the nature of faults, independently of whether we consider players as machines or people controlling machines. Indeed, while we do expect that the number of faulty players grows with the size of the network, it would be quite counterintuitive to expect that it grows sublinearly in this size. (For instance, assume that in a network of n players the number of bad ones is $n/\log n$. Then this would mean that, while we expect 1% of a group of 1000 players to be faulty, we expect a *smaller* percentage of faulty players in a much larger group.) Condition 3 essentially captures that there may be people behind their computers: dishonest people follow whatever strategy is

[5] In a distributed setting, this is the most expensive resource. Typically, the time invested by the processors for performing their local computation is negligible with respect to the time necessary to send electronic mail back and forth several times.

best for them, try much harder than honest ones, and effectively cooperate with one another. In any case, by successfully taming *malicious* faults, we would a fortiori succeed in taming all other more benign—though not necessarily more reasonable—ones.

Let us thus review the main protocols in this difficult model.

2.2. Previous solutions. Dolev et al. [12] exhibited the first solution in the worst fault model. Letting n denote the total number of players in the network and t denote an upperbound on the number of faulty players, they showed that as long as $t < n/3$, Byzantine agreement can be reached deterministically in $\Theta(t)$ rounds. Recently, by a different protocol, Moses and Waarts [31] tightened their number of rounds to be $t+1$ for $t < n/8$. This is optimal for their choice of t, since Fischer and Lynch [20] proved that t rounds are required by any deterministic protocol if t faults may occur in its execution.

In light of the lower bound mentioned, all hope for faster agreement is entrusted to probabilism. Indeed, since the pioneering work of Ben-Or [5], randomization has been extensively used for reaching agreement. In particular, Rabin's notion of a *common coin* [34] has emerged as the right version of probabilism for this setting. A network with a common coin can be described as a network in which a random bit becomes available to all processors at each round but is unpredictable before then. The interest of this notion is due to a reduction of Rabin showing that as long as $t < n/4$, Byzantine agreement can be reached in expected constant number of rounds with the help of a common coin. Of course, common coins are not a standard feature of a point-to-point network; thus this reduction raises a natural and important question: *Are there efficient Byzantine agreement protocols implementable "within the network" and in the worst fault model*?

Prior to our work, no efficient within-the-network Byzantine agreement protocol was known for the worst fault model. Rabin [34] devised a cryptographic Byzantine agreement protocol running in an expected constant number of rounds but relying on a incorruptible party external to the network.[6] Bracha [6] exhibited Byzantine agreement protocols that do not require trusted parties, but his protocols are slowers (they run in expected $O(\log n)$ rounds) and are not explicitly constructed (their existence is proved by counting arguments). Chor and Coan [8] exhibit an explicit and within-the-network Byzantine agreement protocol, but their solution, though attractively simple, is much slower (their protocol tolerates any $t < n/3$ faults but runs in $O(t/\log n)$ rounds; thus it requires expected $O(n/\log n)$ rounds in the worst fault model). Feldman and Micali [18] explicitly exhibited a cryptographic within-the-network protocol that, after a preprocessing step consisting of a single Byzantine agreement (on some specially generated keys), allows any subsequent agreement to be reached in an expected constant number of rounds.[7] While their protocol is actually very practical after the first agreement has been reached, the first agreement may

[6] Rabin's algorithm uses digital signatures—which implies that dishonest processors are bound to polynomial-time computation—and a *trusted party*—i.e., an incorruptible processor outside the network. In his solution, if the trusted party distributes k pieces of reliable information to the processors in the networks in preparation, then these processors can, subsequently and without any external help, compute k common coins. Thus the number of reachable agreements is bounded by the amount of information distributed by the trusted party in the preprocessing stage. A cryptographic Byzantine agreement protocol with a trusted party but without the latter limitation was later found by the authors in [18], in addition to other results mentioned later on.

[7] Thus their protocol does not require any preprocessing if a trusted party distributes the right keys beforehand. The present result can thus be viewed as removing cryptography and preprocessing from their protocol.

very well be the most important one (i.e., whether or not to hold a meeting).

To complete the picture, let us mention that Dwork, Shmoys, and Stockmeyer [16] found a beautiful Byzantine agreement protocol running in expected constant round but not in the worst fault model. (Their algorithm tolerates only $O(n/\log n)$ faults.)

2.3. Our solution. The main theorem in this paper can be informally stated as follows:

There exists an explicit protocol $\mathcal{P}$ reaching Byzantine agreement in the worst fault model and running in in an expected constant number of rounds. Protocol $\mathcal{P}$ actually tolerates any number of faults less than one third of the total number of processors.

Our protocol is probabilistic in the "best possible way": it is *always correct* and *probably fast*; that is, an unlucky sequence of coin tosses may cause our protocol to run longer, but when it halts both consensus and validity are guaranteed to hold. Our algorithm not only exhibits optimal (within a constant) round complexity, but it also achieves optimal fault tolerance. In fact, Karlin and Yao [28] have extended the earlier deterministic lower bound of [32] by showing that even probabilistically Byzantine agreement is unreachable if $t = n/3$ faults may occur.[8]

3. Model of computation. As of today, unfortunately, no reasonable treatment of the notion of probabilistic computation in a malicious fault model can be conveniently pulled off the shelf. (A comprehensive effort in this direction—in the more general context of *secure computation*—was made in [30], but this paper has not yet appeared in print.[9]) Thus we have found it *necessary* to devote a few pages to discuss—though only at a semiformal level— of definitions in what we intended to be a purely algorithmic paper.

The definitions below, presented only at a semiformal level, focus solely on what we immediately need to discuss our Byzantine agreement protocol, purposely ignoring many other subtle issues (addressed in the quoted paper [30]). We only wish to clarify what it means that, in the execution of an n-party protocol, t of the processors may make errors (i.e., deviate from their prescribed instructions) in a most malicious way and that the protocol tolerates these faults.

Basic notation. Below we assume that a proper encoding scheme is adopted. Thus we can treat a string or a set of strings over an arbitrary alphabet as a binary string, we may consider algorithms that output (the encoding of) another algorithm, etc.

We assume that each finite set mentioned in our paper is ordered. If $S_1, \ldots, S_k$ are finite sets, we let the instruction $\forall x_1 \in S_1 \ldots \forall x_k \in S_k\ Alg(x_1, \ldots, x_k)$ stand for the program consisting of running algorithm Alg first on input the first element of $S = S_1 \times \cdots \times S_k$, then ("from scratch," i.e., in a memoryless fashion) on input the second element of S, and so on.

[8] This remains true, as proved by Dolev and Dwork [14], even if one abandons the worst fault model so as to include cryptographic protocols (against faulty processors with polynomially bounded resources). Thus the optimality of our algorithm is retained in this setting as well.

[9] Byzantine agreement aims only at guaranteeing *correctness* in the presence of an adversary (about what was decided upon) but not at keeping secret the original single-bit inputs of the players. A secure protocol must instead simultaneously ensure that a given computation (on inputs some of which are secret) is both correct and *private*, that is, roughly, not revealing the initially secret individual inputs more than is implicitly done by the desired output of the computation. This is much more difficult both to handle and to formalize.

The symbol ":=" denotes the *assignment* instruction. The symbol "$\circ$" denotes the concatenation operator. If σ is a string and τ is a prefix of σ, we denote by the expression "σ/τ" the string ρ such that $\sigma = \tau \circ \rho$.

If *Alg* is a probabilistic algorithm, I is a string, and R is a infinite sequence of bits, by running *running Alg on input I and coins R* we mean the process of executing *Alg* on input I so that, whenever *Alg* flips a coin and $R = b \circ R'$, the bit b is returned as the result of the coin toss and $R := R'$.

Protocols. To avoid any issue of nonconstructiveness, we insist that protocols be uniform.

DEFINITION 1. *Let n be an integer greater than* 1. *An* n-party protocol *is an n-tuple of probabilistic algorithms, $P_1, \ldots, P_n$, where each P_i (which is intended to be run by player i) satisfies the following property. On any input (usually representing player i's previous history in an execution), P_i halts with probability* 1 *computing either an n-tuple of binary strings (possibly empty, representing i's messages to the other players for the next round) or a triple consisting of an n-tuple of strings (with the same interpretation as before), the special character* TERMINATE, *and value v (as its output).*

Notice that each time that P_i is run, one also obtains as "side products" the sequence of coin tosses actually made by P_i and the sequence of its "future" coin tosses.

A protocol *is a probabilistic algorithm that, for all integers $n > 1$, on input the unary representation of n, outputs (the encoding of) an n-party protocol.*

In this paper, the expression *round* denotes a natural number; in the context of an n-party protocol, the expression *player* denotes an integer in the closed interval $[1, n]$.

Executing protocols without adversaries. Let us first describe the notion of executing a protocol when all players are honest. Intuitively, each party runs his own component of the protocol. The only coordination with other parties is via messages exchanged in an organized fashion. Namely, there is a common clock accessible by all players, messages are sent at each clock tick along private communication channels, and they are received by the next tick. The interval of time between two consecutive ticks is called a *round*. At the beginning of a round, a player reads the messages sent to him in the previous round, and then runs (his component of) the protocol to compute the messages he sends in response. These outgoing messages are computed by a player by running the protocol on the just-received incoming messages and its own past "history," (i.e., an encoding of all that has happened to the player during the execution of the protocol up to the last round). We now describe this intuitive scenario a bit more precisely, though not totally formally. In so doing, parties, hardware, private communication channels, and clocks will disappear. However, they will remain in our terminology for convenience of discourse.

DEFINITION 2. *Let n be an integer > 1, $P = (P_1, \ldots, P_n)$ be an n-party protocol, $p_1, \ldots, p_n$ be finite strings, and $R_1, \ldots, R_n$ be infinite binary sequences. Then by* executing P on private inputs $p_1, \ldots, p_n$ and coins $R_1, \ldots, R_n$, *we mean the process of generating, for each player i and round r, the quantities*

- H_i^r, *a string called the* history of player i at round r (*a triple consisting of* (1) *i's history prior to round r,* (2) *the messages received by i in round r, and* (3) *the coin tosses of i in round r*),
- $M_{i\rightarrow}^r$, *the* messages sent by player i in round r (*an n-tuple of strings whose jth entry, $M_{i\rightarrow}^r[j]$, is called* the message sent by i to j in round r),
- $M_{\rightarrow i}^r$, the messages received by player i at round r (*an n-tuple of strings, whose*

jth entry, $M^r_{\to i}[j]$, is called the message received by i from j in round r),

- C^r_i, the coin tosses of i in round r (*a substring of* R_i), *and*
- R^r_i, the coin tosses of i after round r (*a substring of* R_i)

by executing the following instructions:

(Start) *Set* $C^0_i = \epsilon$, $R^0_i = R_i$, $M^0_{i\to} = M^0_{\to i} = (\epsilon, \ldots, \epsilon)$, *and* $H^0_i = (p_i, M^0_{\to i}, C^0_i)$.

"Only the individual input is available at the start of an execution: no message has yet been sent or received, and no coin has been flipped."

(Halt) *Say that* player i halts at round r (*and his* output is σ) *if* r *is the minimum round* s *for which* P_i, *on input* H^{s-1}_i *and coins* R_i, *computes a triple whose second entry is the special character* TERMINATE (*and whose third entry is* σ.) *If* i *halts in round* r, *then* $\forall s > r$, $M^s_{i\to} := (\epsilon, \ldots, \epsilon)$, $M^s_{\to i} := (M^s_{1\to}[i], \ldots, M^s_{n\to}[i])$, $C^s_i := \epsilon$, $R^s_i := R^r_i$, *and* $H^s_i := (H^{s-1}_i, M^s_{\to i}, \epsilon)$.

(Continue) *If* i *has not halted in a round* $< r$, *run* P_i *on input* H^{r-1}_i *and coins* R^{r-1}_i *so as to compute either* (a) *an* n*-tuple of string* M *or* (b) *a triple* $(M, \text{TERMINATE}, v)$, *where* M *is an* n*-tuple of strings,* TERMINATE *is a special character, and* v *is a string. If* C *is actually the entire sequence of coin tosses that* P_i *has made in this computation—and thus* C *is a prefix of* R^{r-1}_i*—then* $C^r_i := C$, $R^r_i := R^{r-1}_i / C$, $M^r_{i\to} := M$, $M^r_{\to i} := (M^r_{1\to}[i], \ldots, M^r_{n\to}[i])$, *and* $H^r_i := (H^{r-1}_i, M^r_{\to i}, C^r_i)$.

For simplicity's sake (since each P_i, on any input, halts with probability 1), above we have neglected dealing with protocol "divergence." Also for simplicity, we let a player, at each round, run his own version of the protocol, P_i, on the just-received messages and on the entire history of his execution of the protocol. This is certainly wasteful. In most practical examples, in fact, it suffices to remember very little of the past history. Also notice that the current round number is not an available input to P_i, but it can be easily derived from the current history. In our protocols, however, we make players very much aware of the round number. In fact, we actually spell out what each P_i should do separately for each round. Notice also that the strings R_i need not to be given "in full." It suffices that a mechanism is provided that "retrieves and deletes" R_i's first bit.

Adversaries. We now allow malicious errors to occur in the execution of a protocol. A processor that has made an error is called *faulty* or *bad*. To formalize the idea that faulty processors may coordinate their strategies in an optimal way, we envisage a single external entity, *the adversary*, that chooses which processors to corrupt and sends messages on behalf of the corrupted processors. Since we wish our adversary to be as strong as possible, we allow it to be a nonuniform probabilistic algorithm. (In fact, in our protocol, we might as well assume that an adversary is an arbitrary probabilistic noncomputable function.)

DEFINITION 3. *Let* n *be an integer greater than* 1. *An* n-party adversary *is a probabilistic algorithm that, on any input* (*usually representing* A*'s previous activity in an execution*) *halts with probability* 1 *and outputs either an integer in the range* $[1, n]$ (*the identity of a newly corrupted player*) *or a sequence of pairs* (j, M), *where* j *is an integer between* 1 *and* n (*the identity of a corrupted player*) *and* M *is an* n*-tuple of strings* (*the messages sent by* j *in the current round*). *An* adversary, A, *is a sequence of* n*-party adversaries:* $A = \{A(n) : n = 2, 3, \ldots\}$.

Executing protocols with an adversary. We now define what it means for an n-party protocol P to be executed with an n-party adversary A. A enters the execution with an *initial adversarial history*, a string denoted symbolically by H^0_A, and an *initially bad set*, $\text{BAD}^0 \subset [1, n]$. String H^0_A may contain some a priori knowledge about the inputs of the players, the result of previous protocols, and so on. Set BAD^0 represents the players corrupted at round 0, that is, before the protocol starts. (In

other words, if P were the first protocol "ever to be executed," BAD^0 would be empty. If, as we shall see, P were called as a subprotocol, BAD^0 would comprise all the players that have been corrupted prior to calling P.) Adversary A may, at any round, *corrupt* an additional processor, j. When this happens, all of j's history becomes available to A;[10] as for all corrupted processors, all future messages sent to j will be read by A; and A will also compute all of the messages that j will be sending. Essentially, j becomes an extension of A. Thus if $k \in \text{BAD}^0$, k's private input is what becomes available to A at the start of P, and A will totally control player k for the entire execution of P.

Since we want to prepare for the worst, we let the adversary be even more powerful by allowing *rushing*; that is, we let the message delivery (which is not simultaneous) be as adversarial as possible. At the beginning of each round, all currently good players read the messages sent to them in the previous round and compute the ones that they wish to send in the present round. We pessimistically assume that the messages addressed to the currently corrupted processors are always delivered immediately, and if based on this information the adversary decides to corrupt an additional processor j, we pessimistically assume that it succeeds in doing so before j has sent any messages to the currently good players, thus giving A a chance to change these messages. Further, we consistently "iterate this pessimism" within the same round. That is, once j is corrupted in round r, we assume that the messages addressed to j by the currently good processors are immediately delivered, while j has not yet sent any messages to the remaining good players. This way A may decide whom to corrupt next in the same round, and so on, until A does not wish to corrupt anyone else in round r. At this point, A computes all messages sent by the corrupted processors in round r. These "bad" messages will be read by the good processors (of course, each processor receives the messages addressed to him), together with all "good" messages, at the beginning of round $r+1$.

The privacy of the communication channels of a concrete network is captured in the formulation below by the fact that messages exchanged between uncorrupted processors are never an available input to the adversary algorithm.

The history of a bad player is essentially frozen at the moment in which he is corrupted because A has essentially subsumed him from that point on.

DEFINITION 4. *Let n be an integer > 1, H_A^0, $p_1, \ldots, p_n$ be finite strings, $R_A, R_1, \ldots, R_n$ infinite binary sequences, BAD^0 be a subset of $[1,n]$ and GOOD^0 be its complement, $P = (P_1, \ldots, P_n)$ be an n–party protocol, and A be an n-party adversary. Then by* executing P with A on initial adversarial history H_A^0, inputs $p_1, \ldots, p_n$, initially bad set BAD^0, and coins R_A and $R_1, \ldots, R_n$, *we mean the process of* (i) *generating, for all players i and rounds r, the quantities*

- H_i^r, $M_{i\rightarrow}^r$, $M_{\rightarrow i}^r$, C_i^r, *and* R_i^r (*whose interpretation, as well as their setting for $r = 0$, is the same as in Definition* 2)

and the new quantities

- H_A^r (*a string called the* history of the adversary at round r),
- C_A^r (*a binary string called* the coin tosses of the adversary at round r),
- R_A^r (*an infinite subsequence of R_A called* the coin tosses of A after round r),

and

- BAD^r *and* GOOD^r (*two sets of players called, respectively, the* bad players at round r *and the* good players at round r, *such that* $\forall r$, $\text{GOOD}^r = [1,n] - \text{BAD}^r$)

[10] This is a clean but pessimistic approach (which makes our result stronger). In practice, though j may wish to fully collaborate with A by sharing all information he has, he may still have trouble in remembering—say—all previously received messages or all previously made coin tosses.

by setting $C_A^0 = \epsilon$ *and* $R_A^0 = R_A$ *and* (ii) *executing the following instructions for* $r = 1, 2, \ldots$:

0. $\text{TEMPH}_A^r := H_A^{r-1}$; $\text{TEMPR}_A^r := R_A^{r-1}$; $\text{TEMPGOOD}^r := \text{GOOD}^{r-1}$; $\text{TEMPBAD}^r := \text{BAD}^{r-1}$.
   "Because A's history, future coin tosses, and sets of good and bad players dynamically change within a round, we shall keep track of these changes in temporary variables. However, their final values within round r, respectively, H_A^r, R_A^r, GOOD^r, *and* BAD^r, *are unambiguously defined."*
1. *"Just as when all processors are honest,"* $\forall g \in \text{GOOD}^{r-1}$, *generate* $M_{g\rightarrow}^r$, *"the messages that g wishes to send in this round (which may be reset if g is corrupted in this round)," and* C_g^r, R_g^r *by running* P_g *on input* H_g^{r-1} *and coins* R_g^{r-1}.
2. $\forall g \in \text{GOOD}^{r-1}$ and $\forall b \in \text{BAD}^{r-1}$, $\text{TEMPH}_A^r := (\text{TEMPH}_A^r, g, b, M_{g\rightarrow}^r[b])$.
3. *Run A on input* TEMPH_A^r *and coins* TEMPR_A^r.
   If in this execution of step 3 *A has output* $j \in \text{TEMPGOOD}^r$ *and made the sequence of coin tosses C, then*
   - $\text{TEMPBAD}^r := \text{TEMPBAD}^r \cup \{j\}$, $\text{TEMPGOOD}^r := \text{TEMPGOOD}^r - \{j\}$,
   - $\text{TEMPH}_A^r := (\text{TEMPH}_A^r, H_j^{r-1}, C_j^r, C)$ *"so that from* H_j^{r-1} *and* C_j^r *A can reconstruct all of the messages that j wished to send in round r, and from* TEMPH_A^r *and C she can reconstruct why she has corrupted j,"*
   - $\text{TEMPR}_A^r := \text{TEMPR}_A^r/C$ *"adjust A's future coin tosses,"*
   - $\forall g \in \text{TEMPGOOD}^r$, $\text{TEMPH}_A^r := (\text{TEMPH}_A^r, g, j, M_{g\rightarrow}^r[j])$, *"i.e., according to rushing, A is also given the messages that the currently good players wish to send to j in this round," and*
   - *go to step* 3 *"to corrupt next processor."*

   Otherwise, if in this execution of step 3, *A has output,* $\forall b \in \text{TEMPBAD}^r$, *a vector* $M_b \in (\{0,1\}^*)^n$ *"as b's round-r messages"*[11] *and made the sequence of coin tosses C, then*
   - $\forall b \in \text{BAD}^r$, $M_{b\rightarrow}^r := M_b$,
   - $\text{TEMPH}_A^r := (\text{TEMPH}_A^r, C)$ *"so that she can reconstruct the bad players' messages of round r," and*
   - $\text{TEMPR}_A^r := \text{TEMPR}_A^r/C$, *and "adjust the final round-r quantities as follows."*
4. *Letting* $\mathcal{C}$ *be the sequence of coin tosses A has made since the last execution of step* 2,
   - $H_A^r := \text{TEMPH}_A^r$; $C_A^r := \mathcal{C}$; *and* $R_A^r := \text{TEMPR}_A^r$;
   - $\text{GOOD}^r := \text{TEMPGOOD}^r$ *and* $\text{BAD}^r := \text{TEMPBAD}^r$;
   - $\forall i, j \in [1, n]$, $M_{\rightarrow i}^r[j] := M_{j\rightarrow}^r[i]$;
   - $\forall g \in \text{GOOD}^r$, $H_g^r := (H_g^{r-1}, M_{\rightarrow g}^r, C_g^r)$;
   - $\forall b \in \text{BAD}^{r-1}$, $H_b^r := (H_b^{r-1}, bad)$, *and* $\forall b \in \text{BAD}^r - \text{BAD}^{r-1}$, $H_b^r := (H_b^{r-1}, C_b^r, bad)$.

Let E be the sequence (of tuples of quantities resulting from the above computation) so defined:

$$E = E_0, E_1, \ldots,$$

where

$$E_r = (H_1^r,\ M_{1\rightarrow}^r,\ M_{\rightarrow 1}^r,\ C_1^r,\ R_i^r,\ \ldots,\ H_n^r,\ M_{n\rightarrow}^r,\ M_{\rightarrow n}^r,\ C_n^r,\ R_n^r,\ H_A^r,\ C_A^r,\ R_A^r,\ \text{BAD}^r, \text{GOOD}^r).$$

[11] By convention, if A's output is not of this format, then it is assumed that $M_b = (\epsilon, \ldots, \epsilon)$ $\forall b \in \text{TEMPBAD}^r$.

We call E the execution of P with A on initial quantities H_A^0, BAD^0, and $p_1, \ldots, p_n$, and coins R_A and $R_1, \ldots, R_n$. *The value E_r is called* round r of E. *If R is a positive integer, by the expression E* up to round R, *in symbols $E_{[0,R]}$, we mean the finite subsequence $E_0, \ldots, E_R$.*

(Note: *The quantities H_i^r, $M_{i\rightarrow}^r$, $M_{\rightarrow i}^r$, C_i^r, R_i^r, H_A^r, C_A^r, R_A^r, BAD^r, and GOOD^r may carry an additional superscript or prefix to emphasize the protocol during the execution of which they have been generated.*)

Remark. The ability of an adversary to corrupt players at arbitrary points in time of a protocol is crucial in a randomized protocol. For a deterministic protocol, the adversary's optimal strategy may be calculated beforehand, but it may profitably change during the execution of a randomized protocol. For example, consider a probabilistic protocol for randomly selecting a "leader," that is, a processor to be put in charge of a given task. Depending on the specifics of the protocol, it may be impossible for the adversary to corrupt a few players beforehand and coordinate their actions so that one of them is guaranteed to be elected leader. It is, however, very easy for her to wait and see which processor is selected as leader and then corrupt it! (This feature models a "real-life" phenomenon: nobody is born a thief, but some may become thieves if the right circumstances arise .... To capture this realistic feature, we must allow—and successfully deal with—adversaries that can corrupt players, during run time, in a dynamic fashion.)

Fractional adversaries. Above we have presented the mechanics of executing a protocol with an adversary exhibiting what is essentially an arbitrarily malicious behavior. To keep things meaningful, however, we wish to put a cap on the *number* of players that an adversary may corrupt without otherwise limiting its actions in any way. In fact, we assume that no n-party adversary may corrupt n players in an execution with an n-party protocol (otherwise, no meaningful property about such an execution could possibly be guaranteed).

We will actually be focusing on adversaries that may corrupt at most a constant fraction of the players. Let c be a constant between 0 and 1; we say that an adversary A is a *c-adversary* if for all $n > 1$ and all n-party protocols P, in any execution with P on an initially bad set with $< cn$ elements, the cardinality of the bad set always remains $< cn$. Whenever we consider an execution of an n-party protocol with a c-adversary, we implicitly assume that the initially bad set contains less than cn players.

We also assume that no more than one adversary is active in an execution of a protocol. Actually, because the adversary that never corrupts any processor is a special type of adversary (indeed, a c-adversary for all possible $c \in (0, 1)$), we shall assume that in every execution of a protocol there is *exactly one adversary* active. Thus the expression "an execution of protocol P" really means "an execution of protocol P with adversary A, for some adversary A."

Initial quantities. As we have seen, to run an n-party protocol with an n-party adversary A, we need to specify, other than the coin tosses of A and the n players, the following initial quantities: (1) the initial adversarial history H_A^0, (2) the initially bad set BAD^0, and (3) the inputs $(p_1, \ldots, p_n)$. For the purpose of defining the mechanics of executing an n-party protocol with an n-adversary, we define $\mathcal{IQ}_n$, the set of *the initial quantities (of size n)* in a most "liberal" manner; that is, $\mathcal{IQ}_n = \{0,1\}^* \times 2^{\{1,\ldots,n\}} \times (\{0,1\}^*)^n$. In an accordingly liberal manner, we let $\mathcal{IQ} = \{\mathcal{IQ}_n : n > 1\}$ be the set of *all (possible) initial quantities.*

In general, however, it is meaningful to prove properties of protocols if the initial quantities of their executions satisfy a given constraint (e.g., reaching Byzantine

agreement on "the message" sent by a given member of a network is meaningful only if the identity of this sender is a common input to all processors in the network). We actually prefer to dismiss nonmeaningful initial quantities from consideration altogether. That is, we *define* each n-party protocol $P(n)$ together with the set of its own *proper* initial quantities, denoted by $\mathcal{IQ}_n^P$, on which—and solely on which—$P(n)$ can be run. Thus whenever we say that some specific values IQ are initial quantities for $P(n)$, it is assumed that $IQ \in \mathcal{IQ}_n^P$. Also, whenever we refer to an execution of a protocol P with some specific initial quantities IQ, if $IQ \in \mathcal{IQ}_n$, we actually refer to an execution of $P(n)$ on initial quantities IQ—quantities which actually belong to $\mathcal{IQ}_n^P$. (Indeed, it should be noticed that n can easily be computed from any member of $\mathcal{IQ}_n$.) In summary, all initial quantities of a protocol are deemed to be proper—and we shall use the expression "proper" only for emphasis.

Notice that by specifying the proper initial quantities of a given protocol, one could easily "cheat" by disallowing certain initial adversarial histories or initially bad sets so as to make protocol design artificially easy. In this paper, however, the proper initial quantities of a protocol will never in any way constrain the initial adversarial history or the initially bad set, except for its cardinality. Moreover, in this paper, proper initial quantities will never impose any restrictions on the inputs of the initially corrupted players. When defining a new protocol, though, we find it convenient to describe the generic element of its (proper) initial quantities by specifying (in particular) the inputs of all players, with the understanding that all constraints on the initially bad players must be dropped; that is, by saying that $(H_A^0, \text{BAD}^0, (p_1, \ldots, p_n)) \in \mathcal{IQ}_n^P$, we simply mean that the private input of player i is p_i if i does not belong to BAD^0. (In other words, if we wish a more extensive notation, an element of $\mathcal{IQ}_n^P$ is of the form $(H_A^0, \text{BAD}^0, \{(i, p_i) : i \notin \text{BAD}^0\})$.)

Random executions and probabilities.

DEFINITION 5. *Let n be an integer > 1, P be an n-party protocol, A be an n-party adversary, and $IQ \in \mathcal{IQ}_n^P$. By* randomly executing P with A on initial quantities IQ, *we mean the process consisting of generating the infinitely long bit sequences R_A, $R_1, \ldots, R_n$ by randomly and independently selecting each of their bits in $\{0,1\}$ and then executing P with A on initial quantities IQ and coins R_A, $R_1, \ldots, R_n$. We call the execution resulting from this process a* random execution *of P with A on initial quantities IQ.*

Thus the probability that an event e occurs in a random execution of P with A on initial quantities IQ is solely computed over the coin tosses of P and A. (Only if we have assumed a probability distribution on the private inputs as well—and if we explicitly say so—we may compute the probability of an event also over the random choices in selecting the private inputs.) The probabilities of events that are most important to us are those that are *intrinsic* properties of our protocols *alone*; that is, we shall prove bounds for these probabilities that are valid for any adversaries, any initial adversarial history, any initially corrupted players, and any players' inputs.

Fault tolerance. The fault tolerance of a protocol is essentially the highest fraction of faults it can tolerate.

DEFINITION 6. *Let Ψ be a property (i.e., a predicate) and c be a constant between 0 and 1. We say that a protocol P is a* c-fault tolerant *protocol (or a protocol* with fault tolerance c) with respect to Ψ *if $\Psi(E) = true$ for any execution E of P with a c-adversary.*

If the property Ψ is clear from context, we may simply say that P is a protocol *with fault tolerance* c rather than with fault tolerance c with respect to Ψ.

Legal shortcuts. For simplicity of discourse, we wish to "legalize" some handy

notation.

• *Highlighting something.* When we want to focus only on some of the quantities determining an execution, we just omit mentioning the others. For instance, the sentence "Let E be an execution of n-party protocol P with n-party adversary A on inputs $p_1, \ldots, p_n$ and initial corrupted set BAD^0" stands for "Let E be an execution of n-party protocol P with n-party adversary A on inputs $p_1, \ldots, p_n$, initially bad set BAD^0, initial adversarial history H_A^0, and coin tosses $R_1, \ldots, R_n$ and R_A, for some string H_A^0 and bit sequences $R_1, \ldots, R_n$ and R_A."

• *Matching types.* If P is an n-party protocol and we say that P is executed with an adversary A, we implicitly assume that A is an n-party adversary. Any adversary mentioned in the context of a protocol with fault tolerance c is meant to be a c-adversary. If P is a protocol and A is an adversary, by saying that n parties execute P with A, we mean that they execute $P(n)$ with adversary $A(n)$. By saying that a value IQ represents some initial quantities, we implicitly assume that $IQ \in \{0,1\}^* \times 2^{[1,n]} \times (\{0,1\}^*)^n$ for some positive integer n. By saying that a protocol P is executed with adversary A on initial quantities $IQ = (H_A^0, \text{BAD}^0, (p_1, \ldots, p_n))$, we mean executing $P(n)$ with $A(n)$ on initial adversarial history H_A^0, initially bad set BAD^0, and inputs $p_1, \ldots, p_n$. By an execution of protocol P with adversary A, we mean an execution of $P(n)$ with $A(n)$ for some number of players n (on some proper initial quantities).

Good, bad, and end. In an execution of a protocol, we say that processor i is *good at round* r if the adversary has not corrupted i at a round $\leq r$, and say that it is *bad at round* r otherwise. When, in an execution, the round under consideration is not specified, we say that a player i is *currently good* (respectively, *currently bad*) to mean that it is good at round r (respectively, bad at round r) if the round under consideration is r. We say that i is *eventually bad* in an execution if it is corrupted at some round of it, and say that it is *always good* otherwise. When no confusion can arise, we may use the simpler expression *good* (respectively, *bad*) instead of currently or always good (respectively, currently or eventually bad).

We say that an execution of a protocol *halts at round* r if r is the smallest integer s for which every good processor has halted in a round $\leq s$. (*Note:* If an execution of an n-party protocol Q has not halted at round r, it continues to be considered an execution of an n-party protocol after that round, whether or not some of the good processors have halted by round r and no longer execute the protocol.) Let R be a constant and P be a protocol; we say that P is an *R-round protocol* if in all executions of P every good processor halts at round R. (*Note:* In every execution of an R-round protocol, all good processors halt "simultaneously," but if an execution of a protocol which is not R-round halts at round R, the good processors may not halt in the same round of that execution.) We say that P is a *fixed-round* protocol if it is an R-round protocol for some value R. All of our protocols, except for the last one, are fixed-round. We say that a protocol *does not halt before round* r if in all its executions no good processor halts at a round $\leq r$.

Subprotocols. To facilitate the description of our Byzantine agreement protocol and to make it possible to use parts of it in other contexts, we have constructed it in a modular way. We thus need the notion of a *subprotocol*, that is, a protocol that is called as a subroutine by another protocol. Fortunately, in this paper, all subprotocols are fixed-round, they are called at rounds specified a priori by protocols (no execution of which halts by those rounds), and n-party protocols call only n-party subprotocols. (This simplifies our formalization somewhat; for instance, it makes it very clear when the call starts and when it ends.)

Let Q be a $> r$-round protocol calling an R-round protocol P at a prescribed round r. Then an execution of Q will be suspended once it reaches round r. At that point, the input value of each player i, p_i, is specified by either P itself (i.e., as when p_i is a constant) or player i's prior history, H_i^{r-1}. (If this is the case, we formally assume that there is a function $\mathcal{I}_i^P$ specified a priori, that, evaluated on H_i^{r-1}, determines p_i.) The execution of P on these inputs then starts. The good players execute P as if it were (rather than a subprotocol) "the first protocol they ever execute in their life," that is, their execution is independent of their prior histories. The adversary, on the other hand, is allowed to take advantage of what it has "learned" in the execution of Q to fine tune its strategy in the execution of P.[12] Moreover, should the adversary corrupt an additional player k during the execution of P, she will get, in addition to k's current history in the execution of P, its "suspended" history in Q. When P ends, each player appends its final "P-history" to its "suspended Q-history," and Q's computation is resumed. Processors corrupted in the execution of P are also considered corrupted in the resumed execution of Q.

DEFINITION 7. *Let n be an integer > 1, A be an adversary, P be an n-party R-round protocol, Q be an n-party protocol calling P at round r, and R_A^1, R_A^P, R_A^2, $R_1^{Q1}, \ldots, R_n^{Q1}$, $R_1^P, \ldots, R_n^P$, $R_1^{Q2}, \ldots, R_n^{Q2}$ be infinite binary sequences. By executing Q with A on initial quantities IQ and coins*

$$R_A^1, R_A^P, R_A^2, R_1^{Q1}, \ldots, R_n^{Q1}, R_1^P, \ldots, R_n^P, R_1^{Q2}, \ldots, R_n^{Q2},$$

we mean the following:

1. *To execute Q "a first time" with adversary A on initial quantities IQ and coins $R_A^1, R_1^{Q1}, \ldots, R_n^{Q1}$ "up to round r" so as to generate an execution up to round R, E^1, and thus quantities $H_A^{r,Q}$, $\text{BAD}^{r,Q}$, and $H_i^{r,Q}$ for each player i.*

2. (*For each player i, we let p_i be the input specified by $H_i^{r,Q}$.*) *To generate an execution up to round R, E^2, by running P with A on initial inputs $p_1, \ldots, p_n$, initial adversarial history $H_A^{r,Q}$, initially bad set $\text{BAD}^{r,Q}$, and coins $R_A^P, R_1^P, \ldots, R_n^P$ as usual except for the following. If A corrupts a new player j at a round x, it receives as an input not only the history of player j in the present execution, $H_j^{x,P}$, but also $H_j^{r,Q}$ ("j's suspended history in Q").*

3. *To generate an execution E^3 by running Q with A on initial adversarial history $H_A^{R,P}$, initially corrupted set $\text{BAD}^{R,P}$, and inputs $(H_1^{r,Q}, H_1^{R,P}), \ldots, (H_n^{r,Q}, H_n^{R,P})$.*

We let the corresponding execution (*of calling protocol Q with A on the above initial quantities and coins*) *consist of the sequence E whose first r elements are the elements of E^1, next R elements are those of E^2, and remaining elements are those of E^3.* (*In other words, the execution of a protocol Q that calls an R-round subprotocol P at round r is obtained by identifying, for each $\rho \in [1, R]$, round ρ of P with round $r + \rho$ of Q.*)

The notion of randomly executing a protocol and that of a random execution of a protocols are extended in the natural way to a protocol that calls a subprotocol at a prescribed round.

The notion of a subprotocol is immediately generalized to allow *nesting* of subprotocols, that is, to allow Q itself to be a subprotocol. Assume that for $1 \leq x < k$, protocol Q_x has called fixed-round protocol Q_{x+1} at round r_x. Then if protocol

[12] For instance, assume that a player j has been corrupted by A during the execution of Q before P was called. Then it is conceivable that from the Q-history of player j at the time of the call, the adversary may infer the Q-history at the time of the call of a good player i well enough to predict i's input to P. (That is, Q may induce some correlation among the inputs of subprotocol P that need not to be there if P were executed "from scratch.")

$Q = Q_k$ calls P at round r, all of the mechanics for calling P, executing P, and including the result of P's execution in the Q-histories remain the same, except that if a new processor i is corrupted during the execution of P, the "suspended history" of player i learned by the adversary, rather than simply being $H_i^{r,Q}$, is actually $(H_i^{r,Q_1}, \ldots, H_i^{r_k Q_k})$.

Concurrent protocols. Since it is the goal of this paper to squeeze as much computation as possible into a few rounds, we need to introduce the notion of concurrently executing more protocols, each protocol on its own inputs.

DEFINITION. *Let R be a positive integer, n be an integer > 1, and $\mathcal{L}$ be a finite set (of labels). Then an* (n-party R-round) concurrent protocol *is a mapping from $\mathcal{L}$ into the set of n-party R-round protocols.*

For each $x \in \mathcal{L}$, we denote by P^x the image of x under P and by P_i^x the program of player i within P^x, that is, $P^x = (P_1^x, \ldots, P_n^x)$.

In an execution of a concurrent protocol P, the good players execute each of the P^x's *independently* of the others. This restriction does not apply to the adversary, who can make use of the information learned in the execution of one of the protocols to choose her actions in the execution of another. Moreover, if the adversary corrupts player i at round ρ of the execution of a protocol P^x, then i becomes corrupted in the execution of every other protocol in P, but the total number of bad player increases only by one. Let us now be more precise.

DEFINITION 8. *Let R be a positive integer, n be an integer > 1, $\mathcal{L}$ be a finite set, $P : x \in \mathcal{L} \to P^x$ be an n-party R-round concurrent protocol, A be an n-party adversary, H_A^0 be a string, R_A be an infinite binary sequence,* BAD^0 *be a subset of* $[1, n]$, *$p_1^x, \ldots, p_n^x$ strings, and $R_1^x, \ldots, R_n^x$ be infinite binary sequences. By* executing P (*or, equivalently, by* concurrently executing $\forall x \in \mathcal{L}\ P^x$) with adversary A on initial adversarial history H_A^0, initially bad set BAD^0, inputs $p_1^x, \ldots, p_n^x$, and coins R_A and $R_1^x, \ldots, R_n^x$, *we mean performing the following instructions for each* player $i \in [1, n]$ *and each round* $r = 0, 1, \ldots$:

(a) *$\forall i \in \text{GOOD}^{r-1}$ and $\forall x \in \mathcal{L}$, compute $M_{\to i}^{r,P^x}$, $M_{i\to}^{r,P^x}$, C_i^{r,P^x}, and H_i^{r,P^x} from $M_{\to i}^{r-1,P^x}$, $M_{i\to}^{r-1,P^x}$, and H_i^{r-1,P^x} by running P_i^x so that the kth coin toss of P_i^x is the kth bit of R_i^x.*

(b) *Execute $H_A^r := H_A^{r-1}$, $\text{GOOD}^r := \text{GOOD}^{r-1}$, $\text{BAD}^r := \text{BAD}^{r-1}$, and $\forall g \in \text{GOOD}^r$, $\forall b \in \text{BAD}^r$, $\forall x \in \mathcal{L}$, $H_A^r := (H_A^r, M_{g\to}^{r,P^x}[b])$.*

(c) *Run A on input H_A^r so that A's kth coin toss is the k-bit of R_A. If C is the sequence of coin tosses made by A in this execution of step* 3, *then $H_A^r := (H_A^r, C)$. If A outputs $j \in \text{GOOD}^r$ in this execution of step* 3, *then $\text{BAD}^r := \text{BAD}^r \cup \{j\}$, $\text{GOOD}^r := \text{GOOD}^r - \{j\}$, and $\forall g \in \text{GOOD}^r$, $\forall x \in \mathcal{L}$, $H_A^r := (H_A^r, M_{g\to}^{r,P^x}[j])$, and go to step* (c). *Else "A has output for each bad player b and label x an n-message vector M_b^x."*

(d) *Letting C be the sequence of coin tosses made by A since the last execution of step* (b), *set $C_A^r = C$ and $\forall b \in \text{BAD}^r$, $\forall x \in \mathcal{L}$, $M_{b\to}^r = M_b^x$.*

The execution corresponding to the above process is $E = E_1, \ldots, E_R$, where

$$E_r = (H_1^r,\ M_{1\to}^r,\ M_{\to 1}^r,\ C_1^r, \ldots,\ H_n^r,\ M_{n\to}^r,\ M_{\to n}^r,\ C_n^r,\ H_A^r,\ C_A^r,\ \text{BAD}^r, \text{GOOD}^r),$$

where $H_i^r = \{(x, H_i^{r,x}) : x \in \mathcal{L}\}$, $M_{i\to}^r = \{(x, M_{i\to}^{r,x}) : x \in \mathcal{L}\}$, $M_{\to i}^r = \{(x, M_{\to i}^{r,x}) : x \in \mathcal{L}\}$, and $C_i^r = \{(x, C_i^{r,x}) : x \in \mathcal{L}\}$. That is, each quantity relative to protocol P^x is labeled with x.

The notions of randomly executing a concurrent protocol and that of a random execution of a concurrent protocol are obtained in the natural way.

A concurrent protocol $P : x \in \mathcal{L} \to P^x$ can be called at a prescribed round r by another protocol Q very much like an ordinary subprotocol. (In this case, the players

Q-histories at round r must specify, for each $x \in \mathcal{L}$, the inputs $p_1^x, \ldots, p_n^x$ on which to run protocol P^x; that is, $\forall x \in \mathcal{L}$, each Q_i^r specifies p_i^x.)

Sequenced protocols. We will also need the notion of a *sequenced protocol*. This consists of a pair of protocols (P, Q), where Q is run after P and on the histories of P. In this paper we actually need to consider only the case of sequenced protocols (P, Q), where P is R-round. Thus after an execution E of P, for all player i, i's input to Q consists of i's round-R history in E.

Like any other protocol, a sequenced protocol may be called as a subprotocol. Note that if (P, Q) is a sequenced protocol, then P and Q can never be executed concurrently. However, if $(P_1, Q_1), \ldots, (P_k, Q_k)$ are sequenced protocols, then it might be possible to concurrently execute $P_1, \ldots, P_k$ and then concurrently execute subprotocols $Q_1, \ldots, Q_n$, running each Q_i on the history of the execution of P_i.

Message bounds. As we have seen, at each round in the execution of a protocol $P = (P_1, \ldots, P_n)$, the adversary sends a message to each currently good player g, which then feeds it to P_g (among other inputs). Thus by sending g arbitrarily long messages, the adversary could arbitrarily increase the amount of g's local computation. To meaningfully discuss complexity issues, we thus need to modify the mechanics of protocol execution by introducing *message bounds*.

The message bound is a variable internal to each processor that at each round evaluates to a positive integer or to $+\infty$ a special value greater than all positive integers. If at round r the message bound of a currently good player g is set to a positive integer k, then g is allowed to compute the k-bit prefix of any incoming message at round r in k computational steps; only after this truncation will a message become part of the input to P_g.

A simple and flexible way to specify the message bounds of a player at every round is to give him a special input, the *message-bound input*: in any execution in which the value of this input is v, a player sets to v the message bound of every round. (With an eye to complexity, these special inputs will be presented in *unary*; in fact, because we charge v steps for extracting the v-bit prefix of a string, we do not wish this operation to be exponential in the message-bound input.) Alternatively, a protocol can specify the message bound of round r within the code of round r itself—and thus may set it to a lower value when shorter messages are expected (from the good players). In either way, if it wishes to keep its own running time under control, a protocol must set the message bounds of each round to finite values. (If it fails to do so in even a single round, it will be in that round that the adversary will send extremely long messages.)

Let us now see what happens to message bounds if a protocol P calls a subprotocol Q. If Q has message-bound inputs, then P calls Q, specifying the values of these inputs, as for all other inputs of Q. If Q sets its own message bounds as part of its code at each round, then it is enough for P to call Q. In either case, throughout the execution of Q, Q's message bounds are to be enforced; only after the call is over and the execution of P is resumed will P's message bounds become effective again.

Complexity measures. We now wish to discuss the two notions of round complexity and local complexity. In so doing, we focus directly on the two cases that are really relevant to this paper; that is, *constant* round complexity and *polynomially bounded* local computation. We leave to the reader—if she so desires—the task of generalizing these notions in meaningful ways.

DEFINITION 9. *Let P be a protocol with fault tolerance ϕ. We say that P* runs in an expected constant number of rounds *if there exists a positive constant d such that for all numbers of players n, for all ϕ-adversaries A, and for all proper initial*

quantities IQ, the expected number of rounds for a random execution of $P(n)$ with $A(n)$ on IQ to halt is d.

In measuring the amount of local computation in an execution of a protocol P with an adversary A, we count only the steps taken by the currently good players. (The adversary attacking the protocol can, of course, compute as much as it wants, but its steps do not contribute to the local computation of the protocol.) Recall that we have defined protocols to be uniform programs. Thus before running a protocol P in an n-size network, the players must first run P on input n so as to compute the exact n-tuple of programs, $P(n)$, that they should execute. (Player i will, in fact, execute the ith component of $P(n)$.) We thus also count as P's local computation the steps necessary for the players of an n-size network to compute $P(n)$.

Following the current tradition, we identify efficiency with polynomial-time computation, and we insist that our polynomial-time bounds hold for any possible adversary attacking the protocol.

DEFINITION 10. *Let P be a protocol with fault tolerance ϕ. We say that P* runs in (expected) polynomial time *if there exists a polynomial Q such that the following hold:*

1. *For all sufficiently large n, the (expected) number of steps for protocol P to output $P(n) = (P_1, \ldots, P_n)$ on input n is less than $Q(n)$.*

2. *For all integers n, for all ϕ-adversaries A, and for all initial quantities IQ, if L is the sum of the lengths of the inputs of the players outside the initially bad set, the (expected) number of protocol steps for a random execution of $P(n)$ with A on IQ to halt is less than $Q(n + L)$.*

(*Here by "protocol step" we mean any step executed by P_g for any currently good player g.*) [13]

Notice that to establish the local complexity of a given protocol P, we regard as an input the size n of the network in which P is run. This is indeed necessary because we consider the steps used to compute $P(n)$ as local computation, but it is also reasonable with respect to the rest of P's local computation. Indeed, even for protocols that have no inputs (and thus $L = 0$, as in the case of our protocol OC of section 7), we expect that when they are "really" executed among n players, at least n messages will be sent, which entails that the local computation is at least $O(n)$.[14]

Notice also that although we have not demanded that a protocol set its message bounds to finite values for the purpose of defining its local complexity, the amount of local computation of a protocol P can be small—or just bounded, for that matter—only if P sets proper message bounds.

4. Presentation and organization. We have chosen to build our Byzantine agreement algorithm in a modular way. We first introduce graded broadcast, a simple primitive weakly simulating the capability of broadcasting. We then use this primitive to build another one: graded verifiable secret sharing. Both primitives are of independent interest. Next, we present a technical construction from graded verifiable secret

[13] Note that the notion of polynomial time is convenient in that we should not worry too much about fine tuning the balance between the effort of computing $P(n)$ and that necessary to run the protocol, nor should we worry about whether the polynomial Q should be evaluated on $n + L$ or—say—the maximum between n and L, or n times the maximum length of the inputs of the initially good players. These specific choices would instead be crucial for defining that a protocol runs in a—say—quadratic amount of time. Similarly, a round complexity that is constant (and thus independent of all possible quantities affecting the computation) is "more or less uncontroversially defined;" however, the same cannot be said if the round complexity of a protocol were—say—quadratic.

[14] In any case, in a Byzantine agreement protocol each player has a single-bit input, and thus $L = O(n)$ in a network of size n.

sharing to a special protocol for collectively generating a special coin flip, that is, a bit that is both sufficiently random and sufficiently often visible by all good players. Finally, we show that Byzantine agreement is reducible to this special coin-flipping protocol.

Let us now discuss the additional choices we have made in presenting our protocols.

Proofs. Everything important becomes easy with time, and we believe that this will be the fate of adversarial computation. However, at this stage of its development, it is so easy to make mistakes that we have chosen to expand our proofs more than is legitimate and bearable in a more familiar setting. (Proofs are, after all, social processes and ought only to be convincing to a given set of researchers at a given point in time.) We have, however, consistently broken our proofs up into shorter claims so as to enable the reader to skip what she personally considers obvious.

Steps. As usual, we conceptually organize the computation of our protocols into *steps*. The primary reason for grouping certain instructions in a step is clarity of exposition. As a result, one step may require many rounds to be implemented, while another may require only one round.

In this paper, we adopt the convention of treating each step as a *subprotocol* in itself; that is, executing a step composed of certain instructions means calling a protocol consisting of those instructions. In view of our mechanism for subprotocol calling, a consequence of our convention is that each step starts being executed at a "new" round; that is, a step requires at least one round to be implemented.

The advantage of this convention is that we gain a more immediate correspondence between steps and rounds. For instance, the number of rounds of a protocol simply becomes the sum of the number of rounds of its steps; for another example, in our proofs, it will be quite easy upon encountering the expression "round r" to realize which is its corresponding step.

A (superficial) disadvantage of our convention is that our protocols "seem longer" since one round may be artificially added for each step. In fact, whenever the last round of a given step consists solely of internal computations of the processors, it can be merged in any practical implementation with the first round of the following step. This is no great loss, however, since we are not interested in claiming $O(1)$ improvements in the running times of our protocols.

Random selections. As we have seen, by saying that a player i flips a coin, we mean that he reads the next unread bit of a string R_i. The use of the expression "flips a coin" is justified by the fact that we will be focusing on random executions of our protocols, in which case, since each bit of R_i is independently and uniformly selected, all coin tosses of i are "genuine" and independent. In describing our protocols, however, we make use of additional suggestive language. By saying that i "randomly selects element e in a set S of cardinality k," we mean that the elements of the set are put in one-to-one correspondence with the integer interval $[0, k-1]$ and that the player i keeps on reading $\lceil \log k \rceil$ consecutive unread bits from string R_i until the "name" of an element in S is found.[15] Thus when executing an instruction of the type "$\forall y \in T$ randomly select $e_y \in S$"—where both T and S are finite sets—all of the resulting selections will be *random* (since no portion of R_i is skipped) and *independent* (since

[15] Thus the possibility that an execution *diverges* exists here, though we do not "protect" ourselves against such an event for two reasons. First, handling divergence properly would have translated into much heavier definitions and notations without adding much to the specific content of this paper. Second, we focus on random executions, and the probability of divergence in a random execution is 0.

no overlapping portions of R_i are ever used). This notation holds for adversaries as well.

Hiding message bounds. Only one of our protocols, *Gradecast*, makes use of message-bound inputs; all others specify their message bounds at each round in their codes. To lighten these codes, however, we omit making the message bounds explicit at any round in which they can be simply computed. For instance, if round $r-1$ consists, for all players, of the instruction

> "if predicate $\mathcal{P}$ is true, then send your name to all players; else send them the empty word ε,"

then for any good player, the message bound for round r is $\lceil \log n \rceil$, the maximum length of a player's name (assuming that the name of a player i is encoded by the binary representation of integer i).[16]

Sending and receiving. When processor j is instructed to send a value v to the processor i, we let v^* denote the value actually received by i since it can be different from v in case j is bad. It may happen that such a value v^* must itself be sent to other processors. In this case, we may write v^{**} for $(v^*)^*$.

It is implicitly understood that whenever a message easily recognizable as not being of the proper form is sent to a good processor, this interprets it as ε, the empty string. Any predicate of ε is defined to evaluate to false.

For any string σ, we let *distribute* σ denote the instruction of sending σ to every processor.

For every *nonempty* string σ, the expression *tally*(σ), which occurs in round $r+1$ of the code for player i of a given protocol, denotes the number of players that sent σ to i in round r. If $\mathcal{P}$ is a predicate, the notation $\mathcal{P}$(*tally*(x)) is shorthand for "there exists a nonempty string x such that $\mathcal{P}$(*tally*(x))."

Self and others. Variables internal to processor i will sometimes carry the subscript i to facilitate the comparison of internal variables of different processors.

Whenever processor i should perform an instruction for all j, this includes $j = i$. For example, when i sends a message to all players, he also sends a message to himself. A distinguished processor follows the code for all players in addition to his special code.

Math. We are concerned only with integral intervals. Thus if x and y are integers, the expression $[x, y]$ stands for the set of integers $\{i \,:\, x \le i \le y\}$.

If S is a set, we let S^2 stand for the Cartesian product of S and itself and 2^S stand for the set of subsets of S.

All logarithms in this paper are in base 2 (but we still use the natural base e for other purposes).

Genders. We will refer to a player as a "he" and an adversary as a "she."[17]

Protocols. To describe a protocol P, we just describe $P(n)$, leaving it to the reader to check that this code can be uniformly generated on input n.

Comments. We interleave the code of our protocols with clearly labeled comments. Often, we label short comments by writing them within quotation marks. In fact, in our protocols, all words within quotation marks are comments, not instructions.

Numberings. Definitions and claims that appear within the proof of a lemma or theorem are not expected to have interest—or even "meaning"—outside of their local

[16] Of course, our "English" protocols can be implemented in polynomial time only if a proper encoding is used. For instance, if for sending the name of a player i we chose to send a string consisting of 2^i 1's, some of our English protocols would not have a polynomial-time implementation. However, any "reasonable" encoding would do.

[17] This gender assignment has been made at random. (Moreover, any additional motive is no longer valid.)

context. To emphasize this, while allowing cross-referencing within the same proof of, say, Theorem X or Lemma Y, claims are given "local" labels, that is, TX-1, TX-2, ..., LY-1, LY-2, ... General definitions, lemmas, and theorems, however, have "global" labels that consist of progressive numbers.

QEDs. We end each claim's proof with the symbol "■" so that, if you find a claim obvious, you can easily locate the next ■ and continue reading after it. For lemmas and theorems, a proof's end (whether or not the proof is easy) is marked by the symbol "□" (per standard SIAM journal style).

5. Graded broadcasting.

5.1. The notion of graded broadcast. Broadcasting is a very useful feature in a network since a processor receiving a message is guaranteed that all other processors are receiving the *same* message.[18] Byzantine agreement provides the best "simulation" of broadcasting when all communication is person-to-person: each time a processor should broadcast a message, it simply sends it to all other processors, which then reach Byzantine agreement on its value. On our way towards fast Byzantine agreement, though, we first need to introduce a new primitive, *graded broadcasting*; this is a weaker simulation of broadcasting, but despite its simplicity, it is quite useful.

DEFINITION 11. *Let P be a protocol in which every player's input comprises the identity of a common distinguished processor, the* sender*, and a common unary string, the* message bound*, while the sender has an additional input string, the* sender's message*, whose length does not exceed the message bound.*[19]

We say that P is a graded broadcast protocol (with fault tolerance c) *if, for all c-adversaries A and for all initial quantities IQ, in every execution of P with A on IQ, each good player i outputs a pair $(value_i, grade_i)$—where $value_i$ is a string at most k-bit long and $grade_i$ equals 0, 1, or 2—so that the following three properties are satisfied:*

1. *If i and j are good players and $grade_i \neq 0 \neq grade_j$, then $value_i = value_j$.*
2. *If i and j are good players, then $|grade_i - grade_j| \leq 1$.*
3. *If the sender is good, then for every good player i, $grade_i = 2$ and $value_i =$ the sender's message.*

Remark. A simpler but weaker simulation of broadcasting was provided by crusader agreement, as introduced in [13] and refined by Turpin and Coan [37]. Basically, sending a message using crusader agreement is like sending a message on a broadcast channel in which some messages may not get delivered: recipients of the message are guaranteed that all other recipients receive the same messages. However, recipients have no guarantee that any other player receives the message. The advantage of graded broadcast over crusader agreement is that when a good player sends a message using graded broadcast, each good player can verify that all other good players receive the same message.

[18] The power of broadcasting is well exemplified by the fact that it allows one to reach Byzantine agreement in a trivial way: each processor (1) broadcasts his input value (the default value being assumed for those processors who do not broadcast anything) then (2) outputs the majority value broadcast (with, say, default 0 in case of a tie).

[19] That is, specifying the initial quantities in extensive notation, an element of $\mathcal{IQ}_n^P$ is a triplet of the form

$$(H_A^0,\ \text{BAD}^0, \{(h,k):\ i \in \text{BAD}^0 \cup \{h\}\}\ \cup\ \{(h,k,\sigma) : h \notin \text{BAD}^0\}),$$

where $H_A^0 \in \{0,1\}^*$, $\text{BAD}^0 \in 2^{[1,n]}$, $h \in [1,n]$, $k \in 1^*$, and σ is a string whose length is less than k.

5.2. A graded broadcast protocol.

PROTOCOL *Gradecast*(n)
Input for every player i: h, the identity of the sender, and k, the message bound.
Additional input for sender h: σ, the sender's message.
1. (for sender h): Distribute σ.
2. (for every player i): Distribute σ^*.
3. (for every player i): If $tally(z) \geq 2n/3$, distribute z; otherwise, send no messages.
4. (for every player i):
4.1. If $tally(x) \geq 2n/3$, output $(x, 2)$ and halt. Else:
4.2. If $tally(x) \geq n/3$, output $(x, 1)$ and halt. Else:
4.3. Output $(\varepsilon, 0)$ and halt.

THEOREM 1. *Gradecast is a four-round, polynomial-time, graded broadcast protocol with fault tolerance* 1/3.

Proof. That protocol *Gradecast* is four-round and polynomial-time is obvious. Before proving the remaining properties, let us establish the following simple claim.

CLAIM T1-1. *In any execution of Gradecast with message bound k, no good player sends a message longer than k.*

Proof. A good player may send messages only at steps 1, 2, and 3. If he sends a message at step 1, then he is the dealer, and the only message he sends in this step is his input string σ, which is guaranteed to be no longer than k. As for steps 2 and 3, what a good player distributes is a message that he has received at the start of the same step; thus, due to the message-bound mechanism, such a message is at most k-bit long. ∎

Let us now show that *Gradecast* is a graded broadcast protocol with fault tolerance 1/3. First, consider property 1. Let i be a player that is good throughout the entire execution of the protocol, and assume that $grade_i > 0$. Then because of message bounding, there must exist a nonempty string X, whose length is at most k, such that i computes $tally(X) \geq n/3$ in step 4. Thus i must receive X from at least a good player g at the start of round 4. Because X is at most k bits long and because of Claim T1-1, this implies that g had actually distributed X in round 3. In turn, this implies that g had received X as the round-2 message from at least $2n/3$ players. Letting w ($w < n/3$) of these players be bad, because of Claim T1-1, at least $2n/3 - w$ good players had thus distributed X in round 2. Therefore, at most $n/3$ good players could have distributed any value other than X in round 2. Thus for any k-bit string Y, $Y \neq X$, at most $n/3$ good players and hence $< 2n/3$ players overall could have sent Y to a good player in round 2. Thus no good player may have distributed Y in round 3, nor—because of Claim T1-1—may a good player have distributed a string Y' longer than k whose k-bit prefix coincides with Y. Hence for all good players, $tally(Y) < n/3$ at round 4 (which, in particular, implies that $value_i$ is uniquely determined). Now let j be another player, good until the end of the protocol, whose output has a positive grade component. Since this implies that there exists a string Z, whose length is at most k, such that j has received Z from at least $n/3$ players in step 4, and since we have just proved that Z cannot be different from X, it must be that in step 4 $tally(X) \geq n/3$ also for player j; that is, also $value_j = X$, which proves property 1.

Property 2 follows from the fact that if a good player i sets $grade_i = 2$, then he has received a k-bit string X from at least $2n/3$ players in round 4. Therefore, by Claim T1-1, at least $n/3$ good players distributed X in step 3; thus all good players must have received X at the start of round 4 from at least $n/3$ players, and thus all good players must decide according to 4.1 or 4.2.

Property 3 is easily verified since if h is good, all good players receive and distribute σ in rounds 2 and 3. □

Remarks.

• Protocol *Gradecast* is still a graded broadcast protocol with fault tolerance 1/3 when it is run on a network whose communication lines are not private (i.e., if the adversary can monitor the messages exchanged by the good players).

• If all message bounds were dropped from *Gradecast*, the resulting protocol would still satisfy properties 1, 2, and 3 of graded broadcast but would no longer be polynomial-time.

Theorem 1 guarantees that certain relationships hold among internal variables of good processors whenever protocol *Gradecast* is executed with a 1/3-adversary. These variables, being internal, are not observable by the adversary. The following simple lemma, however, guarantees that the adversary can infer them from her history—actually, from just a portion of her history, something that will be useful much later in this paper.

LEMMA 1. *For any given adversary A, any execution of* Gradecast *with A is computable from A's initial history and coin tosses after round* 0 *if the sender is bad at the start of the protocol, and from A's initial history and coin tosses after round* 0 *and the sender's message otherwise.*

Proof. As for any determinist protocol, an execution of *Gradecast* with an adversary is solely determined by (1) the inputs of the initially good players and (2) the adversary's history and coin tosses after round 0. Now, for protocol *Gradecast*, quantity (1) coincides with the sender's message if the sender is initially good, and is empty otherwise. □

The use of protocol *Gradecast* in our paper is so extensive that it is worth establishing a convenient notation.

Notation. After an execution of *Gradecast* in which player i is the sender, we use the following terminology:

• If a player j outputs a pair $(v, 2)$, we say that j *accepts* i's gradecast of v, or accepts v from i. If we do not wish to emphasize the value v, we may simply say that j accepts i's gradecast.

• If j outputs (v, x), for $x \geq 1$, we say that j *hears* i's gradecast of v, or hears v from i. If we do not want to emphasize the value v, we simply say that j hears i's gradecast.

• If j outputs $(v, 0)$ for some value v, we say that j *rejects* i's gradecast.

In what follows, we shall make extensive use of *Gradecast* as a subprotocol. It will thus be convenient to specify a call to *Gradecast* at step z of an n-party protocol in a compact way. In particular, since message bounds are necessary only for guaranteeing the polynomiality of *Gradecast*, it will be convenient to keep them in the background as much as possible. For instance, if we are guaranteed that, when executing *Gradecast* at step z of a given protocol, the sender's message is a single bit, we avoid explicitly specifying that *Gradecast* is called with message bound $k = 1$. More generally:

• If, given the possible choices for string σ, k is the least upper bound to the length of σ, then

$$z\text{: (for player } i\text{): gradecast } \sigma$$

means that step z consists of executing *Gradecast*(n) with sender i, sender's message σ, and message bound k.

• If, given the possible choices for the strings σ_i, k is the least upper bound to

their length, then

$$z\text{: (for every player } i\text{): gradecast } \sigma_i$$

means that step z consists of executing *Gradecast* concurrently n times, one for each label $i \in [1, n]$, so that in execution i the sender is i, his message is σ_i, and the message bound is k.

• If, given the possible choices for the strings σ_x, k is the least upper bound to their length, then

$$z\text{: (for player } i\text{): } \forall x \in S\text{, gradecast } \sigma_x$$

means that step z consists of executing *Gradecast* concurrently, once for each label $x \in S$, so that in execution x, the sender is i, his message is σ_x, and the message bound is k.

• If, given the possible choices for the strings σ_{xi}, k is the least upper bound to their length, then

$$z\text{: (for every player } i\text{): } \forall x \in S\text{, gradecast } \sigma_{xi}$$

means that step z consists of executing *Gradecast* concurrently, once for each label xi, where $x \in S$ and $i \in [1, n]$, so that in execution xi, the sender is i, his message is σ_{xi}, and the message bound is k.

Any of the above calls can be made dependent on whether a given property $\mathcal{P}$ holds, with the understanding that *if $\mathcal{P}$ is not true, then the gradecast still takes place, but the sender's message is the empty string.* For instance, if, given the possible choices for the strings σ_x, k is the least upper bound to their length, then

$$z\text{: (for player } i\text{): if } \mathcal{P}\text{, } \forall x \in S\text{, gradecast } \sigma_x$$

means that step z consists of executing *Gradecast* concurrently, once for each label $x \in S$, so that in execution x, the sender is i, the message bound is k, and the sender's message is σ_x if $\mathcal{P}$ evaluates to TRUE (in general, on x and i's current history) and ε otherwise.

Therefore, step z always consists of four rounds. Indeed, though a bit wasteful, the above convention is convenient to keep our protocols and subprotocols fixed-round.[20]

6. Graded verifiable secret sharing. We now need to adapt the earlier and powerful notion of verifiable secret sharing, developed for a different communication model, to the present scenario.

6.1. Verifiable secret sharing and collective coin flipping. The somewhat paradoxical concept of *verifiable secret sharing* (VSS for short) was introduced by Chor et al. [9], who also provided its first cryptographic implementation (tolerating $O(\log n)$ faults). Informally, a VSS protocol consists of two stages. In the first stage, a *dealer* "secretly commits" to a value of its choice. In the second stage, this value is recovered. The value is *secret* at the end of stage 1 in the sense that no subset of players of suitably small size can guess it better than at random, even if they exchange all of the information in their possession thus far (which good players never do in the first stage). The value is *committed* in stage 1 in the sense that a good player can verify that there exists a unique (and unknown) value x such that whenever stage 2

[20] By adopting more complex mechanics for subprotocol calling, we may interpret the above (conditioned) steps differently, and occasionally save rounds and messages.

is performed, with or without the help of the dealer and *no matter what the current or future bad players might do,* all of the good players will recover x. Moreover, this unique but unknown x is the value originally chosen by the dealer.

Verifiable secret sharing has by now found very sophisticated applications, [21] but we will be interested in the simpler, original application of [9]: enabling a group of players, a minority of which may be faulty, to generate a common and random bit. Informally, VSS allows such players to "collectively flip a coin" as follows. Each player privately selects his own random bit and secretly commits to it in stage 1 of a VSS protocol. When all have done so, all of these committed bits are recovered in stage 2 of the corresponding VSS protocol and the common, random bit is set to be the sum modulo 2 of all the decommitted bits.

Since we have already mentioned that the problem of Byzantine agreement is reducible to that of generating a common random bit, the possibility exists of using VSS for reaching Byzantine agreement. Indeed, as we shall see, we will use a special version of VSS (graded VSS) and a much more special version of the above coin-flipping algorithm (oblivious common coin) so as to produce a bit that is "common enough" and "random enough" to reach Byzantine agreement in constant expected time. Why don't we use ordinary VSS to collectively flip a coin in the straightforward way? The reason is simple: we want to use collective coin flipping for reaching fast Byzantine agreement in our point-to-point communication networks, but all implementations of VSS prior to our work either made use of *broadcasting* (an unavailable primitive in our networks!) or Byzantine agreement (which for us is a goal and not a tool!).

6.2. The notion of graded verifiable secret sharing. We now introduce a weaker version of VSS that is more easily obtainable on our networks without broadcasting. We call it *graded verifiable secret sharing* (graded VSS for short). Informally, this is a sequenced protocol with two components: *Graded Share-Verify*, which roughly corresponds to stage 1 of a VSS protocol, and *Graded Recover*, which roughly corresponds to stage 2. To properly define graded VSS, we need the notion of a an event becoming "fixed" at some point of the execution of a protocol.

DEFINITION 12. *Let $\mathcal{X}$ be an event that may occur only after round r in an execution of a protocol P, and let E be an execution of P. We say that $\mathcal{X}$ is* fixed at round r in E *if $\mathcal{X}$ occurs in every execution E' coinciding with E up to round r.*

DEFINITION 13. *Let P be a sequenced protocol, P = (Graded Share-Verify, Graded Recover), in which*

- *all players have a common input consisting of the identity of a distinguished processor, the* dealer*, and (the encoding of) a set of integers, called the* candidate-secret set;
- *the dealer has an additional input, called the* secret*, consisting of an element of the candidate-secret set*; *and*
- *each processor x is instructed to output a value verification$_x \in \{0, 1, 2\}$ at the end of Graded Share-Verify and an element of the candidate-secret set at the end of Graded Recover (if this latter component is ever executed on the history of the first one).*

We say that P is a graded verifiable secret sharing protocol with fault tolerance *c if the following four properties hold:*

1. Semiunanimity. *For all initial quantities (IQ), for all c-adversaries A, and for all executions of Graded Share-Verify with A on IQ, if a good player i outputs*

[21] For instance, since [26], it has become the crucial subroutine of all subsequent completeness theorems for protocols with honest majority, most notably those in [2], [3], [10], [21], and [35].

verification$_i = 2$, *then verification*$_j > 0$ *for all good players* j.

2. Acceptance of good secrets. *For all IQ c-adversaries A and for all executions of Graded Share-Verify with A on IQ, if the dealer is always good, then verification*$_i = 2$ *for all good players* i.

3. Verifiability. *For all on IQ c-adversaries A and for all executions E of Graded Share-Verify with A on IQ, if verification*$_i > 0$ *for a good player* i, *then there exists a value* σ *in the candidate-secret set such that the event that all good players output* σ *when executing Graded Recover* (*on their histories in E*) *is fixed at the end of E. Moreover, if the dealer is always good in E,* σ = *the secret.*

4. Unpredictability. *For all c-adversaries A, for all players h, for all integer m, and for all cardinality-m set S, if*

- *s is randomly chosen in S,*
- *Graded Share-Verify is randomly executed with A, dealer h, candidate-secret set S, and secret s, and*
- *dealer h is good throughout this execution, and the adversary outputs a value* $a \in S$ *(as her "guess" for the secret) at its end,*

then $\text{Prob}(a = s) = 1/m$.

Here the probability is taken not only over the coin tosses of P and A but also over the choice of s. [22]

Remarks.

- Notice that simply saying—in the verifiability condition—"all good players output σ in *Graded Recover*" is not sufficient for our purposes. In fact, although the adversary cannot prevent the good players from outputting the same value, this formulation still allows her to decide what the value of σ should be while executing *Graded Recover*. (An example of this has been constructed by the second author.) Thus *Graded Share-Verify* would not model a secret commitment as discussed above. For this we need the value of σ to be *fixed* at the end of *Graded Share-Verify* (when σ coincides with the dealer's secret and is totally unpredictable to the adversary if the dealer is currently good).
- A definition of VSS can be obtained from the above definition of graded VSS by replacing throughout "*verification*$_i = 2$" by "*verification*$_i > 0$." (The definition of VSS obtained in this way is actually, in our opinion, the most general and satisfactory one in the literature to date.) Similarly, jumping ahead, from our protocol *Graded-VSS*, one can easily derive a verifiable secret sharing protocol with broadcasting by essentially replacing all gradecast instructions with broadcast instructions. (It is the transformation of a verifiable secret sharing protocol with broadcasting to a graded

[22] An equivalent formulation of Unpredictability that does not require that the secret be chosen at random in the candidate secret set can be informally described as follows.

Let $PS(A, h, H_A^0, H_{-\{h\}}^0, S, s)$ denote the probability space over the final histories of A obtained by first randomly executing *Graded Share-Verify* with adversary A, dealer h, initial adversarial history H_A^0, initial histories (in a suitable encoding, of all initially good players other than dealer h) $H_{-\{h\}}^0$, candidate-secret set S, and secret s and then outputting the final adversarial history if dealer h has not been corrupted. Then unpredictability can be reformulated as follows:

4′. For all c-adversaries A, $\forall h$, $\forall H_A^0$, $\forall H_{-\{h\}}$, $\forall S$, and $\forall s_1, s_2 \in S$,

$$PS(A, h, H_A^0, H_{-\{h\}}^0, S, s_1) = PS(A, h, H_A^0, H_{-\{h\}}^0, S, s_2).$$

(The reason for including the histories of all players except the dealer is that we want to maintain unpredictability even when *Graded Share-Verify* is called as a subprotocol. In which case, though the prior histories of the players do not affect the execution of *Graded Share-Verify*, they will appear—for the corrupted players—in the final history of A.)

Personally, we find the above formulation (after properly "cleaning it up") generally preferable, but the one in the main text is in a more convenient form for the purposes of this paper.

VSS protocol without broadcasting that proves to be trickier.)

• Let P be a graded verifiable secret sharing protocol with fault tolerance c, $P = (GSV, GR)$. Then if there are too few players (i.e., if $\lfloor n \cdot c \rfloor = 0$), even a single player (over than the dealer) may at the end of an execution of GSV possess sufficient information to predict with probability 1 the dealer's secret. However, this does not contradict Unpredictability. Indeed, this property demands that no *adversary* can predict a good dealer's secret better than at random, and whenever $\lfloor n \cdot c \rfloor = 0$, she cannot corrupt any player. (The reader who perceives this phenomenon as awkward may prefer to define graded verifiable secret sharing protocols only when there are sufficiently many players. Personally, we prefer to define protocols so that any number of players greater than 1 is admissible, and we find it awkward to make exceptions for c-fault-tolerant protocols.)

6.3. A graded verifiable secret sharing protocol. This subsection is devoted to constructing the first graded VSS protocol. The basis of our construction was provided by an ingenious VSS protocol developed by Ben-Or, Goldwasser, and Wigderson [3]. Their protocol runs in $O(n)$ rounds—when there may be $O(n)$ faults—in a special type of communication network: the *standard-plus-broadcast* network. This is a network in which not only each pair of users communicate via their own private line, but all processors also share a broadcast channel.[23] We have adapted their protocol to our needs in two phases:

1. First, we have improved their result by providing a VSS protocol for standard-plus-broadcast networks, *FastVSS*, that (1) runs in a constant number of rounds and (2) is conceptually simpler.[24] (Ben-Or, Goldwasser, and Wigderson have told us that they have independently found a constant-round version of their result, but it is more complicated than ours.)

2. Second, we have transformed *FastVSS* (a protocol for standard-plus-broadcast networks) into *GradedVSS*, a constant-round graded VSS rotocol for *standard* networks (i.e., without any broadcasting facilities).

For the sake of conciseness, since the focus of this paper is on standard networks, we forgo providing an explicit description of *FastVSS*. (Below we present just the basic intuition behind it since this can effectively be used for *GradedVSS* as well.) Indeed, in our protocol *GradedVSS*, we have merged the above two steps into a single one. (The reader can, however, easily reconstruct the code of *FastVSS* from that of *GradedVSS*.) We will, however, provide separate intuition for each of the above two phases.

Phase 1: *FastVSS.* In *FastVSS*, the dealer encodes his own secret in a special and redundant way. Namely, if the adversary can corrupt at most t players, the dealer selects a bivariate polynomial $f(x, y)$ of degree t in each variable such that $f(0, 0)$ equals his secret. He then privately gives to player i the polynomials $P_i(y) = f(i, y)$ and $Q_i(x) = f(x, i)$ as his shares of the secret, an x-share and a y-share. As we shall see, the shares of any $\leq t$ players do not betray the secret at all. On the other hand, as expressed by the following lemma, any $t + 1$ genuine x-shares determine the secret (and the same is true for the y-shares).

[23] Actually, they share a bit more powerful means of communication: each recipient of a message m traveling along this special shared channel is guaranteed not only that all processors receive the same string m that he does but also that all processors know who the sender of m is.

[24] The protocol of [3] made use of Reed–Solomon codes, while our *FastVSS* does not rely on any error-correcting codes—or at least it succeeds in hiding them away while remaining self-contained in a very simple manner.

Our choice of encoding for the dealer's secret does not guarantee verifiability per se. In fact, a good player cannot check whether his received x-share is genuine or—say—a random polynomial of degree t. It is here that the y-shares come into play. In fact, *FastVSS* performs several checks centered around the following simple property: if two players i and j both hold genuine shares, then it should be that $P_i(j) = Q_j(i)$.

Unpredictability is guaranteed since *FastVSS* is constructed so that, in every check, the information about the secret of a good dealer obtainable by the adversary can be computed from the shares in her possession—which we have already claimed to be insufficient to predict the dealer's secret.

Phase 2: *From VSS to graded VSS.* Our transformation of *FastVSS* into *GradedVSS* possesses a somewhat general flavor: it appears to provide a compiler-type algorithm that, on input any *known* VSS protocol for standard-plus-broadcast networks, outputs a graded VSS protocol running in a standard network, with the same fault tolerance of the input protocol and with essentially the same time and number of rounds.[25] This simple transformation is thus potentially useful: one may be able to turn more efficient VSS protocols developed in the future into more efficient graded VSS protocols.

Quite intuitively, the essence of our transformation consists of replacing the broadcast instructions of the input VSS protocol by gradecast instructions and then of properly branching on the grade produced by each gradecast. As the expression "properly" indicates, however, some care is needed in deciding how to branch. Though some degree of freedom is available, it is crucial to exploit the fact that grades are 3-*valued*. It should be noticed that after a gradecast instruction of *GradedVSS*, we sometimes branch based on whether the gradecast is accepted or not (i.e., on whether the resulting grade is 2 or less than 2) and other times based on whether the gradecast is heard or not (i.e., on whether the resulting grade is ≥ 1 or equal to 0). Now, although some of these "accepted-or-not" branchings could be replaced by "heard-or-not" branchings (and vice versa), it can be shown that adopting only a single type of branching does not work. Carrying VSS from standard-plus-broadcast networks to standard ones without losing too much meaning is indeed the very reason that we have introduced our 3-valued graded broadcasting primitive.

(If going from VSS protocols to graded VSS protocols requires a minimum of attention, the "reverse" transformation is instead quite straightforward. Protocol *FastVSS* is in fact immediately obtainable from protocol *GradedVSS*.)

Before presenting protocol *GradedVSS*, let us state and prove a variant of the classic Lagrange interpolation theorem.

LEMMA 2. *Let p be a prime, t be a nonnegative integer, $x_1, \ldots, x_{t+1}$ be distinct elements in Z_p, and $Q_1(y), \ldots, Q_{t+1}(y)$ be polynomials* mod p *of degree t. Then there exists a unique bivariate polynomial $F(x, y)$ of degree t (in each of the variables x and y) such that*

$$(*) \qquad F(x_i, y) = Q_i(y) \quad \textit{for } i = 1, \ldots, t+1.$$

Proof. Define (Lagrange interpolation)

$$F(x, y) = \sum_{i=1}^{t+1} Q_i(y) \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)}.$$

[25] While our transformation works for all known VSS protocols, it is still conceivable that it cannot be applied to some future "bizarre" one.

Then $F(x, y)$ has degree t and satisfies $(*)$. We now argue that this polynomial is unique. Now assume that there exist two different t-degree bivariate polynomials $F_1(x, y)$ and $F_2(x, y)$ that satisfy $(*)$. We will prove that the polynomial

$$R(x, y) = F_1(x, y) - F_2(x, y) = \sum_{i,j} r_{ij} x^i y^j$$

is identically 0. For each $k = 1, \ldots, t + 1$, we have

$$\sum_{i,j=0}^{t} r_{ij} x_k^i y^i = R(x_k, y) \stackrel{\text{def.}}{=} F_1(x_k, y) - F_2(x_k, y) \stackrel{(*)}{=} Q_k(y) - Q_k(y) = 0.$$

That is, for each $k = 1, \ldots, t+1$, the polynomial in $y \sum_{j=0}^{t} (\sum_{i=0}^{t} r_{ij} x_k^i) y^j$ is identically 0. Thus for each fixed $\bar{j}$, $\sum_{i=0}^{t} r_{i\bar{j}} x_k^i = 0$ for $k = 1, \ldots, t + 1$, that is, the polynomial $\sum_{i=0}^{t} r_{i\bar{j}} x^i$ evaluates to 0 at the $t + 1$ points $x_1, \ldots, x_{t+1}$. This implies that $r_{i\bar{j}} = 0$ for all $i = 1, \ldots, t + 1$. Thus $R(x, y)$ is identically 0, which proves the uniqueness of $F(x, y)$. □

Notation for protocol GradedSV. In most of the scientific literature, the upper bound on the number of corruptable processors, denoted by t, is integral and explicitly given as an input to a fault-tolerant protocol. Having t as an input to each protocol would, however, be a bit cumbersome in our case: we have quite a few subprotocol calls and thus we would need to continuously specify the value of t for each call. Also, we are primarily interested in the highest possible value of t (i.e., $t = \lfloor (n - 1)/3 \rfloor$) and our protocols—with the singular exception of *GradedVSS*—do not become more efficient for smaller values of t. However, to allow the reader to appreciate how the efficiency of *GradedVSS* decreases with t, we set $t = \lfloor (n - 1)/3 \rfloor$ at its start (rather than making t an input to the protocol).

THEOREM 2. *GradedVSS is a graded verifiable secret sharing protocol with fault tolerance* 1/3 *that runs in expected polynomial time*; *GradedSV is a* 25*-round protocol, and GradedR is a two-round protocol.*

Proof. The claims about the number of rounds of *GradedSV* and *GradedR* are trivially verified. Equally simple to verify is the claim about the running time. (Recall that our choice of notation allows us to hide our message bounds.) The only difficulty that perhaps arises is about the computation of the prime p in step 1 of *GradedSV*(n). Actually, this prime can be found in deterministic polynomial time. In fact, for all sufficiently large k, there is a prime in the interval $[k, 2k]$. Thus, letting k be the maximum between n and m, we can in poly(k) time (and thus in time polynomial in n plus the total length of our inputs since $h < n$ and m is presented in unary) consider all integers in $[k, 2k]$ in increasing order until one is found that is proved prime by exaustive search of its divisors.

Let us now address the other claims.

Semiunanimity. If any good player G outputs $\mathit{verification}_G = 2$, he has received *recoverable* from at least $2t + 1$ players, of which at least $t + 1$ are good. Thus every other good player g has received *recoverable* from at least $t + 1$ players, so he outputs $\mathit{verification}_g \geq 1$.

Acceptance of good secrets. Let us first show that if the dealer is good (throughout *GradedSV*), then no good player gradecasts *badshare* in step 6. Since in this step a good player G can gradecast *badshare* only in three cases, let us show that none of them can occur.

Case 1. Assume that G has accepted the gradecast of *disagree*(j) from a player k in step 4. Then because of property 2 of any gradecast protocol, the dealer has heard

PROTOCOL *GradedSV*(n)

Input for every player i: h, the identity of the dealer, and m, a unary string encoding the candidate-secret interval $[0, m-1]$.

1. (for every player i): Compute p, the smallest prime greater than n and m, and set $t = \lfloor (n-1)/3 \rfloor$.
   Comment. All computations are done modulo p.
2. (for dealer h): Randomly select a t-degree bivariate polynomial $f(x, y)$ such that $f(0,0) = s$. "In other words, set $a_{00} = s$, for all $(i, j) \in [1, t]^2 - \{(0,0)\}$, randomly select a_{ij} in $[0, p-1]$, and set $f(x, y) = \sum_{i,j} a_{ij} x^i y^j$." For all i, privately send (P_i, Q_i) to player i, where $P_i = P_i(y) = f(i, y)$ and $Q_i = Q_i(x) = f(x, i)$.
   Comment. $f(0,0) = s$, your secret. $P_i(j) = Q_j(i)$ for all i and j.
3. (for every player i): For all j, if the dealer has not sent you a pair of t-degree polynomials mod p, send ε to player j; else, privately send j the value $Q_i^*(j)$.
4. (for every player i): For all j, if $P_i^*(j) \neq (Q_j^*(i))^*$, gradecast *disagree* (j).
   Comment. Either j or the dealer is bad or both are bad.
5. (for dealer h): For all $(i, j) \in [1, n]^2$, if you heard *disagree* (j) from player i in step 4, gradecast $(i, j, Q_j(i))$.
   Comment. i or j is bad or both are bad: what you reveal is already known to the adversary.
6. (for every player i): For all $(k, j) \in [1, n]^2$, if you accepted *disagree* (j) from player k in step 4 and
   - in the previous step you did not accept from the dealer exactly one value of the form (k, j, V), where $V \in [0, p-1]$; or
   - you accepted such a value and $k = i$, "i.e., you are player k," but $V \neq P_i^*(j)$; or
   - you accepted such a value and $j = i$, "i.e., you are player j," but $V \neq Q_i^*(k)$,

   gradecast *badshare*. "The dealer is bad."
7. (for dealer h): For all i, if you heard *badshare* from player i in step 6, gradecast $(i, P_i(y), Q_i(x))$.
   Comment. i is bad: what you reveal is already known to the adversary.
8. (for every player i): If
   (a) you gradecasted *badshare* in step 6; or
   (b) you accepted *badshare* from more than t players in step 6; or
   (c) for each player j whose gradecast of *badshare* in step 6 you have accepted, a step ago you did not accept from the dealer the gradecast of a value (j, U, V), where U and V are t-degree polynomials, or you accepted such a value but $Q_i^*(j) \neq U(i)$ or $P_i^*(j) \neq V(i)$,

   distribute *badshare*. "The dealer is bad."
9. (for each player i): If $tally(badshare) \leq t$, distribute *recoverable*.
10. (for each player i):
    If $tally(recoverable) > 2t$, output $verification_i = 2$.
    Comment. The secret is recoverable and all good players know it.
    Else: If $tally(recoverable) > t$, output $verification_i = 1$.
    Comment. You know that the secret is recoverable, but other good players may not know it.
    Else: Output $verification_i = 0$.
    Comment. The secret may or may not be recoverable.

PROTOCOL *GradedR*(n)

1. (for every player i): Distribute P_i^* and Q_i^*.
2. (for every player i):
   For each player j, set $P_j^i(y) = P_j^{**}$ and $Q_j^i(y) = Q_j^{**}$. If you have accepted *badshare* from j in step 6 of *GradedSV* and you have heard $(j, U(y), V(x))$ from the dealer in step 7, reset $P_j^i(y) = U(y)$ and $Q_j^i(x) = V(x)$.
   Comment. $P_j^i(y)$ and $Q_j^i(x)$ are your own view of player j's *final* shares.
   Let $count_i(j)$ consist of the number of players k for which $P_j^i(k) = Q_k^i(j)$.
   Comment. If a good player has output *verification* > 0 in *GradedSV*, all good players are given *count* $> 2t + 1$. However, a bad player may be given *count* $> 2t + 1$ by some good player and a low count by another good player.
   If possible, select a set of $t + 1$ players k such that $count_i(k) \geq 2t + 1$. Let $k_1, \ldots, k_{t+1}$ be the members of this set.
   Comment. If a good player has output *verification* > 0 in *GradedSV*, there will be such a set. In this case, although different good players may select different sets and some of these sets may contain bad players, each set determines the same bivariate polynomial
   Compute the unique bivariate polynomial $\mathcal{P}(x, y)$ such that $\mathcal{P}(k_j, y) = P_{k_j}^i(y)$ $j \in [1, t + 1]$ and output $\mathcal{P}(0, 0)$ mod m as the dealer's secret.

k's gradecast and has thus responded in step 5 by gradecasting $(k, j, Q_j(k))$. Since the dealer's gradecast is proper, it is necessarily accepted by G.

Case 2. If G gradecasts *disagree*(j) in step 4, the dealer accepts this proper gradecast and thus properly responds by gradecasting $(G, j, Q_j(G))$, and $Q_j(G)$ coincides with G's x-share, $P_G^*(y)$, evaluated at point j since for a good dealer $P_G^*(j) = P_G(j) = f(G, j) = Q_j(G)$.

Case 3. If G has accepted *disagree*(G) from k in step 4, the dealer has at least heard this value and has thus responded by properly gradecasting $(k, G, Q_G(k))$. This value is thus accepted by G; moreover, since the dealer is good, $Q_G^*(x) = Q_G(x)$ and thus $Q_G^*(k) = Q_G(k)$. Thus if the dealer is good, in no case does a good player G gradecast *badshare* in step 6. This implies that no good player can distribute *badshare* in step 8 according to conditions 8(a) or 8(b). Moreover, as long as the dealer continues to be good in step 7, a good player G cannot distribute *badshare* because of 8(c) as well. In fact, if G has accepted *badshare* from j in step 6, the dealer has at least heard this value and responded by properly gradecasting the polynomials $P_j(y)$ and $Q_j(x)$; since G accepts all proper gradecasts, and because when the dealer is good $P_j^*(i) = P_j(i) = Q_i(j) = Q_i^*(j)$ for all i and j, all of G's checks in steps 8 will be passed. We conclude that if the dealer is good in *GradedSV*, only the bad players may distribute *badshare* in step 8. Thus, since they are at most t in number, all good players G will distribute *recoverable* in Step 9 and output $verification_G = 2$ in step 10.

Verifiability. Let $\mathcal{S}$ be an execution of *GradedSV* in which at most $t < n/3$ players are corrupted and a good player outputs *verification* > 0, and let $\mathcal{R}$ be an execution of *GradedR* on the histories of $\mathcal{S}$. We need to show that (1) there exists a value $\sigma_{\mathcal{S}}$ such that the event that all good players output $\sigma_{\mathcal{S}}$ in $\mathcal{R}$ is fixed at the end of $\mathcal{S}$ and (2) $\sigma_{\mathcal{S}}$ coincides with the dealer's secret if he is always good in $\mathcal{S}$.

To this end, let us establish a convenient notation and a sequence of simple claims relative to $\mathcal{S}$ and $\mathcal{R}$. Recall that, since (*GradedVSS, GradedR*) is a sequenced protocol,

any player good in $\mathcal{R}$ (actually, in $\mathcal{R}$'s first round) is always good in $\mathcal{S}$.

LOCAL DEFINITION. *In an execution of GradedVSS, a player is said to be* satisfied *if he is good throughout the execution and does not distribute badshare in step* 8.

CLAIM T2-0. *In $\mathcal{S}$, there is a set of $t+1$ satisfied players.*

Proof. The proof is by contradiction. Were our claim false, then since there are at least $2t+1$ good players in $\mathcal{S}$, at least $t+1$ of them would have distributed *badshare* in step 8. Thus no good player would have distributed *recoverable* in step 9, and no good player would have output *verification*> 0 in step 10. ■

Due to condition 8(a), we also know that a satisfied player does not gradecast *badshare* in step 6 either; thus for all satisfied players i and j, $P_i^*(j) = Q_j^*(i)$. In view of Claim T2-0 and Lemma 2, we can thus present the following (local) definition.

LOCAL DEFINITION. *In execution $\mathcal{S}$, we let $\mathcal{G}_\mathcal{S}$ denote the lexicographically first set of $t+1$ satisfied players,*[26] *and we let $\mathcal{F}_\mathcal{S}$ denote the unique, bivariate, t-degree polynomial associated with $\mathcal{G}_\mathcal{S}$ by Lemma* 2; *that is, $\mathcal{F}(i,y) = P_i^*(y)\ \forall i \in \mathcal{G}_\mathcal{S}$.*

CLAIM T2-1. *$\forall i \in \mathcal{G}_\mathcal{S}$, $P_i^*(y) = \mathcal{F}_\mathcal{S}(i,y)$ and $Q_i^*(x) = \mathcal{F}_\mathcal{S}(x,i)$.*

Proof. Clearly, $P_i^*(y) = \mathcal{F}_\mathcal{S}(i,y)\ \forall i \in \mathcal{G}_\mathcal{S}$ by construction. We now prove the second set of equalities. Let $i \in \mathcal{G}_\mathcal{S}$; since we are dealing with polynomials of degree t, it is enough to prove that $\mathcal{F}_\mathcal{S}(x,i)$ equals $Q_i^*(x)$ at $t+1$ points. Indeed, for all $j \in \mathcal{G}_\mathcal{S}$, we have

$$\mathcal{F}_\mathcal{S}(j,i) \overset{\text{by construction}}{=} P_j^*(i) = Q_i^*(j),$$

where the last equality has been checked to hold by satisfied player j in step 4 since good player i sent him $Q_i^*(j)$ in step 3. ■

CLAIM T2-2. *Let G and i be good in $\mathcal{R}$, and let i also be satisfied in $\mathcal{S}$. Then $P_i^* = P_i^G$ and $Q_i^* = Q_i^G$.*

Proof. Informally, we must show that G's own view of i's final shares coincides with that of i himself. Being good in $\mathcal{R}$, i sends P_i^* and Q_i^* to G in step 1 of $\mathcal{R}$. Thus G sets $P_i^G = P_i^*$ and $Q_i^G = Q_i^*$ at the beginning of step 2 of $\mathcal{R}$. Finally, G does not reset these variables in the remainder of step 2. Indeed, he may reset these variables only under certain conditions, which include having accepted *badshare* from i in step 6 of *GradedSV*; however, no satisfied player gradecasts *badshare* in step 6. ■

CLAIM T2-3. *Let G and g be good in $\mathcal{R}$. Then $P_g^G(y) = \mathcal{F}_\mathcal{S}(g,y)$ and $Q_g^G(x) = \mathcal{F}_\mathcal{S}(x,g)$.*

Proof. We prove only the first equality since the second one is proved similarly. To this end, it is enough to show that $\mathcal{F}_\mathcal{S}(g,y)$ and $P_g^G(y)$ agree on every $i \in \mathcal{G}_\mathcal{S}$. Indeed, by Claim T2-3, we have

$$\forall i \in \mathcal{G}_\mathcal{S}, \quad \mathcal{F}_\mathcal{S}(g,i) = Q_i^*(g).$$

We now prove that

$$\forall i \in \mathcal{G}_\mathcal{S}, \quad Q_i^*(g) = P_g^G(i).$$

We break the proof of the above statement into two cases:

(a) G does not reset $P_g^G(y)$ (i.e., $P_g^G(y)$ coincides with the polynomial in y that g sent to G in step 1 of $\mathcal{R}$);

(b) G resets $P_g^G(y)$ (i.e., $P_g^G(y)$ is the polynomial in y gradecasted by the dealer in step 7 of $\mathcal{S}$).

[26] Any uniquely specified set of $t+1$ satisfied players in $\mathcal{S}$ would do.

Case (a). In this case, $P_g^G(y) = P_g^*(y)$. We must now argue that if $i \in \mathcal{G}_\mathcal{S}$, then $Q_i^*(g) = P_g^*(i)$. To begin with, notice that since i privately sent value $Q_i^*(g)$ to g in step 3 of $\mathcal{S}$, player g could compare these two values. We now show that $P_g^*(i) \neq Q_i^*(g)$ leads to a contradiction. In fact, if the latter inequality holds, g gradecasts *disagree*(i) in step 4 of $\mathcal{S}$. Since i accepts this gradecast, to remain satisfied he must accept $(g, i, Q_i^*(g))$ from the dealer in step 5 of $\mathcal{S}$. This causes player g to gradecast *badshare* in step 6 of $\mathcal{S}$—either because he does not accept the dealer's answer or because he accepts exactly the same answer that i does by property 1 of gradecast and thus we still have $Q_i^*(g) \neq P_g^*(i)$. Since i must accept g's gradecast of *badshare*, to keep him satisfied, the dealer must reply by gradecasting g's x-share and y-share in step 7 of $\mathcal{S}$ in order to have these values accepted by i. Thus these shares are at least heard by G, who thus resets P_g^G and Q_g^G. This contradicts the assumption that we are in Case (a).

Case (b). In this case, G must have first accepted the gradecast of *badshare* from g in step 6. Since g is good, this gradecast must have been accepted by good player i as well. Since i is satisfied, he must also have accepted the value replied by the dealer, $(g, U(y), V(x))$. By property 1 of gradecast, U and V are the same values heard by G and are thus G's own view of g's final shares; that is, $U(y) = P_g^G(y)$ and $V(x) = Q_g^G(x)$. The reason that now $Q_i^*(g) = P_g^G(i)$ is that i satisfactorily checked in step 8 that $Q_i^*(g) = U(i)$. ∎

Claim T2-4. *For all G good in $\mathcal{R}$ and for all k, $count_G(k) \geq 2t+1 \Rightarrow P_k^G(y) = \mathcal{F}_\mathcal{S}(k, y)$.*

Proof. Since $P_k^G(y)$ and $\mathcal{F}_\mathcal{S}(k, y)$ are t-degree univariate polynomials, it is sufficient to show that they agree on $t+1$ points. Indeed, if $count_G(k) \geq 2t+1$ and the bad players are at most t, there must exist $t+1$ good players g such that

$$P_k^G(g) \overset{\text{def. of } count_G}{=} Q_g^G(k) \overset{\text{Claim T2-3}}{=} \mathcal{F}_\mathcal{S}(k, g). \qquad \blacksquare$$

We are now ready to prove that in step 2 of $\mathcal{R}$, every good player G computes the bivariate polynomial $\mathcal{F}_\mathcal{S}$ and thus outputs $\mathcal{F}_\mathcal{S}(0,0) \bmod m$. To this end, first notice that there will be at least $t+1$ players k such that $count_G(k) \geq 2t+1$. In fact, for all good players g_1 and g_2, we have

$$P_{g_1}^G(g_2) \overset{\text{Claim T2-3}}{=} \mathcal{F}_\mathcal{S}(g_1, g_2) \overset{\text{Claim T2-3}}{=} Q_{g_2}^G(g_1).$$

Thus $count_G(g_1)$ will be at least $2t+1$, that is, at least the number of good players g_2; since there are at least $2t+1$ such players g_1, it will be possible for G to select $t+1$ players for which $count_G \geq 2t+1$. Let $k_1, \ldots, k_{t+1}$ be the ones he actually selects—some of them possibly bad. Then G outputs the polynomial $\mathcal{P}(x, y)$ such that

$$\forall l \in [1, \ldots, t+1], \quad \mathcal{P}(k_l, y) = P_{k_l}^G(y) \overset{\text{Claim T2-4}}{=} \mathcal{F}_\mathcal{S}(k_l, y).$$

Thus by Lemma 2, $\mathcal{P}$ and $\mathcal{F}_\mathcal{S}$ must be equal, and all good players output $\mathcal{F}_\mathcal{S}(0,0) \bmod m$ at the end of $\mathcal{R}$. Because $\mathcal{R}$ was just any execution of *GradedR* on the histories of $\mathcal{S}$, because $\mathcal{F}_\mathcal{S}$ is determined by execution $\mathcal{S}$, and because $\mathcal{F}_\mathcal{S}(0,0) \bmod m$ is guaranteed to belong to the candidate-secret interval $[0, m-1]$, this shows that the event that all good players in an execution of *GradedR* on the histories of $\mathcal{S}$ output $\sigma_\mathcal{S} = \mathcal{F}_\mathcal{S}(0,0) \bmod m$ is fixed at the end of $\mathcal{S}$.

Let us now show that if the dealer is good throughout $\mathcal{S}$, $\mathcal{F}_\mathcal{S}$ actually coincides with the polynomial $f(x, y)$ originally selected by the dealer in step 2 of $\mathcal{S}$. By

Lemma 2, it is enough to prove that there are $t+1$ distinct values v such that $\mathcal{F}_{\mathcal{S}}(v,y) = f(v,y)$. This is our case. In fact, letting G be a fixed good player, for any good player g, we have

$$\mathcal{F}_{\mathcal{S}}(g,y) \overset{\text{Claim T2-3}}{=} P_g^G(y) \overset{\text{Claim T2-2}}{=} P_g^*(y) \overset{\text{good dealer}}{=} P_g(y) \overset{\text{by def.}}{=} f(g,y).$$

Because a good dealer chooses $f(x,y)$ so that $f(0,0)$ coincides with his secret, which belongs to the candidate-secret interval $[0, m-1]$, whenever the dealer is good, the value output by the good players in $\mathcal{R}$, $\sigma_{\mathcal{S}} = \mathcal{F}_{\mathcal{S}}(0,0) \bmod m$, fixed at the end of $\mathcal{S}$, coincides with the dealer's secret.

Because $\mathcal{S}$ was just any execution of *GradedSV* in which $\mathit{verification}_i > 0$ for some good player i, this completes the proof that Verifiability holds for *GradedVSS*.

Unpredictability. An appealing, rigorous, and general proof of Unpredictability is obtainable utilizing the notion of "secure (or zero-knowledge) computation" [30]. As we have remarked, however, such a notion has not yet been published, and it is too difficult to be quickly summarized here. Therefore, we shall use an ad hoc and "quick-and-dirty" argument.

Our proof consists of showing that, in an execution of *GradedSV* in which the dealer is never corrupted, there exists a special piece of information (a string), independent of the secret, from which the adversary's history can be deterministically computed. Because the adversary cannot but predict the secret on the basis of her history (and of her coin tosses, which are clearly independent of the secret), this proves that the adversary cannot predict the secret better than at random.

The existence of such a special piece of information follows in part from the general mechanics of (any) protocol execution, and in part from the specific characteristics of our *GradedSV* protocol. We find it useful to present separately the following two parts of our argument: we present the first part in claims T2-5 and T2-6 and the second part in claim T2-8. Let us first establish some convenient notation.

Local definitions. Let P be a protocol, A be an adversary, E be an execution of P with A, and r be a round in E. Then:

- we say that a player is *eventually bad* in E if he is corrupted at some point of it and *always good* otherwise.
- we denote by $M_r^{AG \to EB}$ the set of messages (labeled with their senders and receivers) sent by the always good players to the eventually bad ones in round r.
- We refer to the quantities
  1. BAD^r
  2. the round-r history of A,
  3. the round-r histories of the eventually bad players,
  4. the coin tosses of A after round r, and
  5. the coin tosses of the eventually bad players after round r

  as the *final quantities of round r*. (Thus, the final quantities of round 0 include A's initial history.) If x is a step of P and r is x's last round, we refer to round r's final quantities as *the final quantities of step x*.

Claim T2-5. Given a protocol P and an adversary A in an execution of P with A, the final quantities of a round r ($r > 0$) are computable from the final quantities of round $r-1$ and $M_r^{AG \to EB}$.

Proof. The proof consists of recalling Definition 4 (i.e., how a protocol is executed with an adversary) and verifying that one can execute instructions 0–4 of Definition 4 to the extent necessary to compute the desired final quantities.

(In essence, A's history and coin tosses after each point of the execution of round r are computable given A's history and coin tosses after round $r-1$ (available by hypothesis), the history of each newly corrupted processor at the end of round $r-1$ (also available by hypothesis), and the message that each currently good processor g wishes to send to each currently bad processor b in round r. Now, if g is always good, such a message is among the available inputs, and if g is eventually bad, it can be computed by running P_g on g's history and g's future coin tosses at the end of round $r-1$, both of which belong to the available inputs. The history and coin tosses of each eventually bad player i after round r can be computed from the corresponding and available quantities after round $r-1$ by running P_i.) ■

Repeated application of Claim T2-5 immediately yields the following claim.

CLAIM T2-6. Given a protocol P and an adversary A, in an execution of P with A, A's final history is computable from the final quantities of round 0 and $\{M_r^{AG\to EB} : r = 1, 2, \ldots\}$.

Properties stronger than those of Claim T2-6 hold for our *Gradecast* and *GradedSV* protocols. Indeed, Lemma 1 implies that the final quantities of round 0 and only $M_0^{AG\to EB}$ suffice for reconstructing an entire execution of *Gradecast*. (Notice that in fact, in an execution of *Gradecast*, $M_0^{AG\to EB}$ coincides with the sender's message whenever the sender is always good.) As we show below for protocol *GradedSV*, the final quantities of round 0 and $M_1^{AG\to EB}$ suffice for reconstructing the final history of the adversary. (Notice that in an execution of *GradedSV*, $M_1^{AG\to EB}$ coincides with the x- and y-shares of the eventually bad players whenever the dealer is always good.)

CLAIM T2-7. *Given an adversary A, in an execution of* GradedSV *with A in which the dealer is good, A's final history is computable from the final quantities of round 0 and the x- and y-shares of the eventually bad players.*

Proof. Let us show how we compute (one by one) the final quantities of each step of *GradedSV* from the quantities available by hypothesis. For the reader's convenience, we recall (emphasizing it and omitting our comments) each step of *GradedSV*.

1. *(For every player i): Compute p, the smallest prime greater than n and m, and set $t = \lfloor (n-1)/3 \rfloor$.*

This step consists of a single round in which no good player sends any message (i.e., denoting by r the single round of this step, $M_r^{AG\to EB}$ is empty). Thus, as per Claim T2-5, we compute the final quantities of step 1 from the final quantities of round 0 alone.

2. *(For dealer h): Randomly select a degree-t bivariate polynomial $f(x, y)$ such that $f(0, 0) = s$. For all i, privately send (P_i, Q_i) to player i, where $P_i = P_i(y) = f(i, y)$ and $Q_i = Q_i(x) = f(x, i)$.*

This step consists of a single round. Denoting it by r, $M_r^{AG\to EB}$ coincides with the x- and y-shares of the eventually bad players (which are available by hypothesis). Thus, as per Claim T2-5, we can compute the final quantities of step 2 from the final quantities of step 1 and $M_r^{AG\to EB}$.

3. *(For every player i): For all j, if the dealer has not sent you a pair of t-degree polynomials mod p, send ε to player j; else, privately send j the value $Q_i^*(j)$.*

This step also consists of a single round. Denote it by r, and let i be an always good player and j be an eventually bad one. Because the dealer is good, the message sent by i to j is not ε but $Q_i^*(j) = Q_i(j)$. Although we do not know Q_i, we compute $Q_i^*(j)$ by evaluating polynomial P_j (which is available as the x-share of player j) at point i. Doing so for each always good player and each eventually bad one, we compute the entire $M_r^{AG\to EB}$. We compute the final quantities of step 3 from $M_r^{AG\to EB}$ and the final quantities of step 2, as per Claim T2-5.

4. *(For every player i): For all j, if $P_i^*(j) \neq (Q_j^*(i))^*$, gradecast* disagree(j).

Step 4 consists of four rounds in which n^2 simultaneous executions of *Gradecast*, properly labeled, take place.[27] Let us now show that for each execution of *Gradecast* in which the sender is (currently) good, we readily compute the sender's message. Let ij be the label of such an execution (and thus i its good sender). There are two mutually exclusive cases to consider: (a) j is a currently bad player and (b) j is currently good. In case (a) the message $(Q_j^*(i))^*$ sent by j to i in step 3 is contained in A's history of the previous round, and thus in the computed final quantities of step 3. In addition, because both the dealer and i are good, we know that $P_i^*(j) = P_i(j) = Q_j(i)$. Thus we compute $Q_j(i)$ by evaluating polynomial Q_j (which is available as the y-share of eventually bad player j) on input i. Consequently, we compute whether i's message in this execution of *Gradecast* is *disagree (j)*. In case (b) we know a priori that j has sent the proper quantity to i, and thus i will not gradecast *disagree (j)*. In either case, therefore, whenever the sender is good we compute the sender's message. Consequently, as per Lemma 1, we compute all executions of *gradecast* of step 4.

5. *(For dealer h): For all $(i, j) \in [1, n]^2$, if you heard* disagree (j) *from player i in step* 4*, gradecast* $(i, j, Q_j(i))$.

Because we have computed all executions of *Gradecast* of step 4, in particular we have computed, for each label (i, j), whether the dealer has heard *disagree (j)* from player i in execution (i, j). That is, we have computed whether execution (i, j) of *Gradecast* occurs. Moreover, whenever this is the case we can also compute the sender's message of execution (i, j). (The proof of this fact is similar to the corresponding fact of step 4; namely, if both i and j are good, then we know that no execution of *Gradecast* labeled ij occurs. Else, if i is good and j is bad, we compute the sender's message, $(i, j, Q_j(i))$, by evaluating polynomial Q_j—available as the y-share of bad player j—on input i.) Thus, as per Lemma 1, from these sender's messages and from the final quantities of step 4, we compute the final quantities of step 5 as well as the grades and values output by the good players in step 5.

6. *(For every player i): For all $(k, j) \in [1, n]^2$, if you accepted disagree* (j) *from player k in Step* 4 *and*
   - *in the previous step you did not accept from the dealer exactly one value of the form (k, j, V), where $V \in [0, p-1]$; or*
   - *you accepted such a value and $k = i$ but $V \neq P_i^*(j)$; or*
   - *you accepted such a value and $j = i$ but $V \neq Q_i^*(k)$,*

   gradecast badshare.

Because we have reconstructed the grades and values output by the good players in steps 4 and 5, we easily determine whether a good player gradecasts *badshare* in step 6. Thus, as per Lemma 1 and per the computed final quantities of step 5, we readily compute the final quantities of step 6 as well as all grades and values output in it by the currently good players.

7. *(For dealer h): For all i, if you heard* badshare *from player i in step* 6*, gradecast* $(i, P_i(y), Q_i(x))$.

Given the final quantities of step 6 and the grades and values output by the good players in step 6, we compute per Lemma 1 the final quantities of step 7 as well as the grades and values output by the good players of step 7.

8. *(For every player i): If*
   (a) *you gradecasted* badshare *in step* 6; *or*
   (b) *you accepted* badshare *from more than t players in step* 6; *or*

[27] Recall the notation established at the end of section 6.

(c) *for each player j whose gradecast of* badshare *in step* 6 *you have accepted, if a step ago you did not accept from the dealer the gradecast of a value (j, U, V)—where U and V are t-degree polynomials—or, if you accepted such a value but $Q_i^*(j) \neq U(i)$ or $P_i^*(j) \neq V(i)$,*

distribute badshare.

This step consists of a single round which we now denote by r. We then compute $M_r^{AG \to EB}$ by computing which good players distribute *badshare*. This determination is easily made from the computed senders' messages, grades, and outputs of step 6 and from the following three facts: (a) If both i and j are good, then i does not distribute *badshare*; (b) because the dealer is good, $Q_i^*(j) = Q_i(j)$ and $P_i^*(j) = P_i(j)$; and (c) if i is good and j is bad, then both $Q_i(j)$ and $P_i(j)$ are computable from the available x- and y-shares of bad player j. Thus, as per Claim T2-5 and per the final quantities of step 7, we compute the final quantities of step 8.

9. *(For each player i): If tally(badshare) $\leq t$ distribute* recoverable.

Note that we have just computed in step 8 which good players distribute *badshare*. Moreover, whether or not in that step a bad player sends *badshare* to a good player appears in A's history of step 8 (computed as part of the final quantities of step 8). Therefore, we compute *tally(badshare)* for each currently good player and thus determine which currently good players wish to distribute *recoverable* in step 9. Thus, as per Claim T2-5 and the final quantities of step 8, we compute the final quantities of step 9.

10. *(For each player i):*
If tally(recoverable) $> 2t$, output verification$_i = 2$.
Else, if tally(recoverable) $> t$, output verification$_i = 1$.
Else, Output verification$_i = 0$.

Since in this last step no good player sends any message, given just the final quantities of step 9, we compute, as per Claim T2-5, the final quantities of step 10.

Because the final history of the adversary is part of the final quantities of step 10, we have established our claim. ■

We can now easily finish the proof of Unpredictability. In Claim T2-6 we saw that A's final history in *GradedSV* depends solely on (1) the final quantities of round 0 and (2) the x- and y-shares of the eventually bad players. Now, in a random execution of *GradedSV* in which the dealer is always good and the secret s is randomly selected in $[0, m-1]$, the value of the secret is clearly independent of quantities (1). Thus, to prove that no strategy exists for the adversary to guess this secret with probability greater than $1/m$, it is sufficient to show that the dealer's secret is also independent of the x- and y-shares of the $t' \leq t < n/3$ players corrupted by A. Since we can modify any adversary so that she corrupts an additional $t - t'$ players just prior to finishing her last round of *GradedSV*, we can actually limit ourselves to prove our claim for the case $t' = t$. (In fact, if the adversary is such that the x- and y-shares of the first t' corrupted players are not independent of the dealer's secret, then by adding the shares of $t - t'$ other players we cannot obtain shares that are independent of the secret.) Thus, we now want to prove that for any choice of t eventually bad players, $b_1, \ldots, b_t$, any choice of $2t$ t-degree polynomials $P_{b_1}(y), Q_{b_1}(x), \ldots, P_{b_t}(y), Q_{b_t}(x)$, and any choice of secret s in $[0, m-1]$, there exists a unique bivariate polynomial $F(x, y)$ such that

(A) $F(b_i, y) = P_{b_i}(y) \ \ \forall i \in [1, t]$,
(B) $F(x, b_i) = Q_{b_i}(x) \ \ \forall i \in [1, t]$, and
(C) $F(0, 0) = s$.

We first show F's existence. Set $b_0 = 0$ and let $P_{b_0}(y)$ be the univariate, t-degree polynomial passing through the $t+1$ points $(0, s)$ and $(b_i, Q_{b_i}(0))$, $i \in [1, t]$. Then, by Lemma 2, there exists a unique bivariate polynomial F satisfying $F(b_i, y) = P_{b_i}(y)$ $\forall i \in [0, t]$. We now show that F enjoys three of the above required properties:

(A) By construction.

(B) Fix $a \in [1, t]$. We prove that $F(x, b_a) = Q_{b_a}(x)$ by showing that these two t-degree polynomials are equal at $t+1$ points. In fact, by construction we have $\forall j \in [1, \ldots, t]$ $F(b_j, b_a) = P_{b_j}(b_a) = Q_{b_a}(b_j)$, and $F(0, b_a) = P_0(b_a) = Q_{b_a}(0)$.

(C) By construction, $F(0, 0) = P_0(0) = s$.

The uniqueness of F is thus a consequence of the uniqueness of $P_0(y)$. This proves that "unpredictability" holds for *GradedSV* and thus completes the proof of Theorem 2. ∎

Remark. We have chosen the VSS protocol of [3] as the basis of our *GradedSV* protocol because it relied solely on private and broadcast channels but not on cryptography. (Several beautiful cryptographic VSS protocols are available, but our transformation would have yielded a cryptographic graded VSS protocol, and thus a Byzantine agreement algorithm tolerating only computationally bounded adversaries.) Another ingenious VSS protocol for standard networks that, *in addition*, possess broadcast channels, was found by Chaum, Crépeau, and Damgård [10]. We could have also adapted their protocol in our setting, but at the expenses of some additional complications since their protocol allows a—controllable but positive—probability of error.

Theorem 2 guarantees that whenever protocol *GradedSV* is executed with a 1/3-adversary, certain properties hold for the verification values of the good players. These values, however, are internal to the good players and not directly "observable" by the adversary at that point. The following simple lemma, however, shows that these values can be inferred from (a portion of) the adversarial history. We will make use of this result in the next section.

LEMMA 3. *At the end of any execution of* GradedSV*, the verification value output by each good player is computable from the final history of the adversary.*

Proof. If, in an execution of *GradedSV*, all players are good, then the dealer must have been good throughout the protocol. Thus, due to property 2 of verifiable secret sharing (acceptance of good secrets), we do know that every good player must output 2 as his own verification value. Now assume that one or more players are bad—including, possibly, the dealer. Then the verification value output by a good player i at the end of *GradedSV* is determined by the number of players that sent him *recoverable* in step 9. Now, the number of bad players that sent *recoverable* to i is immediately evident from the messages sent from bad players to good players (messages that are part of the final history of the adversary). Moreover, the number of good players that have sent *recoverable* to i is immediately computable from the messages sent from the good players to the bad players (messages that are also part of the final history of the adversary); in fact, a good player g sends *recoverable* to i if and only if he distributes it to all players, including the bad ones. □

7. Oblivious common coins. In this section, we want to show that processors of a network with private channels can exchange messages so that, in the presence of any 1/3-adversary, the outcome of a reasonably unpredictable coin toss becomes available to all good players. We start by defining what this means.

7.1. The notion of an oblivious common coin.

DEFINITION 14. *Let P be a fixed-round protocol in which each processor x has no input and is instructed to output a bit r_x. We say that P is an* oblivious common coin protocol (with fairness p and fault tolerance c) *iff for all bits b, for all c-adversaries A, and for all initial quantities IQ, in a random execution of P with A on IQ,*

$$\text{Prob}(\forall \text{ good players } i,\ r_i = b) \geq p.$$

We will refer to an execution of P as a coin; *by saying that this coin is* unanimously *b, we mean that $r_i = b$ for every good processor i.*

Remarks.

• Our notion of an oblivious coin is a strengthening of Dwork, Shmoys, and Stockmeyer's *persuasive coin* [16], which they implemented for at most $O(n/\log n)$ faults.

• We chose the term *oblivious* to emphasize that, at the end of the protocol, the good processors are "unaware" of whether the outcome of the reasonably unpredictable coin toss is "common." That is, by following the protocol, each good processor computes a bit, but it does not know whether the other good processors compute the same bit. We shall see how to successfully cope with this ambiguity in section 8, but let us first exhibit an oblivious common coin protocol with fault tolerance 1/3.

7.2. An oblivious common coin protocol.

LEMMA 4. *At the end of every execution of steps* 1—3 *of OC with a* 1/3 *adversary, for every good player i and every player j, whether $SUM_{ij} = bad$ can be computed from the final history of the adversary.*

Proof. The adversary's history at the end of step 3 of OC includes the adversary's history at the end of step 1. Thus, as per lemma 3, from the latter history we compute the value $verification_i^{hj}$ for all good players i and players j.

Consequently, as per Lemma 1, we compute each entire execution of *Gradecast* of step 2 of OC. From this information we readily compute, for each good player i and player j, whether conditions (3.a), (3.b), and (3.c) apply, and thus whether $player_{ij}$ (and consequently SUM_{ij}) equals *bad*. □

LEMMA 5. *In every execution of OC with a* 1/3*-adversary, for all good players g and G, $SUM_{gG} \neq bad$.*

Proof. This is so because of the following:

(a) By property 3 of graded broadcast, g accepts G's gradecast of G's confidence list.

(b) By the semiunanimity property of *GradedSV*, $|verification_g^{hj} - verification_G^{hj}| \leq 1$ for all labels hj.

(c) By the acceptance-of-good-secrets property of *GradedSV*, $verification_G^{hG} = 2$ for each of the $n - t$ good players h. □

LEMMA 6. *For all $n > 1$, for all executions of $OC(n)$ with a* 1/3*-adversary, and for all players j, there exists an integer $sum_j \in [0, n-1]$ such that for all good g, either $SUM_{gj} = sum_j$ or $SUM_{gj} = bad$.*

Proof. We distinguish three mutually exclusive cases.

Case 1: *Player j looks bad to all good players.* In this case, setting $sum_j = 1$ trivially satisfies our claim.

Case 2: *j looks okay to a single good player g and bad to all other good players.* In this case, choosing $sum_j = SUM_{gj}$ satisfies our claim for the following reasons. First, notice that SUM_{gj} is a well-defined integer value belonging to the interval

PROTOCOL $OC(n)$
Input for every player i: None.

1. (for every player i): For $j = 1 \ldots n$, randomly and independently choose a value $s_{ij} \in [0, n-1]$. "We will refer to s_{ij} as the ith secret assigned to j, or the secret assigned to j by i."
   Concurrently run *GradedSV* n^2 times, one for each label hj, $1 \le h, j \le n$. In execution hj, the candidate-secret set is $[0, n-1]$ "and thus the number of possible secrets equals the number of players," the dealer is h, and the secret is s_{hj}, "i.e., the dealer chooses s_{hj} to be his secret whenever he is good."
   Let $verification_i^{hj}$ be your output of execution hj, "that is, your own opinion about the existence/recoverability of s_{hj}."
2. (for every player i): Gradecast the value $(verification_i^{1i}, \ldots, verification_i^{ni})$. "This is your confidence list, that is, your own opinion about the existence/recoverability of each secret assigned to you."
3. (for every player i): for all j, if
   (a) in the last step, you have accepted j's gradecast of a vector $\vec{e}_j \in \{0,1,2\}^n$, "i.e., j's own confidence list—thus if j's is good, $\vec{e}_j = (verification_j^{1j}, \ldots, verification_j^{nj})$";
   (b) for all h, $|verification_i^{hj} - \vec{e}_j[h]| \le 1$, "that is, your opinion about the recoverability of every secret assigned to j differs by at most 1 from the opinion that j has gradecasted to you"; and
   (c) $\vec{e}_j[h] = 2$ for at least $n - t$ values of h,
   set $player_{ij} = ok$, "meaning that j looks okay to you;" otherwise, set $player_{ij} = bad$, "in which case j looks bad to you and he is bad."
4. (for every player i): "Recover all possible secrets:"
   Concurrently run *GradedR* on the n^2 histories of *GradedSV* that you generated in step 1, and denote by $value_i^{hj}$ your output for execution hj.
   If $player_{ij} = bad$, set $SUM_{ij} = bad$. Else: Set

$$SUM_{ij} = \left(\sum_{\substack{h \text{ such that} \\ \vec{e}_j[h]=2}} value_i^{hj} \right) \bmod n.$$

   "That is, if player j looks okay to i, SUM_{ij} equals the sum modulo n of all those secrets assigned to j that j *himself* thinks are optimally verified."
   If for some player j, $SUM_{ij} = 0$, output $r_i = 0$; otherwise, output $r_i = 1$.

$[0, n-1]$. This is so because each of the addenda contributing to SUM_{gj} is a well-defined integer (and thus taking the sum of these addenda mod n necessarily yields a value in $[0, n-1]$). Indeed, if $value_g^{hj}$ is an addendum of SUM_{gj}, then in the confidence list of j accepted by g, $\vec{e}_j[h] = 2$. Moreover, since j looks okay to g, step 3(b) ensures that $verification_g^{hj} > 0$. In turn, by the verifiability property of *GradedSV*, this guarantees that the corresponding secret is "well shared," that is, that the value output by g running *GradedR*, $value_g^{hj}$, belongs to the candidate-secret set $[0, n-1]$, as we wished to prove.

Case 3: *Player j looks okay to more than one good player.* Let g and G be any two such good players—thus $SUM_{gj} \neq bad \neq SUM_{Gj}$. To begin with, notice that the

value $\vec{e}_j$ is the same for both g and G due to property 1 of graded broadcast. We now show that $SUM_{gj} = SUM_{Gj}$. Indeed, we have

$$SUM_{Gj} = \left(\sum_{\substack{h \text{ such that} \\ \vec{e}_j[h]=2}} value_G^{hj} \right) \bmod n$$

and

$$SUM_{gj} = \left(\sum_{\substack{h \text{ such that} \\ \vec{e}_j[h]=2}} value_g^{hj} \right) \bmod n.$$

First, notice that the set of values h for which $\vec{e}_j[h] = 2$ are the same for both G and g—in fact, both g and G accepted j's gradecast of his own confidence list in step 2 and by virtue of property 1 of any graded broadcast protocol, their accepted lists are equal. Moreover, corresponding addenda in the two summations are equal. In fact, since j looks okay to G, $\vec{e}_j[h] = 2$ implies that $verification_G^{hj} > 0$, which in turn, due to the verifiability property of *GradedVSS*, implies that all good players will recover the same value as the secret of execution hj of *GradedSV*. □

LEMMA 7.[28] *Let $n > 1$ and let S and $\mathcal{G}$ be subsets of $[1, n]$. Let the set*

$$\mathcal{O} = \{\mathcal{O}_{gj} \in \{ok, bad\} : g \in \mathcal{G},\ j \in [1, n]\}$$

be such that for all $j \in S$, there exists $g \in \mathcal{G}$ such that $\mathcal{O}_{gj} = ok$. Then for all 1/3-adversaries A, in a random execution of $OC(n)$ with $A(n)$ in which $\mathcal{G}$ is the set of always good players and $\forall g \in \mathcal{G}\ \forall j \in [1, n]$ $player_{gi} = \mathcal{O}_{gj}$, the values $\{sum_j : j \in S\}$ are uniformly and independently distributed in $[0, n-1]$.

Before proving Lemma 7, let us consider a simpler but naïve argument. We have three good reasons for doing so: to use this naive argument as an introduction to our subsequent proof; to reassure the reader that our subsequent proof, though admittedly somewhat tedious, at least does not possess any obvious shortcuts; and to bring to light a subtle point that, unless it becomes known, may become a common as well as "fatal" logical trap in similar cryptographic contexts. For simplicity, let us state our naïve argument in a particularly simple case, that is, when S's cardinality equals 1, $S = \{1\}$. In this case, all we have to prove is that the unique sum_j is uniformly distributed in $[0, n-1]$.

Naïve argument: If $SUM_{gj} \neq bad$, then $sum_j = SUM_{gj} = \alpha + \beta \bmod n$, where

$$\alpha = \left(\sum_{\substack{h \text{ such that } h \text{ is good,} \\ verification_j^{hj}=2}} value_g^{hj} \right) \bmod n$$

[28] We condition the uniform and independent distribution of the sum_j's on a rather rich set of events. This is so because Lemma 7 will be invoked in rather diverse contexts, each with its own "conditioning," and we wish to make it very easy to see that it applies properly.

and

$$\beta = \left(\sum_{\substack{h \text{ such that } h \text{ is bad,} \\ verification_j^{hj}=2}} value_g^{hj} \right) \bmod n.$$

(These two values are well defined at the end of *GradedSV*, though no good player knows them because he does not know who else is good.) Value α is uniformly distributed in $[0, n-1]$ and is, in addition, unpredictable to the adversary at the end of step 1. Value β is controllable by the adversary (since each secret that contributes to it might have been chosen by the adversary via a processor h corrupted sufficiently early in step 1). Nonetheless, since at the end of *GradedSV* each of the secrets contributing to SUM_{gj} is fixed, so is β; that is, β is fixed at a point in which α is unpredictable. "Thus" no matter how much the adversary can control β, $\alpha+\beta$ is uniformly distributed in $[0, n-1]$.

Why is this naïve? The flaw in the above argument is that, in principle, the unpredictability of α may be consistent with the fact that, say, $\alpha + \beta$ always equals 0. Indeed, in principle, the adversary may be capable of guarateeing that $\beta = -\alpha \bmod n$ without knowing α nor (necessarily!) β.[29] This "magic correlation," though possible in principle, is actually impossible to achieve in our protocol due to many of its *specificities*, which have been ignored by the above reasoning. For instance, our protocol is such that the value β is actually "*known*" to the adversary at the end of step 1. This and other specificities are indeed an integral part of the following simulation-based argument, which properly corrects and formalizes the above naïve argument. The reader who, at this point, finds it obvious can proceed to Theorem 3.

Proof of Lemma 7. The proof is by induction on k, the cardinality of the set S. For $k = 0$, our statement is vacuously true. We now prove the inductive step by contradiction. Assume that our statement holds for $k-1$ but not for k. Then a simple averaging argument implies the following proposition.

PROPOSITION P1. *There exist an integer $n > 1$, a subset $\mathcal{G} \subset [1, n]$, a set of values $\mathcal{O} = \{\mathcal{O}_{gi} \in \{ok, bad\} : g \in \mathcal{G}\ i \in [1, n]\}$, a subset $S' \subset [1, n]$, whose cardinality is $k-1$, an additional player $j \notin S'$ such that $\forall i \in S' \cup \{j\} \exists g \in \mathcal{G}\ \mathcal{O}_{gi} = ok$, a set of $k-1$ values $\{v_i \in [0, n-1]\ :\ i \in S'\}$, an additional value $v \in [0, n-1]$, a distinguished*

[29] Mutatis mutandis, consider the following simpler scenario (simpler because it envisages computationally bounded players and thus the possibility of successfully using uniquely decodable encryptions) in which this is indeed the case. Two players desire to compute a common and random bit in the following manner. First, player 1 chooses a random bit b_1 and announces its encryption $E_1(b_1)$. Then player 2 chooses a random bit b_2 and announces its encryption $E_2(b_2)$. Then player 1 releases his own decryption key, d_1, and, finally, player 2 releases his own decryption key, d_2. This will enable both players to compute bits b_1 and b_2 and thus b, their sum modulo 2. Is such a b a random bit if, say, player 2 is bad? The answer is no. Player 2 may force bit b to be 0. Although he cannot predict b_1, he may exploit the fact that player 1 announces his encrypted bit first. (Recall that in our scenario, simultaneity is not guaranteed! Messages arrive by the next clock tick, but the adversary is allowed "rushing.") The strategy of player 2 is as follows. First, he announces the same ciphertext that player 1 does. Then he announces the same decryption key that player 1 does. This is a quite serious problem and does not have easy solutions. Simply requiring that the second player announce a different value than the one announced by the first is not a solution. However, discussion of this point is beyond the scope of this paper. (Let us just say that Micali devised a cryptographic protocol that enables two mutually distrusting people to announce independent values—but the protocol and its proof are not at all straightforward. Dolev, Dwork, and Naor [15] have provided a new type of public-key cryptosystem that would make easy to solve this and similar problems. Neither method, however, can be applied to the context of this paper, where the adversary is not restricted in the amount of computation she can perform and thus could break any public-key cryptosystem.)

player $G \in \mathcal{G}$ such that $\mathcal{O}_{Gj} = ok$, a constant $\varepsilon > 0$, a 1/3-adversary A, a string H, and a subset $B \subset [1,n]$, whose intersection with $\mathcal{G}$ is empty, such that, in a random execution of OC(n) *with $A(n)$ on initial adversarial history H and initially bad set B, letting "$\mathcal{G} = AG$" denote the event that the set of always good players coincides with $\mathcal{G}$ and defining*

$$\mathcal{X} = (\mathcal{G} = AG) \ \wedge \ (\forall i \in [1,n] \ \forall g \in \mathcal{G}, \ player_{gi} = \mathcal{O}_{gi}),$$

then

(a) $\text{Prob}(\mathcal{X}) > \varepsilon$;

(b) $\text{Prob}(\forall i \in S', \ sum_i = v_i \mid \mathcal{X}) = (1/n)^{k-1}$; *and*

(c) $\text{Prob}(\forall i \in S', \ sum_i = v_i \ \wedge \ sum_j = v \mid \mathcal{X}) > (1/n)^k$.[30]

CLAIM L7-0. *Let n, S', $\mathcal{G}$, $\mathcal{O}$, j, v, G, ϵ, A, H, B, and $\mathcal{X}$ be as in Proposition* P1*, and let $\mathcal{Y}$ be the event defined as follows:*

$$\mathcal{Y} = \mathcal{X} \ \wedge \ (\forall i \in S', \ sum_i = v_i).$$

Then in a random execution of steps 1–3 *of OC(n) with A on initial adversarial history H and initially bad set B in which G is not corrupted, the following holds:*

1. *$\mathcal{Y}$ occurs with positive probability.*

2. *Whether $\mathcal{Y}$ occurs is computable on the following inputs:* (2.1) *the set of the always good players,* (2.2) *the vectors $\vec{e}_x \in \{0,1,2\}^n$ accepted by at least one good player in step* 2*, and* (2.3) *for all good players g and for all players $x \neq j$, g's history of execution gx of GradedSV.*

3. *If $\mathcal{Y}$ occurs (and thus $G \in \mathcal{G}$ is good), the secret s_{Gj} (i.e., the secret—randomly selected in $[0, n-1]$—of execution Gj of GradedSV, where good G is the dealer) is predictable with probability $> 1/n$ on inputs* (2.1)*,* (2.2)*, and* (2.3) *above.*

Proof of Claim L7-0.1. First, notice that, because $G \in \mathcal{G}$, G is not corrupted whenever $\mathcal{X}$ occurs. Thus $\text{Prob}(\mathcal{Y} \mid G \text{ good}) = \text{Prob}(\mathcal{Y}) = \text{Prob}(\mathcal{Y} \mid \mathcal{X}) \cdot \text{Prob}(\mathcal{X})$. Now $\text{Prob}(\mathcal{X}) > \varepsilon$ by Proposition P1(a), and $\text{Prob}(\mathcal{Y} \mid \mathcal{X}) = (1/n)^{k-1}$ by Proposition P1(b).

Proof of Claim L7-0.2. Inputs (2.1) and (2.2) are by definition sufficient to determine whether $\mathcal{X}$ holds, and if this is the case, $\forall i \in S'$, $sum_i \neq bad$, and thus $sum_i = sum_{gi}$ for some good player g who has accepted i's gradecast of a vector $\vec{e}_i$ in step 3. Also, inputs (2.3) and (2.4) are more than sufficient to compute which actual value in $[0, n-1]$ sum_{gi} takes because this value depends only on $\vec{e}_i$ and g's history of execution hi of *GradedSV*, $h = 1, \ldots, n$, none of which coincides with execution Gj since $i \in S' and S' \not\ni j$.

Proof of Claim L7-0.3. If $\mathcal{Y}$ occurs, $SUM_{Gj} \neq bad$ and, on inputs (2.1), (2.2), and (2.3), one can compute all addenda that contribute to SUM_{Gj}, with the singular exception of $value_G^{Gj}$. Indeed, for each $h \neq G$ such that $\vec{e}_j[h] = 2$, the occurrence of $\mathcal{Y}$ implies that G is good, that $player_{Gj} = ok$, that $|verification_G^{hj} - \vec{e}_j[h]| \leq 1$, and thus that $verification_G^{hj} > 0$. In turn, $verification_G^{hj} > 0$ implies that the secret of each such execution hj is recoverable no matter what the currently bad players (and those which may become bad while running *GradedR*) may do. In particular, the secret of each such execution hj is recoverable if no more players are corrupted during *GradedR* and the bad players do not send any messages during *GradedR*. Thus when one has the histories of the currently good players (i.e., those in $\mathcal{G}$) at the end of each such

[30] Statement (c) is equivalent to the following statement:

($\tilde{c}$) $\text{Prob}(\forall i \in S, \ sum_i = v_i \ \wedge \ sum_j = v \mid \mathcal{X}) \neq (1/n)^k$.

In fact, (c) clearly implies ($\tilde{c}$) and the converse can again be established by averaging.

execution hj of *GradedSV*, one can run *GradedR* so as to reconstruct $value_G^{hj}$ for each of the above labels hj. Having done this, one can trivially compute the sum modulo n of these values; that is, one can compute

$$\omega = \left(\sum_{\substack{h \text{ such that } h \neq G \\ \vec{e}j[h]=2}} value_G^{hj} \right) \bmod n$$

and output $v - \omega$ as a prediction for $value_G^{Gj} = s_{Gj}$. We now show that $\text{Prob}(v - \omega = value_G^{Gj}) > 1/n$. Indeed, given the above notation, in Proposition P1 we can rewrite inequality (c) as follows:

(c) $\text{Prob}(sum_j = v \mid \mathcal{Y}) \cdot \text{Prob}(\mathcal{Y} \mid \mathcal{X}) > (1/n)^k$.

Thus, since $\text{Prob}(\mathcal{Y} \mid \mathcal{X}) = (1/n)^{k-1}$, Proposition P1(b) implies that

$$\text{Prob}(sum_j = v \mid \mathcal{Y}) > 1/n.$$

Now, whenever $\mathcal{Y}$ occurs, we have, in particular, $SUM_{Gj} \neq bad$, and thus $sum_j = value_G^{Gj} + \omega \bmod n$. Therefore, as we wanted,

$$\text{Prob}(value_G^{Gj} = v - \omega \mid \mathcal{Y}) > 1/n. \qquad \blacksquare$$

Notice that Claim L7-0 is not (yet!) a violation of the unpredictability of *GradedSV*.[31] To reach such a contradiction, we now show that (letting n, S', $\mathcal{G}$, $\mathcal{O}$, G, j, v, ϵ, A, H, B, and $\mathcal{X}$ be as in Claim L7-0 and Proposition P1) Claim L7-0 implies the existence of a 1/3-adversary for *GradedSV*, A' $(= A'_{n,S',\mathcal{G},\mathcal{O},j,v,G,\epsilon,A,H,B})$, that in a random execution with *GradedSV*(n) succeeds in achieving the following two goals. First, her random execution with *GradedSV*(n) coincides with execution Gj of *GradedSV* in a random execution of the first three steps of *OC*(n) with $A(n)$. Second, she possesses all inputs (2.1), (2.2), and (2.3) relative to said execution of *OC*(n) with $A(n)$.

Informal description of A' $(= A'_{n,S',\mathcal{G},\mathcal{O},G,j,\epsilon,A,H,B})$. Although adversaries A' and A are different and attack different protocols, they both act on networks with n players. Thus for any given player $i \in [1, n]$, we must specify at all points whether he is a player executing *GradedSV*(n) with A' or a player executing *OC*(n) with A. We find it convenient to do so by writing i' in the first case and i in the second.

Let us now describe the behavior of A' in a random execution, E', with *GradedSV* when the dealer is G' and both the adversarial history and the initially bad set are empty. During E', A' orchestrates and monitors portions of a "*virtual*" execution, E, of *OC*(n) with adversary A. (We thus think of adversary A' as acting in the actual network N'—where *GradedSV*(n) is executed—and of A as acting in the virtual network N where *OC*(u) is executed.)

Since we shall only consider n-party executions of protocols *GradedSV* and *OC* (where n is as in Proposition P1) in the proof of our lemma, we may more simply write *GradedSV* and *OC* instead of, respectively, *GradedSV*(n) and *OC*(n).

Adversary A' causes E to start by letting the adversarial history of A be H, the initially bad set be B (where H and B are as in Proposition P1), and the initial

[31] Indeed, for this to be the case, it is necessary that a 1/3-adversary succeed in predicting better than at random the random secret of a good dealer in a random execution between this adversary and *GradedSV*, that is, without assuming that such a random execution is embedded into an execution of *OC* for which certain key quantities are an available inputs.

histories of players $1, \ldots, n$ be those of a random execution of *OC*. As usual, the first 25 rounds of E consist of the concurrent execution n^2 times of the 25-round protocol *GradedSV*: one execution for each label hj (where h and j are player names and h is the dealer of execution hj).

For the first 25 rounds, adversary A' keeps E' in *lockstep* with E, *identifying* execution Gj (where G and j are as in Proposition P1) with E'. By "lockstep," we mean that, for each round $\rho > 0$, the round-ρ quantities of E' and E depend on and are generated after the round-$(\rho - 1)$ quantities of both E' and E. By "identifying" E', with execution Gj of *GradedSV* in E, we mean that A' corrupts processors in N' while interfering with the delivery of messages in N in the following way. Adversary A' corrupts player j' in network N' at round ρ if and only if A corrupts j in network N at round ρ. (Since the computation of A starts with initially bad set B, adversary A' corrupts j' in network N' at round 0 for all $j \in B$.) Let us now discuss how A' interferes with the delivery of messages in network N. At every round $\rho = 1, \ldots, 25$, after A has ended her corruption process and computed the messages from each bad player to each good one for all executions xy of *GradedSV*, A' acts as follows:

- For each execution $xy \neq Gj$, she delivers the proper messages to the proper recipients in network N.
- For execution Gj, if A outputs m as the message from bad player b to good player g, then A' has b' send m to g' in network N'; vice versa, if good player g' sends a message m' to bad player b' in execution E', then A' delivers m' as the message from g to b in round ρ of execution Gj.

As we shall prove, after making this description a bit more precise, the virtual execution E thusly generated is actually a "genuine" random execution of the first three steps of $OC(n)$ with A. Moreover, A' will be capable of computing inputs (2.1), (2.2), and (2.3) specified in Claim L7-0. This will enable her to contradict the unpredictability property of *GradedSV*.

More formal description of A' $(= A'_{n,S',\mathcal{G},\mathcal{O},G,j,\epsilon,A,H,B})$. Let us now describe a bit more precisely the way A' acts in a random execution of *GradedSV*(n) in network N', where the dealer is G', the candidate-secret set is $[0, n-1]$, the dealer's secret is randomly chosen in said candidate-secret set, the initial adversarial history equals the empty string, and the initially bad set is empty. We have already specified each player's version of *GradedSV* and the mechanics of an execution of an n-party protocol with an adversary. Thus, choosing the players' and adversary's coin tosses at random, our description of A' specifies the values taken by all possible players' quantities $H^r_{i'}$, $M^r_{i'\rightarrow}$, $M^r_{\rightarrow i'}$, $C^r_{i'}$, and $R^r_{i'}$ as well as the values taken by the adversarial quantities $H^r_{A'}$, $C^r_{A'}$, and $R^r_{A'}$ and by the sets BAD'^r and GOOD'^r.

In order to determine her actions in network N', adversary A' will construct only in part the quantities $H^{r,xy}_i$, $M^{r,xy}_{i\rightarrow}$, $M^{r,xy}_{\rightarrow i}$, $C^{r,xy}_i$, and $R^{r,xy}_i$, where $i \in [1, n]$ and $xy \in [1, n]^2$, but she will construct all of the possible quantities H^r_A, C^r_A, and R^r_A and the sets BAD^r and GOOD^r.

She generates these quantities with the same mechanics of a random execution of the first three steps of protocol $OC(n)$ with adversary A, initially bad set B, and initial adversarial history H. The quantities generated thusly by A', however, fall short of constituting such a random execution because they are *incomplete*. Indeed, they miss some Gj-labeled quantities—e.g., $H^{r,Gj}_i$ whenever $i \in \text{GOOD}^r$.[32]

[32] Since the quantities reconstructed by A' relative to network N do not quite constitute an execution of $OC(n)$ with A, and since it can be recognized that these quantities can be integrated so as to yield a virtual execution only after the entire behavior of A' has been described, it would be improper during our description to use suggestive expressions—such as "good at round r"—that,

It will be clear from our description, however, that if there is no r for which BAD^r contains G, then if one were to integrate these missing quantities with the corresponding quantities of E' (i.e., the execution of *GradedSV*(n) with A' in network N'), one would obtain a random execution of *OC*(n) with A, on initially bad set B and adversarial history H, in which G is not corrupted.

Notice that A' is active in network N' for 25 rounds because *GradedSV*(n) is a 25-round protocol. Notice also that the number of rounds in step 1 of *OC*(n) is also 25 if one imagines (as we do) that in each execution xy of *GradedSV*(n), the dealer randomly chooses the secret in round 0. Thus for $r = 1, \ldots, 25$ and each label $xy \in [1, n]^2$, the ith round of step 1 is the ith round of an execution of *GradedSV*. Indeed, for $r = 0, \ldots, 25$, adversary A' decides her action at round r in network N' "simultaneously" with her generation of round-r quantities in virtual network N (i.e., after having generated round-$(r-1)$ quantities in network N and before generating round-(r_1) quantities in network N).

To facilitate seeing that the round-r quantities generated by A' for virtual network N follow the mechanics of an execution of *OC*(n) with A on initially bad set B and adversarial history H, we break the instructions for this generation into instructions 1∗–4∗, thus matching the instructions 1–4 that we used in section 3 to describe how a protocol is executed with an adversary.

LOCAL DEFINITION. *Let $b_1, \ldots, b_k$ be the elements of subset B; denote by $\mathcal{L}$ the set of all execution labels of GradedSV in step* 1 *(i.e., $\mathcal{L} = \{hj : 1 \le h, j \le n\}$) and by $\mathcal{L}^-$ the set $\mathcal{L} - \{Gj\}$.*

Instructions for round $r = 0$.

(In virtual network N):

Set $H_A^0 = H$, $\text{BAD}^0 = B$, $\text{GOOD}^0 = [1, n] - \text{BAD}^0$, and $C_A^0 = \epsilon$. Then construct a binary string R_A by selecting randomly and independently each of its coins, and set $R_A^0 = R_A$.[33]

For all $xy \in \mathcal{L}^-$, randomly and independently select S_{xy} in $[0, n-1]$, and let C_{xy} denote the sequence of random bits used for this selection.

For all $xy \in \mathcal{L}^-$ and for all $i \in [1, n]$, construct an infinite bit string R_i^{xy} by choosing each of its bits randomly and independently. [34] Then reset $R_x^{xy} := C_{xy} \circ R_x^{xy}$.

Finally, for all $xy \in \mathcal{L}^-$ and for all $i \in [1, n]$, set $C_i^{0,xy} = \epsilon$, $R_i^{0,xy} = R_i^{xy}$, and $M_{i\rightarrow}^{0,xy} = M_{\rightarrow i}^{0,xy} = (\epsilon, \ldots, \epsilon)$, and, if $i \neq x$, $H_i^{0,xy} = ((x, n), M_{\rightarrow i}^{0,xy}, C_i^{0,xy})$—otherwise (i.e., $i = x$), set $H_i^{0,xy} = ((x, n, S_{xy}), M_{\rightarrow i}^{0,xy}, C_i^{0,xy})$.

"In an execution of *GradedSV*(n) with dealer x, the input for any player other than x is (x, m), that is, the name of the dealer and an encoding of the candidate-secret set, $[0, m-1]$. The private input for x is instead (x, m, s), that is, x is given his secret as an *additional* input, unrelated to his sequence of future coin tosses. (Therefore, should the dealer be corrupted by the adversary at round 1, she would discover his input secret but not the random choices made to come up with that secret, even if it were randomly selected.) In any execution xy of *GradedSV*(n) as a subprotocol of *OC*(n), there are two peculiarities. First, $m = n$ (which is easily reflected in the initial histories of the players in execution xy). Second, the dealer x of execution xy is not given his secret S_{xy} as an outside input; rather, he randomly chooses it in $[0, n-1]$

though very useful in building up intuition, presuppose that we are already dealing with a genuine execution. Notice, in fact, that all quantities relative to the virtual network N are constructed using only a syntactic description. Only in our comments do we use suggestive language.

[33] This is expressed thusly for convenience. In reality, each R_A will be constructed on an "as-needed basis."

[34] Again, in reality, each R_i^{xy} will be constructed on an "as-needed basis."

prior to calling *GradedSV*(n). Therefore, should the adversary corrupt player x at round 1 of her execution with $OC(n)$, then, she should be able to discover not only x's random secret relative to each execution xy but also the coin tosses that led x to choose S_{xy}. This is exactly what is accomplished by the above steps (which also accomplish giving these secrets suitable names—i.e., S_{xy}—and making it evident that all of them (for $xy \neq Gj$) are known to A')."

(In network N'):

Corrupt processors $b'_1, \ldots, b'_k$,

Instructions for $r = 1, \ldots, 25$.

0∗ (in virtual network N):

$\text{TEMPH}_A^r := H_A^{r-1}$; $\text{TEMPR}_A^r := R_A^{r-1}$; $\text{TEMPGOOD}^r := \text{GOOD}^{r-1}$; $\text{TEMPBAD}^r := \text{BAD}^{r-1}$.

1∗ (in virtual network N):

For all $g \in \text{GOOD}^{r-1}$ and for all $xy \in \mathcal{L}^-$, generate $M_{g\rightarrow}^{r,xy}$, "the messages g wishes to send in this round (which may be reset if g is corrupted in this round)," $C_g^{r,xy}$, and $R_g^{r,xy}$ by running $GradedSV(n)_g$ on input $H_g^{r-1,xy}$ and coins $R_g^{r-1,xy}$.

2∗ (in virtual network N):

For all $xy \in \mathcal{L}^-$, for all $g \in \text{GOOD}^{r-1}$, for all $b \in \text{BAD}^{r-1}$, $\text{TEMPH}_A^r := (\text{TEMPH}_A^r, g, b, xy, M_{g\rightarrow}^r[b])$.

"In other words, for each message m from a good player g to a bad player b computed by running *GradedSV* relative to label $xy \in \mathcal{L}^-$, deliver m to A as usual (i.e., specifying the name of the sender, the recipient, and the execution label). Then":

For each message m' received by a bad player b' from a good player g' in network N' at round r, $\text{TEMPH}_A^r := (\text{TEMPH}_A^r, g, b, Gj, m')$.

3∗ (in virtual network N):

Run A on input TEMPH_A^r and coins TEMPR_A^r.

(In network N' and in virtual network N):

If in this execution of step 3 A has output j "as the next player to corrupt," then HALT—"both the virtual execution in network N and the real execution in network N' are terminated"—and output a random value in $[0, n-1]$ as your guess for the secret of dealer G'. "You will be correct with probability $1/n$. Else":

(In virtual network N):

If A has output $q \in \text{TEMPGOOD}^r$ and made the sequence of coin tosses C, then

$\text{TEMPBAD}^r := \text{TEMPBAD}^r \cup \{q\}$, $\text{TEMPGOOD}^r := \text{TEMPGOOD}^r - \{q\}$,

$\text{TEMPH}_A^r := (\text{TEMPH}_A^r, xy, H_q^{r-1,xy}, C_q^{r,xy}, C)$,

$\text{TEMPR}_A^r := \text{TEMPR}_A^r / C$,

$\forall xy \in \mathcal{L}^-$, $\forall g \in \text{TEMPGOOD}^r$, $\text{TEMPH}_A^r := (\text{TEMPH}_A^r, g, q, xy, M_{g\rightarrow}^{r,xy}[q])$.

(In network N'):

Corrupt q' in network N', thereby learning his history $H_{q'}^{(r-1)}$ and coin tosses $C_{q'}^{(r-1)}$, as well as $M_{\rightarrow q'}^r[g']$ for all currently good players g',

"i.e., as well as each message sent by a currently good player g' to q'. Note that a player g' (respectively, b') is currently good (respectively, bad) in network N' if $g \in \text{TEMPGOOD}^r$ (respectively, $b \in \text{TEMPBAD}^r$) in the virtual network N."

(In virtual network N):

$\text{TEMPH}_A^r := (\text{TEMPH}_A^r, xy, H_{q'}^{r-1}, C_{q'}^{r-1}, C)$,

$\forall g \in \text{TEMPGOOD}^r$, $\forall b \in \text{TEMPBAD}^r$, $M_{g\rightarrow}^{r,Gj}[b] := M_{g'\rightarrow}^r[b']$.

Go to step 3∗ "to corrupt next processor."

If "A no longer wishes in this round to corrupt additional players," if in this execution of step 3 A has output, for all $xy \in \mathcal{L}$ and for all $b \in \text{TEMPBAD}^r$, a vector

$M_b^{xy} \in (\{0,1\}^*)^n$ "as b's round-r messages" and made the sequence of coin tosses C, then:

(In virtual network N):

$\forall xy \in \mathcal{L}^-$, $\forall b \in \text{BAD}^r$, $M_{b\rightarrow}^{r,xy} := M_b^{xy}$,

$\text{TEMPH}_A^r := (\text{TEMPH}_A^r, C)$ "so that she can reconstruct the bad players' messages of round r,"

$\text{TEMPR}_A^r := \text{TEMPR}_A^r / C$;

(In network N'):

$\forall b \in \text{BAD}^r$, $M_{b'\rightarrow}^r := M_b^{r,Gj}$.

"In other words, for each message m from a bad player b to a currently good player g in network N relative to execution Gj, have b' send m to g' as his round-r message in network N'."

4∗ (In virtual network N):

"Adjust the final round-r quantities as follows." Letting $\mathcal{C}$ be the sequence of coin tosses that A has made since the last execution of step 2,

$H_A^r := \text{TEMPH}_A^r$; $C_A^r := \mathcal{C}$; and $R_A^r := \text{TEMPR}_A^r$;

$\text{GOOD}^r := \text{TEMPGOOD}^r$ and $\text{BAD}^r := \text{TEMPBAD}^r$;

$\forall xy \in \mathcal{L}^-$, $\forall k, i \in [1,n]$, $M_{\rightarrow i}^{r,xy}[k] := M_{k\rightarrow}^{r,xy}[i]$; and

$\forall k \in [1,n]$, $\forall i \in \text{BAD}^r$, $M_{\rightarrow i}^{r,Gj}[k] := M_{k\rightarrow}^{r,Gj}[i]$;

"Adversary A' does not know the messages exchanged among good processors in network N' and thus does not construct the corresponding messages in execution Gj."

$\forall xy \in \mathcal{L}^-$, $\forall g \in \text{GOOD}^r$, $H_g^{r,xy} := (H_g^{r-1,xy}, M_{\rightarrow g}^{r,xy}, C_g^{r,xy})$;

"A' does not know the histories of the good players in N' and thus does not construct $H_g^{r,Gj}$."

$\forall xy \in \mathcal{L}$, $\forall b \in \text{BAD}^{r-1}$, $H_b^{r-1,xy} := (H_b^{r-1,xy}, bad)$, and $\forall b \in \text{BAD}^r - \text{BAD}^{r-1}$, $H_b^{r,xy} := (H_b^{r-1,xy}, C_b^{r,xy}, bad)$.

"Thus far, each time that a new round was added to the partial virtual execution of OC with A, the execution of *GradedSV* with A' also progressed one round. At this point, however, the rounds added to the partial virtual execution only allow A' to make additional internal computations and, possibly, additional corruptions, but her execution with *GradedSV* remains at round 25.

At the start of the execution of step 2 of $OC(n)$, the prior history of a good player consists of an n^2-vector, $\{H_g^{25,xy} : xy \in \mathcal{L}\}$. But at this point of the computation in the virtual network N, there is no quantity $H_g^{25,Gj}$. However, there is a quantity $H_g^{25,xy}$ whenever $g \in \text{GOOD}^{25}$ and $xy \in \mathcal{L}^-$. Such a quantity $H_g^{25,xy}$ specifies the quantity $\mathit{verification}_i^{xy}$ (via some proper input function $\mathcal{I}_g^{steps2-3}$). Thus for all $xy \in \mathcal{L}^-$ and for all $g \in \text{GOOD}^{25}$, $\mathit{verification}_i^{xy}$ is computable by A' because she has already computed $H_g^{25,xy}$."

Additional instructions for round 25.

LOCAL DEFINITION. *We denote by steps* 2–3 *the protocol consisting of steps* 2 *and* 3 *of* $OC(n)$.

For all $g \in \text{GOOD}^{25}$, set $\mathit{verification}_g^{Gj}$ to be the verification value output at round 25 by player g' in the execution of *GradedSV*(n) with you in network N'.

"Although these values are internal to good processors of network N' and thus invisible to you, you can compute them—by virtue of Lemma 3—from your knowledge of the sets of good and bad players and the messages exchanged between good and bad players."

For all $g \in \text{GOOD}^{25}$, set $H_g^{25,Gj} = rc_g^{Gj}$, where rc_g^{Gj} is a reserved character that (via the input function $\mathcal{I}_g^{steps2-3}$) specifies $verification_g^{Gj}$.

Execute subprotocol *steps* 2–3 with A (with the initial adversarial history being the computed quantity H_A^{25}, the initially bad set being the computed quantity BAD^{25}, and the prior history of each good player g being the ("thusly completed") vector $\{H_g^{25,xy} : xy \in \mathcal{L}\}$), handling corruptions as follows.

"In this execution of *Gradecast* as a subprotocol, you know the sender, the sender's message, the set of initially bad players, the active adversary, and the initial adversarial history. Thus you do not need to know the players' prior histories exactly in order to exactly reconstruct all messages exchanged up to the next corruption. Indeed, the good players do not rely on their prior histories (more than is needed to figure out which message to gradecast). Once a corruption occurs, however, in order to update the adversarial history in a proper manner, you need the corrupted player's prior history."

Whenever A corrupts an additional player k, corrupt k' in network N' so as to find his current history, H'_k. In the current *steps* 2–3 history of k, replace the reserved character rc_k^{Gj} by the string H'_k and deliver the thusly updated history to A (in the syntactically proper manner).

Instructions for predicting the secret.

"If you have not already predicted the secret of G' at random, do the following":

Detect whether event $\mathcal{Y}$ "of Claim L7-0" occurs.

"You can do that by virtue of Claim L7-0 because you may compute all inputs (2.1), (2.2), and (2.3) envisaged in that claim."

If $\mathcal{Y}$ has not occurred, then predict the secret of G' by outputting a random number in $[0, n-1]$.

Else output the value $v - \omega \bmod n$ as your prediction of the secret of G'. "v is the value of Proposition P1 and Claim L7-0 and

$$\omega = \left(\sum_{\substack{h \text{ such that } h \neq G \\ verification_j^{hj} = 2}} value_G^{hj}, \right) \bmod n."$$

This ends our description of A'. Notice that Steps 2 and 3 of OC take, respectively, four rounds and one round. Thus the subprotocol consisting of Steps 1–3 of OC is a 30-round protocol.

Let us now more precisely claim (without proof) that if one replaces the missing quantities constructed by A' for the virtual network N with the "corresponding" quantities that arise in the execution of adversary A' with $GradedSV(n)$ in network N', one obtains a random execution of Steps 1—3 of $OC(n)$ with A.

CLAIM L7-1. *Let E' be a random execution of protocol $GradedSV(n)$ with adversary A' in which the initially bad set is empty, the initial adversarial history is the empty string, the dealer is G', the candidate-secret set is $[0, n-1]$, and the secret is randomly selected in $[0, n-1]$.*

- *For all $r = 0, \ldots, 25$, for all $xy \in \mathcal{L}^-$, and for all $i \in [1, n]$, let $H_i^{r,xy}$, $M_{i\rightarrow}^{r,xy}[j]$, $M_{\rightarrow i}^{r,xy}[j]$, $C_i^{r,xy}$, GOOD^r, BAD^r, H_A^r, and C_A^r be the network-N quantities generated by A' during E'.*
- *For all $r = 1, \ldots, 25$ and for all $g \in \text{GOOD}^r$, let $H_{g'}^r$, $M_{g'\rightarrow}^r$, $M_{\rightarrow g'}^r$, and $C_{g'}^r$, be, respectively, the round-r history, messages sent, messages received, and coin tosses of player g' in E'.*

• *For all* $r = 0, \ldots, 25$ *and for all* $g \in \text{GOOD}^r$, *set* $H_g^{r,Gj} = H_{g'}^r$, $M_{g\rightarrow}^{r,Gj} = M_{g'\rightarrow}^r$, $M_{\rightarrow g}^{r,Gj} = M_{\rightarrow g'}^r$, *and* $C_g^{r,Gj} = C_{g'}^r$.[35]

• *For all* $r = 1, \ldots, 25$ *and for all* $xy \in \mathcal{L}$, *set* $H_i^r = \{H_i^{r,xy} : xy \in \mathcal{L}\}$, $M_{i\rightarrow}^r = \{M_{i\rightarrow}^{r,xy} : xy \in \mathcal{L}\}$, $M_{\rightarrow i}^r = \{M_{\rightarrow i}^{r,xy} : xy \in \mathcal{L}\}$, *and* $C_i^r = \{C_i^{r,xy} : xy \in \mathcal{L}\}$.

• *For all* $r = 26, \ldots, 30$ *and for all* $i \in [1, n]$ *let* H_i^r, $M_{i\rightarrow}^r$, $M_{\rightarrow i}^r$, C_i^r, GOOD^r, BAD^r, H_A^r, C_A^r, *and* R_A^r *be the network-*N *quantities generated by* A' *when executing protocol steps* 2–3 *with* A *during* E'.

Then the sequence of tuples

$$E = E_0, \ldots, E_{30},$$

where

$$E_r = (H_1^r,\ M_{1\rightarrow}^r,\ M_{\rightarrow 1}^r,\ C_1^r,\ R_i^r, \ldots,$$
$$H_n^r,\ M_{n\rightarrow}^r,\ M_{\rightarrow n}^r,\ C_n^r,\ R_n^r,\ H_A^r,\ C_A^r,\ R_A^r,\ \text{BAD}^r, \text{GOOD}^r),$$

is a random execution (*up to round* 30) *of protocol* $OC(n)$ *with adversary* A, *on initially bad set* B *and initial adversarial history* H, *in which player* G *is not corrupted.*

The above claim follows from our description of A' and by the observation that all n^2 secrets of E have been randomly and independently selected in $[0, n-1]$: secrets s_{xy} for $xy \in \mathcal{L}$ by construction, and secret s_{Gj} (i.e., the secret of G' in execution E' of *GradedSV*(n) with A') by hypothesis.

Notice that A' is a 1/3-adversary (because A is a 1/3-adversary and because A' corrupts a player in E' if and only if A corrupts the corresponding player in E), that A' may compute all inputs (2.1), (2.2), and (2.3) envisaged in Claim L7-0, and that A' never corrupts dealer G in an execution with *GradedSV*(n) when the initially bad set is empty and the initial adversarial history is the empty string.

Let us now show that A' can predict the secret of G' with probability $> 1/n$. Indeed, whenever A wishes to corrupt G, A' halts, outputting a random number in $[0, n-1]$ as her prediction of the secret of G'. Thus she will be correct with probability $1/n$ in these cases. If A does not corrupt G in E, but neither does event $\mathcal{Y}$ (which A' detects), then A' again guesses the secret of G' at random and is right with probability $1/n$. However, whenever $\mathcal{Y}$ occurs, which by virtue of Claims L7-0 and L7-1 is with positive probability, then A' (per Claim L7-0) correctly guesses the secret of G' with probability $> 1/n$. Finally, because the event that A does not corrupt G occurs whenever $\mathcal{Y}$ occurs, we have that A' correctly guesses the secret of G' with probability $> 1/n$. This contradiction of the unpredictability property of *GradedSV* establishes Lemma 7. □

THEOREM 3. *OC is an expected-polynomial-time* 32*-round oblivious common coin protocol with fairness* $> .35$ *and fault tolerance* 1/3.

Proof. The claims regarding round complexity and running time are easy to verify. (Recall that—though in a "hidden" way—we do make use of message bounds.) Let us thus prove the other claims. We start with some convenient notation.

Let A be a 1/3-adversary and $\mathcal{IQ} \in \mathcal{IQ}_n^{OC}$ be proper initial quantities for OC. Then in an execution of $OC(n)$ with a 1/3-adversary on $\mathcal{IQ}$, let C_0 denote the event that the coin is unanimously 0, C_0^{good} denote the event that $sum_g = 0$ for some good player g (i.e., the coin is unanimously 0 "thanks to a good player"), and C_1 denote the event that the coin is unanimously 1. Correspondingly, let P_0, P_0^{good}, and P_1 denote

[35] Notice that some of these quantities might have already been computed by A' for virtual network N, in which case they would be reset to the same value.

the probabilities of C_0, C_0^{good}, and C_1, respectively, in a random execution of $OC(n)$ with $A(n)$ on $\mathcal{IQ}$. (Notice that $C_0 \neq \neg C_1$, where $\neg E$ denotes the complement of an event E.)

We are now ready to lower-bound both P_0 and P_1.

Lower-bounding P_0. Since $P_0 \geq P_0^{good}$, we lower-bound P_0 by lower-bounding P_0^{good}.[36] If S is a subset of $[1,n]$, let "$AG = S$" denote the event that the set of the always good players coincides with S. Notice that if $\text{Prob}(AG = S) > 0$ then $|S| > 2n/3$ and that $\sum_{|S|>2n/3} \text{Prob}(AG = S) = 1$. (In fact, in any execution with a 1/3-adversary, there must be at least $2n/3$ always good players.) Also notice that (because for all good players G and g, $SUM_{Gg} \neq bad$ by Lemma 5) Lemma 7 implies that whenever $AG = S$ occurs, the values $\{sum_g : g \in S\}$ are independently and uniformly distributed in $[0, n-1]$. Thus, because it is sufficient that any such value equals 0 for the coin to be unanimously 0, and because of S's cardinality, we have

$$\text{Prob}(\neg C_0^{good} \mid AG = S) < (1 - 1/n)^{2n/3} < e^{-2/3}.$$

Hence

$$\text{Prob}(C_0^{good} \mid AG = S) > 1 - e^{-2/3}.$$

Thus

$$\begin{aligned}\text{Prob}(C_0^{good}) &= \sum_{|S|>2n/3} \text{Prob}(C_0^{good} \mid AG = S) \cdot \text{Prob}(AG = S) \\ &> \sum_{|S|>2n/3} (1 - e^{-2/3}) \,\text{Prob}(AG = S) \\ &= (1 - e^{-2/3}) \sum_{|S|>2n/3} \text{Prob}(AG = S) = 1 - e^{-2/3}.\end{aligned}$$

Lower-bounding P_1. If $S \subset [1,n]$, let $bad \neq S$ denote the following event: $S = \{j \in [1,n] : \forall \text{ good } g,\ SUM_{gj} \neq bad\}$. Notice that if $\text{Prob}(bad \neq S) > 0$, then S's cardinality is greater than $2n/3$. (Indeed, $SUM_{gG} \neq bad$ for all good players g and G.) Also notice that $\sum_{|S|>2n/3} \text{Prob}(bad \neq S) = 1$. (Indeed, in any of our random executions, there must be more than $2n/3$ good players.)

We now lower-bound P_1 as follows.

$$\begin{aligned}P_1 &= \sum_{|S|>2n/3} \text{Prob}(\forall j \in S\ sum_j \neq 0 | bad \neq S) \cdot \text{Prob}(bad \neq S) \\ &\overset{\text{Lemma 7}}{=} \sum_{|S|>2n/3} (1 - 1/n)^{|S|} \cdot \text{Prob}(bad \neq S) \\ &\geq \sum_{|S|>2n/3} (1 - 1/n)^n \cdot \text{Prob}(bad \neq S) > e^{-1} \cdot \sum_{|S|>2n/3} \text{Prob}(bad \neq S) = e^{-1}.\end{aligned}$$

Since our lower bounds on P_0 and P_1 do not depend on n, A, or IQ, we have proved that the fairness of protocol OC is $\min(P_0, P_1) = \min(1 - e^{-2/3}, e^{-1}) = e^{-1} > .35$. □

[36] The bad players may also contribute to raising the probability of the coin being unanimously 0. For instance, it is enough that, for some bad player j, the adversary acts so that for all good g, $SUM_{Gj} \neq bad$. By Lemma 7, sum_j then has a $1/n$ chance of being equal to 0. Our lower bound, however, must hold for all possible adversaries; thus we have to disregard this probability from our computation since it may be 0 for some adversaries. Instead, we must consider and guard against such possible behavior of the adversary when lower-bounding the probability of the coin being unanimously 1.

8. Byzantine agreement from oblivious common coins.

8.1. The notion of Byzantine agreement. When Byzantine agreement is needed, the values to be agreed upon may have arbitrary length. Without loss of generality, however, we restrict our attention to the case where every initial value is a single bit; in fact, in [13] and [37], it is proved that general Byzantine agreement is reducible to the binary case in a constant number of rounds.

DEFINITION 15. *We say that a protocol P is a* Byzantine agreement protocol (with fault-tolerance c) *if, for all c-adversaries A, any string H_A^0, any number of players n, and any bits $b_1, \ldots, b_n$, in any execution of $P(n)$ with adversary A on initial adversarial history H_A^0 and inputs $b_1, \ldots, b_n$, there exists a bit d such that the following two properties hold:*

1. Consistency: *Every good player that halts outputs d.*
2. Validity: *If there exists a bit b such that, for all initially good player i, $b_i = b$, then $d = b$.*

Notice that the above definition does not require that a Byzantine agreement protocol ever terminate. A Byzantine agreement protocol is most interesting, however, only if it terminates with positive probability, has high fault tolerance, and requires only a "moderate computational effort" from the good players.

8.2. An optimal Byzantine agreement protocol. We are finally ready to construct our Byzantine agreement protocol from our discussed primitives. It consists of three basic subprotocols: $\mathcal{P}_r$, $\mathcal{P}_1$, and $\mathcal{P}_0$. Subprotocol $\mathcal{P}_r$ includes instructions for "randomly flipping" an oblivious common coin. Protocols $\mathcal{P}_0$ and $\mathcal{P}_1$ actually consist of protocol $\mathcal{P}_r$, where the outcome of the coin flip is *forced* to be, respectively, 0 and 1. Thus, although the coin flips of subprotocols $\mathcal{P}_0$ and $\mathcal{P}_1$ are predictable, they have the advantage that all good processors are "aware" of the result, and this result is always the same for all good processors.

The goal of protocol $\mathcal{P}_r$ is to give the network a chance of reaching an "oblivious agreement" (i.e., with positive probability, all players adopt the same bit without knowing that this has happened). The goal of protocols $\mathcal{P}_0$ and $\mathcal{P}_1$ is to provide a *proof*, if all players are in oblivious agreement, that agreement has indeed been reached, so as to allow everyone to terminate. More precisely, if the good players obliviously agree on 0 (respectively, 1), an execution of $\mathcal{P}_0$ (respectively, $\mathcal{P}_1$) makes them aware that they are in agreement on 0 (respectively, 1) and terminate.

Our Byzantine agreement protocol is not a fixed-round protocol. Rather, each good player i keeps on executing, in order, subprotocols $\mathcal{P}_r$, $\mathcal{P}_0$, and $\mathcal{P}_1$ until he *individually* terminates. It thus may happen that different good processors terminate at different rounds. Nonetheless, in a random execution with a 1/3-adversary, our protocol terminates with probability 1, and when that happens, the outputs of all good players, though produced at different times, will always satisfy the consistency and validity requirements.

THEOREM 4. *BA is a Byzantine agreement protocol with fault tolerance* 1/3 *and runs in expected polynomial time and in an expected constant number of rounds.*

(More precisely, there exists a polynomial Q and constant c such that, for any number of players n, any 1/3-adversary A, any initial quantities IQ, and any positive integer k, the probability that, in randomly executing $BA(n)$ with A on IQ, the protocol does not halt within $Q(n)k$ BA-steps and ck rounds is less than 2^{-k}.)

Proof. Let us start by establishing a convenient notation.

LOCAL DEFINITIONS. *In an execution of BA, we call a good player* dead *if he has already terminated and* alive *otherwise.*

PROTOCOL $BA(n)$

Input for player i: b_i, a bit. "We actually consider b_i as a variable of player i whose initial value coincides with i's input bit."

Code for every P layer i.

0: For all players j, set $B_j = 0$. "B_j represents the last one-bit message received from player j."

(Subprotocol $\mathcal{P}_r$.)

1: Distribute b_i.

2: For all j, if $b_j^* \in \{0,1\}$, then reset $B_j := b_j^*$; else, reset $b_j^* := B_j$. Let $count_i = tally(1)$.

"In other words, for the purpose of computing $tally(1)$, if you did not receive a bit from player j, assume that he *virtually* sent you the same bit that he *really* sent you last."

Run $OC(n)$ and let r_i be your output. Then:

(a) If $count_i \in [0, n/3)$, then reset $b_i := 0$. Else:

(b) If $count_i \in [n/3, 2n/3)$, then reset $b_i := r_i$. Else:

(c) If $count_i \in [2n/3, n]$, then reset $b_i := 1$.

(Subprotocol $\mathcal{P}_0$.)

3: Distribute b_i.

4: For all j, if $b_j^* \in \{0,1\}$, then reset $B_j := b_j^*$; else, reset $b_j^* := B_j$. Let $count_i = tally(1)$. Then:

(a) If $count_i \in [0, n/3)$, then output 0, distribute 0 in next round, and TERMINATE

"In a round from now, you will be dead and will keep on *virtually* distributing 0. Every other good player is either dead and his output is 0, or will terminate, outputting 0."

(b) If $count_i \in [n/3, 2n/3)$, then reset $b_i := 0$. Else:

(c) If $count_i \in [2n/3, n]$, then reset $b_i := 1$.

(Subprotocol $\mathcal{P}_1$)

5: Distribute b_i.

6: For all j, if $b_j^* \in \{0,1\}$, then reset $B_j := b_j^*$; else, reset $b_j^* := B_j$. Let $count_i = tally(1)$. Then:

(a) If $count_i \in [0, n/3)$, then reset $b_i := 0$. Else:

(b) If $count_i \in [n/3, 2n/3)$, then reset $b_i := 1$. Else:

(c) If $count_i \in [2n/3, n]$, then output 1, distribute 1 in next round, and TERMINATE.

"In a round from now, you will be dead and will keep on *virtually* distributing 1. Every other good player is either dead and his output is 1, or will terminate outputting 1."

Go to step 1.

Let $\mathcal{P} \in \{\mathcal{P}_r, \mathcal{P}_1, \mathcal{P}_0\}$ and $b \in \{0,1\}$. Within an execution of BA, we say that at the start (at the end) of an execution of subprotocol $\mathcal{P}$, the network is in agreement *on b if, for all good players g, either g is dead and his output is b or he is alive and the current value of variable b_g is b.*

We say that at the start (at the end) of an execution of $\mathcal{P}$, the network is in agreement *if there exists a bit b such that the network is in agreement on b.*

CLAIM T4-1. *For any subprotocol $\mathcal{P} \in \{\mathcal{P}_r, \mathcal{P}_1, \mathcal{P}_0\}$, any execution of $\mathcal{P}$ with a 1/3-adversary, and any alive good players g and G, $|count_g - count_G| < n/3$.*

Proof. Only the bad processors may send different bits to different players in the same step. Thus, at any given step, the difference between the tallies (of 1) of two good players is upper-bounded by the number of currently bad players and thus by $n/3$. ■

CLAIM T4-2. *For all $\mathcal{P} \in \{\mathcal{P}_r, \mathcal{P}_1, \mathcal{P}_0\}$, for all executions of $\mathcal{P}$ with a 1/3-adversary, and for all bits b, if the network is in agreement on b at the start of the execution, it is in agreement on b at its end.*

Proof. Since each execution of subprotocols $\mathcal{P}_1$ and $\mathcal{P}_0$ is in essence a special execution of $\mathcal{P}_r$, it is sufficient to prove our claim with respect to this latter protocol. Assume that at the start of $\mathcal{P}_r$, the network is in agreement on 0; that is, every good dead player outputs 0 and, for all good alive players g, $b_g = 0$. Then all good players ("really" the alive ones and "virtually" the dead ones) distribute 0 in step 1. This implies that only the bad players can distribute 1 in step 1; thus in step 2, for all good alive g, $count_g < n/3$. As a consequence, independently of his own output of subprotocol OC, at the end of step 2, each good alive player g sets $b_g := 0$ in accordance with instruction 2(a); that is, the network is in agreement on 0 at the end of $\mathcal{P}_r$. The case in which the network is in agreement on 1 at the start of $\mathcal{P}_r$ is handled similarly. ■

CLAIM T4-3. *In any execution of BA with a 1/3-adversary, whenever a good player outputs a bit, the network is in agreement on that bit.*

Proof. A good processor generates an output only during the execution of either subprotocol $\mathcal{P}_1$ or subprotocol $\mathcal{P}_0$. Let E be the first execution of either $\mathcal{P}_1$ or $\mathcal{P}_0$ in which a good player produces an output, and let g be one such player. Assume that E is an execution of $\mathcal{P}_0$; then all good players are alive during E, and g must output 0 at E's end. Thus at E's end, $count_g \in [0, n/3)$. Therefore, by Claim T4-1, for all good G, $count_G \in [0, 2n/3)$. This entails that, because of either rule (a) or rule (b), every good player G resets $b_G := 0$; that is (because there are no dead good players), the network is in agreement on 0 at the end of E (though only those good players whose counter belongs to $[0, n/3)$ are aware of this and will thus output 0 and terminate). If there are no more executions of either $\mathcal{P}_1$ or $\mathcal{P}_0$ in which a good player outputs a bit, then we are done. Otherwise, because of Claim T4-2, since the network is in agreement on 0 at E's end, it will remain in agreement on 0 thereafter. Thus, whenever a good player outputs a bit later on, this bit must be 0, in accordance with our claim. The case in which E is an execution of $\mathcal{P}_1$ is argued in the "symmetric" way. ■

CLAIM T4-4. *For any random execution of BA with a 1/3-adversary and any positive integer k, if the network is not in agreement at the start of the kth execution of subprotocol $\mathcal{P}_r$, then the probability that it will be in agreement at its end is greater than .35.*

Proof. Because of Claim T4-3, our hypothesis implies that all good players are alive throughout our execution of $\mathcal{P}_r$. Moreover, since by Claim T4-1 we have $|count_g - count_G| < n/3$ for any good players g and G, one of the following two cases must occur:

(0) $\forall good\ i,\ count_i \in [0, 2n/3)$, or

(1) $\forall good\ i,\ count_i \in (n/3, n]$.

When case (0) occurs, if the oblivious common coin is unanimously 0, then each good player i resets $b_i := 0$ (if $count_i \in [0, n/3]$ because of rule 2(a), if $count_i \in (n/3, 2n/3)$ because of rule 2(b)) and thus the network is in agreement on 0. Similarly, when case (1) occurs, if the oblivious common coin is unanimously 1, then every good processor i resets $b_i := 1$ and thus—since there are no dead good players to worry about—the

network is in agreement on 1. Since either case (0) or case (1) must occur, and since OC is an oblivious coin protocol with fairness .35, the probability that the network is in agreement at the end of a random execution of $\mathcal{P}_r$ (though the good players may not be "aware" of this event) is greater than .35. ■

CLAIM T4-5. *In any execution of BA with a 1/3-adversary, if at the beginning of an execution of subprotocol $\mathcal{P}_1$ (respectively, $\mathcal{P}_0$), the network is in agreement on 1 (respectively, 0), then one round after the end of the subprotocol execution, BA terminates and the output of every good player is 1 (respectively, 0).*

Proof. Assume that the network is in agreement on 1 at the beginning of an execution of $\mathcal{P}_1$ (the "0 case" is similarly handled). Then all dead good processors have output 1 prior to the present execution of $\mathcal{P}_1$, and all alive good processors distribute 1 in the first step of the execution. Thus, since their tallies of 1 belong to the interval $(n/3, n]$, all good (and alive) processors will perform instruction 4(c) throughout this execution. Therefore, each one of them outputs 1 and will terminate in the next round, unless he will get corrupted in the next round, an event that cannot, in any case, change the output of the still uncorrupted players or the termination of the protocol since, as usual, BA ends when all good players have terminated. ■

It is now easy to complete the proof of Theorem 4; we start by proving our claim about BA's round complexity and fault tolerance. $\mathcal{P}_r$ is a 36-round protocol, while $\mathcal{P}_0$ and $\mathcal{P}_1$ are both two-round protocols. Protocol BA iterates the ordered execution of $\mathcal{P}_r, \mathcal{P}_0$, and $\mathcal{P}_1$ until all good players terminate. Claim T4-4 guarantees that, no matter what the initial quantities and the strategy of a 1/3-adversary may be, in a random execution of BA, the probability that an "oblivious" agreement is not reached after the $2k$th execution of $\mathcal{P}_r$ is less than $(.35)^{2k} < 2^{-k}$. Once oblivious agreement is reached at the end of an execution of $\mathcal{P}_r$, Claim T4-5 guarantees that, no matter what the actions of the 1/3-adversary may be, protocol BA halts—with all the good processors "aware" of having reached Byzantine agreement—within the next five rounds (i.e., at most one round after the end of $\mathcal{P}_1$ if the agreement was on 1). Thus the probability that protocol BA does not reach Byzantine agreement within $80k + 5$ rounds is less than 2^{-k}.

Let us now prove our claim about the amount of local computation of protocol BA. Having set (hidden) message bounds, the good processors do not waste running time reading excessively long messages sent by the adversary. Moreover, except for some occasional random selections, each round of protocol BA can be performed in fixed polynomial-in-n time. As for those random selections, they consist of the random choices of elements in integer intervals of the form $[0, z-1]$. Now, whenever z is a power of 2, a random selection in $[0, z-1]$ can be performed by flipping $\log z$ coins—and thus in fixed (as opposed to expected) polynomial time by a probabilistic Turing machine. However, if z is not a power of 2, then the adopted strategy for this task consists of randomly selecting a $\lceil \log z \rceil$-bit string until a member of the desired interval is found. Clearly, the probability that more than T such trials are needed is less than 2^{-T}. Since in each iteration of $\mathcal{P}_r$, $\mathcal{P}_0$, and $\mathcal{P}_1$, at most $Q(n)$ such selections (where Q is a given polynomial) must be made by the good players, since the rest of the computation can be performed in fixed polynomial time, and because of our recently proven claim about the round complexity of BA, our claim about the running time of BA easily follows. □

Remarks.

• Our reduction of Byzantine agreement to (oblivious) common coins was inspired by an earlier work of Rabin [34]. His reduction is much simpler, but it assumes a common coin that not only is *externally provided* but also is not oblivious (i.e., all

players are *guaranteed* to see the same common random bit[37]), and it requires that the number of faults is $< n/4$.

• As we have seen, protocol BA enjoys the property of being *always correct and probably fast*; that is, our use of probabilism introduces some uncertainty of how long it will take to terminate (a modest uncertainty since we prove that the expected number of rounds is constant), but no possibility of error in the correctness of the final agreement. This desirable property implies that one cannot get rid of "expected" in our round complexity. In fact, an algorithm that reaches a guaranteed agreement in a fixed number of rounds, no matter what the sequence of its coin tosses may be, is immediately transformed to a fixed-round, deterministic algorithm. Thus the result of Fischer and Lynch [20] would imply that at least $O(n)$ rounds are needed if the number of possible faults is $O(n)$.

• In general, as we have said, the input of a processor is a *private* value; that is, the adversary has no way of knowing it unless she corrupts its corresponding processor or this processor is instructed by the protocol to divulge it. Privacy of the initial inputs is also a necessary condition for certain protocols to be meaningful. This is indeed the case, for instance, with protocol *GradedVSS*—indeed, unless the input of an honest dealer is secret, there is no hope that an adversary cannot guess it better than at random. In the case of Byzantine agreement, on the other hand, the privacy of the initial inputs plays no role in defining the problem, which in fact remains totally meaningful even if we assume that the players' initial bits are known to the adversary.[38] Indeed, it should be noted that our protocol BA instructs each good player to distribute his input bit at the very first step, and it thus works even in the case in which the adversary knows the input bits of all players in advance.

• As we know, protocol BA relies on subprotocol OC. One may describe this subprotocol as producing a bit that is "sufficiently random and common." Such a description would, however, be quite incomplete. Namely, the output of OC is also sufficiently *unpredictable* at the start of each execution of the protocol. In fact, if the fairness of the coin that OC produces is positive, then we know that in any random execution both 0 and 1 have a positive probability of being output. It should be noticed that this unpredictability is used in our Byzantine agreement protocol: in protocol $\mathcal{P}_r$, "the oblivious coin" is flipped *after* every processor i distributes his current value b_i; thus in step 1, the adversary must choose which values the bad players distribute when the oblivious coin is still unpredictable. Actually, the unpredictability of the oblivious common coin flip is more than merely *used* in our protocol; it is actually *crucial* to it: should the adversary know the result of the coin flips of OC in advance, she could prevent agreement indefinitely. In fact, a bit more precisely, it can be shown that if all processors have as a common input—at the beginning of the protocol—a sequence of *truly random and independent* (but also, necessarily, *predictable*) coin tosses and use these bits instead of the outcomes of OC in subprotocol $\mathcal{P}_r$, a 1/3-adversary can easily and indefinitely prevent agreement from being reached.

The above discussion can be summarized by saying that our protocol BA relies heavily on hiding—at least temporarily—information. We will further elaborate on this crucial point in section 9.2.

• As we have indicated, the good processors need not terminate simultaneously. Indeed, the adversary can force "staggered termination" if she so desires.

• To avoid staggered termination, one may consider iterating subprotocol $\mathcal{P}_r$ a

[37] In his scenario, random coin flips are "predistributed" by a trusted party. Thus once they are "revealed," all good processors will see the same result.

[38] This is a quite plausible scenario since bad guys tend to "know" more than good ones.

prescribed number of times. If this number of times is large enough, upon termination, agreement would be reached with high probability. However, such a protocol would be unsatisfactory. First, from a theoretical point of view, it would introduce a probability of error. (In other words, there would be a chance that upon termination the good processors may not be in agreement—an event that is not allowed by our definition.) Second, from a "practical" point of view, to ensure that agreement is reached with probability $1 - 2^{-k}$, the envisaged protocol would have *always* to run $\mathcal{P}_r$ k times. By comparison, our protocol will run $\mathcal{P}_r$ k times only "very seldomly," that is, with probability 2^{-k}. (Truly, each time that our protocol runs $\mathcal{P}_r$, it also runs $\mathcal{P}_0$ and $\mathcal{P}_1$, but these latter protocols require only two rounds each and are extremely simple. The brunt of the computation is constituted by $\mathcal{P}_r$ alone, which is a quite complex 34-round protocol.)

• It should be noticed that, since some good processors may be alive and some others may be dead, in some executions of $\mathcal{P}_r$, there may not be a 2/3 majority of good processors. In fact, the dead ones do not participate in the protocol but simply "virtually" send a bit at given times. Under these circumstances, the coin tosses of OC need not to be common or fair in any way. This is not a problem, though. As we have shown, when a good processor terminates, the processors are in agreement and agreement cannot be disrupted. Protocol $\mathcal{P}_r$ is thus executed at most once without an honest majority of players; in fact, all alive good processors will terminate one round after the next execution of either $\mathcal{P}_0$ or $\mathcal{P}_1$, whose coin toss the adversary does not control.

9. Adjustments and improvements.

9.1. The model independence of our Byzantine agreement protocol.

Pros and cons of standard networks. In presenting our Byzantine agreement protocol, following a time-honored tradition, we have chosen standard networks (i.e., networks in which every pair of processors is connected by a dedicated and private communication line) as its underlying communication model. This model has notably simplified our argument and has helped us to focus on the *essential* distributed aspects of the quintessentially distributed problem at hand without getting sidetracked by a variety of important but quite different issues. (*Essence*, of course, is in the eyes of the beholder!) Moreover, the standard-network model is quite realistic in some contexts—for instance, in the case of computer networks whose processors are not directly controlled by humans.[39] Unfortunately, this is also the context in which, in our opinion, Byzantine agreement is less meaningful, at least for the extremely malicious fault model addressed in this paper—which, regrettably, belongs to the domain of human interactions. As a matter of fact, people being what they are, private channels may prove to be too much of an abstraction. If a Byzantine agreement protocol were run in the context of an adversarial negotiation conducted in a computer network, it would be remarkable that impostors would chivalrously confine themselves to purely software attacks, refraining from tampering with the network itself. Indeed, if they did, communication channels would not remain "private" for too long, no matter how much metal they could be shielded with or how deep they could be routed. We thus wish to briefly discuss what happens to our algorithm when its communication model is more... "humanized."

[39] Indeed, when such computers malfunction, they may start running algorithms that are different from their intended ones, may act—due to Murphy's law—as if they coordinate their disruptful efforts, and so on, but they cannot gain access to the dedicated line connecting two properly working processors!

Other possible models. If the adversary may prevent messages between good processors from being delivered, Byzantine agreement would be impossible. However, we may still trust our network to be asynchronous; that is, the adversary might delay messages arbitrarily long but cannot prevent them from eventually reaching their intended recipients. (For a discussion of this model, consult, for instance, [19].) Fortunately, our Byzantine agreement protocol has been ingeniously extended by Feldman [17] and Canetti and Rabin [7] to work on asynchronous networks as well.

If the adversary is able to change the messages exchanged between two good processors, Byzantine agreement would also be impossible since a single faulty processor could impersonate as many processors as it likes. Alternatively, the adversary may be capable of reading messages between good processors but not altering them.[40] In either case, one can still run our protocol using cryptography to simulate the privacy of such "public" lines (assuming, of course, that the adversary is computationally bounded). The basic underlying idea is that injecting or altering messages may be made infeasible by secure digital signatures that are secure in the sense of [25], while reading messages can be made infeasible by an encryption scheme that is secure in the sense of [23]. (One caveat, however: for very subtle reasons that exceed the scope of this paper, this basic strategy is surprisingly hard to implement correctly.)

If the adversary can "disconnect" two good processors, Byzantine agreement would again be impossible. However, rather than assuming that our network is complete, we may trust that it has some special, uncorruptable nodes that do not perform any computation but simply reliably route properly labeled messages. (Indeed, this may allow for quite sparse networks.) In this setting, our protocol would work essentially without any changes and with the same efficiency. Alternatively, one may consider networks with fewer communication lines but with sufficiently high connectivity. This way, for every set of faulty processors with small enough cardinality, every two good processors are still connected by a path consisting solely of good processors. Solving the problem in this new setting would require encrypting each message and sending it to its recipient through several node-disjoint paths. This, of course, would increase the running time of our protocol by a "network-topology" factor, but, most likely, the same increase in running time would be suffered by other protocols.

9.2. Improvements of our results. Our results have been found useful in several ways.

• As we have already mentioned in subsection 9.1, our Byzantine agreement protocol has been extended by Feldman [17] to work on asynchronous networks in which each pair of players is connected by a private channel. His asynchronous protocol tolerates up to $t < n/4$ faults. Using cryptography and assuming a computationally bounded adversary, it regains optimal fault tolerance, $t < n/3$, in the asynchronous case as well. Quite recently, Canetti and Rabin [7] have exhibited (for the same networks) an asynchronous Byzantine agreement protocol running in expected constant time and possessing resiliency 1/3 against an adversary with unbounded computational capabilities—though allowing a probability of error. (Let us note in passing that the notion of "constant time" must—and can—be meaningfully formulated in the asynchronous setting.)

• Ben-Or and El-Yaniv [4] have extended our algorithm to to reach Byzantine agreement in standard networks, in an expected constant number of rounds, for an entire collection of players' initial values.

[40] This may be the case in an ordinary telephone network, whose lines can be easily eavesdropped, while the voices of its users may be hard to imitate.

• Using our results and those of [4], Micali and Rabin [29] have obtained a VSS protocol (i.e., a "nongraded one"!) that works in standard networks (rather than standard-plus-broadcast ones), runs in polynomial time and an expected constant number of rounds, and tolerates any $n/3$ faults in the worst model. They have also exhibited a *nonoblivious* common coin protocol, with fairness $1/2$ and fault tolerance $1/3$, that works in standard networks and runs in expected polynomial time and an expected constant number of rounds. (Dolev, Dwork, and Yung have informed them that they have independently found these same protocols.)

• Goldreich and Petrank [27] have shown how to modify our algorithm so as to keep its expected running time and round complexity, while guaranteeing termination in the worst case (i.e., with the most unlucky sequence of coin tosses) in $t + O(\log t)$ rounds whenever the upper bound on the number of faulty players is t. (Thus termination is guaranteed in $O(n)$ rounds in the worst-fault model.)

10. Significance.

10.1. The "right" significance of Byzantine agreement. Until now, we have been advocating that Byzantine agreement is "the best one can do, in an adversarial scenario, when broadcasting is impossible." At this point, having gained more experience with adversarial behavior, we wish to point out that this informal saying is misleading in that it seems to imply that broadcasting is an available resource, and only when you are deprived of it should you turn to Byzantine agreement as a meaningful substitute. The truth is that, in an adversarial setting, closer scrutiny reveals broadcasting to be "almost always impossible."

Consider, for instance, a radio network. The recipient of a message in such a network cannot tell whether a satellite has aimed its signal to his specific geographical area or to the whole country. Moreover, since imitating (or cutting and pasting recorded pieces of) one's voice is quite possible, the recipient of a radio message cannot have any certainty about the identity of the sender of the message. Indeed, in an adversarial setting, *broadcasting is an abstraction.* Thus a natural question arises:

> In what "reasonable" communication models can one "concretely implement" an abstract notion satisfactorily close to that of broadcasting?

It is in light of this question that Byzantine agreement achieves, in our view, its true significance; namely, it demonstrates that standard networks offer a reasonable communication model to approximate, despite the presence of adversaries, the abstract notion of a broadcasting. Better said:

> We regard Byzantine agreement as showing that the abstraction of broadcasting can be meaningfully approximated by "simpler" abstractions: strong honest majority, synchrony, and private channels (and by even simpler ones, as we have discussed in section 9.1).

10.2. The significance of our results. It is now time to ask ourselves, "What is the significance of our own result?"

While our simplest primitives—*Gradecast* and, for small n, *GradedVSS*—are quite practical, we do not expect our Byzantine agreement protocol to have a *direct* practical impact. In fact, though it does not have any monstrous "hidden constants" and is actually quite feasible, our protocol starts outperforming prior ones when run in standard networks (or networks with "simulated standardness," as discussed in section 9.1) with a few hundred players.[41]

[41] Should standard networks of this size become feasible, our result actually opens the possibility of *artificially increasing* the number of players so as to increase the reliability of the network without

However, our results should have an *indirect* practical impact. Solving a long-standing open problem always marks a technical advance in a given field, and it is reasonable to expect that in our case as well this increased level of understanding will eventually translate into more practical protocols than ours.

More importantly, our techniques will be quite effective when dealing with much more complex problems than Byzantine agreement, that is, with those problems for which the existence of any solution is by itself a blessing and no superpractical answer can be legitimately expected.[42] In fact, it should be appreciated that our protocol solves a more difficult problem than Byzantine agreement (a fact that may perhaps excuse some of our complications): it provides a reasonably fair and fault-tolerant coin-flipping protocol in a quite unmanageable communication and fault model.[43]

Finally, *scientists shall not live by technique alone*, and we now wish to argue that our result is more significant from a purely conceptual point of view.

Probabilism versus determinism. Can randomness speed up computation? This is one of the most intriguing and fundamental questions of complexity theory. The celebrated probabilistic algorithms for primality *testing* of Solovay and Strassen [36] and Rabin [33] (and the more recent and equally beautiful ones for primality *proving* of Goldwasser and Kilian [22] and Adleman and Huang [1]) show that efficient probabilistic solutions exist for problems for which no polynomial-time solution is yet known. We cannot, however, prove that no deterministic, polynomial-time primality algorithm exists. Indeed, the fact that generating a sequence of coin tosses, independently from the problem at hand, may help solve our problem much faster is quite puzzling.

From this point of view, our result takes on a more serious significance. Namely, contrasting its performance with the quoted $t + 1$-round lower bound [20] for any deterministic protocol in which t malicious faults may occur, our Byzantine agreement protocol offers a dramatic example that, at least in some scenarios, probabilistic solutions *are provably vastly superior* to all deterministic ones.

Such a speedup was already demonstrated by Rabin [34], but by making the additional assumption of a common source of randomness *external to the network:* a common coin toss magically available to all processors at every clock tick. We instead demonstrate that randomness alone (i.e., individual and independent random choices made by individual processors), without any additional assumptions, suffices to beat any deterministic Byzantine agreement protocol in a dramatic way.

Privacy versus correctness. Our probabilistic solution to the synchronous Byzantine agreement problem sprung from recent advances in the field of *zero-knowledge computation.* Roughly said, this is the science of communication protocols that need to satisfy both a *correctness* and a *privacy* requirement. (For example, following the original application of Goldwasser, Micali, and Rackoff [24], a *zero-knowledge proof* shows that a given statement indeed possesses a correct proof but does not reveal what this proof might be.)

It should be noticed, however, that while Byzantine agreement has subtle correct-

making the time needed to reach agreement helplessly long. (In fact, if we know that—say—10% of the players are expected to become faulty during a decade, to ensure that 2/3 of them will be working properly in such a period, we are better off having a network of hundreds of processors rather than just a dozen of them.)

[42] Indeed, the usefulness of our algorithm for solving the problems mentioned in section 10.1 provides some support for this claim, and it augurs wonderfully for future ventures.

[43] Indeed, flipping a coin with adversaries does not get much easier even in friendlier scenarios than ours.

ness requirements, it has no constraints whatsoever about privacy.[44] Nonetheless, the correctness and speed of our protocol depend in a fundamental way on *GradedVSS*, a protocol where privacy *is* the central issue. We thus wish to advocate a novel role for privacy: namely, a tool for reaching correctness. This is less puzzling than it sounds. Our intuition behind it is simple:

> *Error in computation can be modeled as an adversary, and if your adversary "knows little," she can do little to disrupt your computation.*

Indeed, we believe that privacy will become a fundamental ingredient in the design of fault-tolerant protocols. Are we right? Time will tell. But may our journey be enjoyable in any case.

Acknowledgments. We are particularly grateful to Michael Fischer, Rosario Gennaro, Nancy Lynch, and David Shmoys for their generous, attentive, and constructive criticism.

Special thanks go to Ray Sidney, Tal Rabin, and Philip Rogaway. As we have already mentioned, the second author has collaborated with Philip Rogaway in modeling computation in the presence of faults in more complicated scenarios than the present one. The computational model of this paper has benefitted from the insights gained during that collaboration.

We would also like to acknowledge Michael Ben-Or, Benny Chor, Cynthia Dwork, Peter Elias, Rosario Gennaro, Oded Goldreich, Shafi Goldwasser, and Michael Rabin for many wonderful discussions about the Byzantine agreement problem.

Thanks also to two anonymous referees for their wonderful comments. The present version of our paper corresponds to the point in which one referee lamented that formalization exceeded intuition and another that intuition outmatched formalization.

Finally, our main motivation for working on the Byzantine agreement problem came from the *beauty* and *novelty* of the ideas of those who preceded us. We have immensely enjoyed standing on such tall shoulders!

REFERENCES

[1] L. M. Adleman and M. A. Huang, *Recognizing primes in random polynomial time*, in Proc. 19th ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 462–469.

[2] D. Beaver, S. Micali, and P. Rogaway, *The round complexity of secure protocols*, in Proc. 22th ACM Symposium on Theory of Computing, ACM, New York, 1990.

[3] M. Ben-Or, S. Goldwasser, and A. Wigderson, *Completeness theorems for fault-tolerant distributed computing*, in Proc. 20th ACM Symposium on Theory of Computing, ACM, New York, 1988, pp. 1–10.

[4] M. Ben-Or and R. El-Yaniv, *Interactive consistency in constant time*, Distrib. Comput., 1991, submitted.

[5] M. Ben-Or, *Another advantage of free choice: Completely asynchronous agreement protocols*, in Proc. 2nd Annual Symposium on Principles of Distributed Computing, ACM, New York, 1983, pp. 27–30.

[6] G. Bracha, *An "o(log n)" expected rounds randomized Byzantine generals protocol*, in Proc. 17th ACM Symposium on Theory of Computing, ACM, New York, 1985.

[7] R. Canetti and T. Rabin, *Fast asynchronous agreement with optimal resilience*, in Proc. 25th ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 42–51.

[8] B. Chor and B. Coan, *A simple and efficient randomized Byzantine agreement problem*, IEEE Trans. Software Engrg., SE-11 (1985), pp. 531–539.

[9] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, *Verifiable secret sharing and achieving simultaneity in the presence of faults*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 383–395.

[44] Indeed, our protocol BA starts by having each good processor distribute his own input value to all players.

[10] D. Chaum, C. Crepeau, and I. Damgård, *Multi-party unconditionally secure protocols*, in Proc. 20th ACM Symposium on Theory of Computing, ACM, New York, 1988.
[11] B. Chor and C. Dwork, *Randomization in Byzantine agreement*, in Randomness and Computation, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 433–498.
[12] D. Dolev, M. Fischer, R. Fowler, N. Lynch, and H. Strong, *An efficient algorithm for Byzantine agreement without authentication*, Inform. and Control, 52 (1982), pp. 257–274.
[13] D. Dolev, *The Byzantine generals strike again*, J. Algorithms, 3 (1982), pp. 14–30.
[14] D. Dolev and C. Dwork, manuscript, 1987.
[15] D. Dolev, C. Dwork, and M. Naor, *Non-malleable cryptography*, in Proc. 23rd ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 542–552.
[16] C. Dwork, D. Shmoys, and L. Stockmeyer, *Flipping persuasively in constant expected time*, SIAM J. Comput., 19 (1990), pp. 472–499.
[17] P. Feldman, *Optimal algorithms for Byzantine agreement*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1988.
[18] P. Feldman and S. Micali, *Byzantine agreement in constant expected time (and trusting no one)*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 267–276.
[19] M. Fischer, *The consensus problem in unreliable distributed systems (a brief survey)*, in Proc. International Conference on Foundations of Computation, 1983.
[20] M. Fischer and N. Lynch, *A lower bound for the time to assure interactive consistency*, Inform. Process. Lett., 14 (1982), pp. 183–186.
[21] Z. Galil, S. Haber, and M. Yung, *Cryptographic computation: Secure falt-tolerant protocols and public-key model*, in Proc. CRYPTO '87, Springer-Verlag, Berlin, 1987, pp. 135–155.
[22] S. Goldwasser and J. Kilian, *Almost all primes can be quickly certified*, in Proc. 18th ACM Symposium on Theory of Computing, ACM, New York, 1986, pp. 316–329.
[23] S. Goldwasser and S. Micali, *Probabilistic encryption*, J. Comput. System Sci., 28 (1984), pp. 270–299.
[24] S. Goldwasser, S. Micali, and C. Rackoff, *The knowledge complexity of interactive proof-systems*, SIAM. J. Comput., 18 (1989), pp. 186–208.
[25] S. Goldwasser, S. Micali, and R. Rivest, *A digital signature scheme secure against adaptive chosen-message attacks*, SIAM J. Comput., 17 (1988), pp. 281–308.
[26] O. Goldreich, S. Micali, and A. Wigderson, *How to play any mental game, or a completeness theorem for protocols with honest majority*, in Proc. 19th ACM Symposium on Theory of Computing, ACM, New York, 1987, pp. 218–229.
[27] O. Goldreich and E. Petrank, *The best of both worlds: Guaranteeing termination in fast randomized Byzantine agreement protocols*, Inform. Process. Lett., 36 (1990), pp. 45–49.
[28] A. Karlin and A. Yao, manuscript, 1987.
[29] S. Micali and T. Rabin, *Collective coin tossing without assumptions nor broadcasting*, in Proc. CRYPTO '90, Springer-Verlag, Berlin, 1990, pp. 253–266.
[30] S. Micali and P. Rogaway, *Secure computation*, in Proc. CRYPTO '91, Springer-Verlag, Berlin, 1992; full paper available from authors.
[31] Y. Moses and O. Waarts, *Coordinated travel: $(t+1)$-round Byzantine agreement in polynomial time*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 246–255.
[32] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228–234.
[33] M. Rabin, *Probabilistic algorithms for testing primality*, J. Number Theory, 12 (1980), pp. 128–138.
[34] M. Rabin, *Randomized Byzantine generals*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1983, pp. 403–409.
[35] T. Rabin and M. Ben-Or, *Verifiable secret sharing and multiparty protocols with honest majority*, in Proc. 21th ACM Symposium on Theory of Computing, ACM, New York, 1989.
[36] R. Solovay and V. Strassen, *A fast Monte-Carlo test for primality*, SIAM J. Comput., 6 (1977), pp. 84–85.
[37] R. Turpin and B. Coan, *Extending binary Byzantine agreement to multivalued Byzantine agreement*, Inform. Process. Lett., 18 (1984), pp. 73–76.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 934–949, August 1997

002

ERROR-RESILIENT OPTIMAL DATA COMPRESSION*

JAMES A. STORER† AND JOHN H. REIF‡

Abstract. The problem of communication and computation in the presence of errors is difficult, and general solutions can be time consuming and inflexible (particularly when implemented with a prescribed error detection/correction). A reasonable approach is to investigate reliable communication in carefully selected areas of fundamental interest where specific solutions may be more practical than general purpose techniques.

In this paper, we study the problem of error-resilient communication and computation in a particularly challenging area, *adaptive lossless data compression*, where the devastating effect of error propagation is a long-standing open problem that was posed in the papers of Lempel and Ziv in the late 1970s. In fact, the non-error resilience of adaptive data compression has been a practical drawback of its use in many applications. Protocols that require the receiver to request retransmission from the sender when an error is detected can be impractical for many applications where such two-way communication is not possible or is self-defeating (e.g., with data compression, retransmission may be tantamount to losing the data that could have been transmitted in the mean time). In addition, bits of encoded data that are corrupted while data is in storage will in general not be recoverable and may corrupt the entire decompressed file. By *error resilience*, we mean that even though errors may not be detected, there are strong guarantees that their effects will not propagate.

Our main result is a provable error-resilient adaptive lossless data-compression algorithm which nevertheless maintains optimal compression over the usual input distributions (e.g., stationary ergodic sources). We state our result in the context of a more general model that we call *dynamic dictionary communication*, where a sender and receiver work in a "lock-step" cooperation to maintain identical copies of a *dictionary D* that is constantly changing. For lossless data compression, the dictionary stores a set of strings that have been seen in the past and data is compressed by sending only indices of strings over the channel. Other applications of our model include robotics (e.g., remote terrain mapping) and computational learning theory.

Key words. data compression, adaptive algorithm, communication channel, error propagation

AMS subject classification. 68Q25

PII. S0097539792240789

1. Introduction.

1.1. Dynamic dictionary communication.

We use the term *dynamic dictionary* communication to refer to the process of transmitting data over a communication channel that is encoded/decoded with respect to dynamically changing dictionary of data. As depicted in Figure 1, a sender generates an *input stream* of characters drawn from a fixed finite-input alphabet, which are encoded, transmitted over a communication channel (or saved on a storage device), decoded, and received as the *output stream*. The encoder and decoder cooperate to maintain identical copies of a *dictionary D* that is constantly changing. The encoder reads characters from the input stream that form an entry of D, transmits the index of this entry, and updates D with some method that depends only on the current contents of D and the current match. Similarly, the decoder repeatedly receives an index, retrieves the corresponding entry of D to write to the output stream, and then performs the same algorithm as the encoder to update D. Note that although both the encoding and decoding algorithms refer to their dictionary as D, errors on the communication channel or storage medium

*Received by the editors November 25, 1992; accepted for publication (in revised form) July 31, 1995.

http://www.siam.org/journals/sicomp/26-4/24078.html

†Computer Science Department, Brandeis University, Waltham, MA 02254 (storer@cs.brandeis.edu).

‡Computer Science Department, Duke University, Durham, NC 27707 (reif@cs.duke.edu).

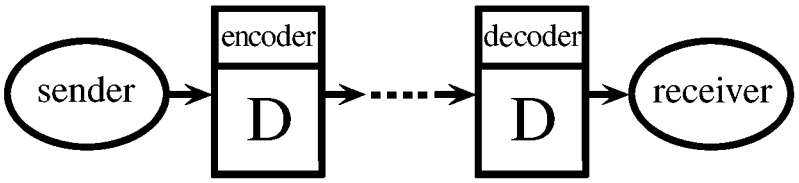


FIG. 1.1. *Dynamic dictionary communication.*

may cause the decoder's dictionary to differ from that of the encoder. Note also that D contains a set of entries (so when we talk about adding a new entry to D, we always mean to add it if it is not already present).

1.2. Applications of dynamic dictionary communication. The major application of dynamic dictionary communication that motivates this work is adaptive lossless data compression by textual substitution (Storer and Szymanski [1978]), where the dictionary stores a set of strings that have been seen in the past and data is compressed by sending only indices of strings over the channel; such compression algorithms are often called "LZ algorithms" after the work of Lempel and Ziv [1976] and Ziv and Lempel [1977, 1978]. (See Storer [1988] for an introduction to the subject and references to the literature.) Other possible applications of dynamic dictionary communication include computational learning theory and robotics (e.g., reporting of data by an autonomous remote robot that is mapping unexplored terrain and transmitting coordinates as displacements from previous locations).

1.3. Error resilience. A potential drawback of dynamic dictionary communication is *error propagation*; that is, a single error on the communication channel (e.g., insertion, deletion, or change of an index sent on the channel) could cause the dictionaries of the encoder and decoder to differ, which in the worst case could corrupt all data to follow. That is, since decoding is guaranteed to be correct only when the dictionary remains the same as the one used to encode the data, there is no way to bound the effects of a single error in the worst case. With *one-way* communication channels, where the decoder cannot send messages to the encoder, the problem becomes even more critical. In addition, with many existing communication systems where the full bandwidth of the channel is consumed, even if a two-way channel is available to let the encoder know that an error has occurred, retransmission of data can often be tantamount to losing new data that could have been transmitted during the time used to retransmit the old data. Retransmission can also introduce further error propagation problems. In addition, a data-storage device where data may be corrupted during storage can be viewed as a one-way communication channel.

In some sense, dynamic dictionary communication can be viewed as "raising the stakes" for the effects of errors on a one-way channel; even if the chance of an error is made very small via a standard error-detection/correction protocol, if an error does occur on a single data item, it can have the catastrophic effect of corrupting an unbounded number of additional data items. We present a technique for "error resilience," where no attempt is made to detect or correct errors but strong guarantees are provided that errors do not propagate. This technique can be combined with standard error-detection/correction protocols to yield one-way dynamic dictionary communication with a low rate of errors that do not propagate.

1.4. Outline. The next section contains basic definitions; it formally defines the dynamic dictionary communication model, which assumes several axioms, reviews

how data compression by textual substitution fits into the model and shows that the axioms hold for key methods that are provably optimal for stationary ergodic sources (the standard assumption in data compression literature), presents a model for errors on the communication channel, and defines error-resilient communication. Section 3 then presents a scheme for protecting against error propagation from k errors for any constant k under any distribution of errors; we make no attempt to analyze how this scheme affects the amount of compression achieved (although it appears to be quite practical). In section 4, we employ randomized techniques to "expand" the k-error technique to give propagation protection with very high probability against a fixed uniformly and independently distributed error rate of probability $1/r$; in particular, for any $k \geq 1$, the probability of error propagation can be made less than $1/r^k$. For example, an error rate of $1/10^{12}$ (a relatively "clean" communication channel with a low-overhead error-correcting mechanism) can be effectively "damped" to $1/10^{48}$ by choosing $k = 4$. In addition, the generalized k-error protocol presented in section 4 has no asymptotic affect on the amount of compression achieved. Section 5 discusses practical considerations.

2. Basic definitions.

2.1. Dynamic dictionary communication. Throughout this paper, when discussing dynamic dictionary communication with respect to a dictionary D, we use the following notation:

- $|D|$ = the current number of entries in D.
- $|D_{\max}|$ = the maximum number of entries that D may contain.
- $|s|$ = the number of characters in the string s.
- $BITS(i)$ = the number of bits needed to write i in binary (i.e., $\text{BITS}(i) = \lfloor \log_2(i) \rfloor + 1$).

Generic encoding and decoding algorithms for dynamic dictionary communication are shown below. Indices of D run from 0 to $|D| - 1$ and are represented using exactly $\text{BITS}(|D| - 1)$ bits, although in practice it is often the case that for simplicity $\text{BITS}(|D_{\max}| - 1)$ bits are used for all indices. We refer to codes sent over the channel that represent indices as *pointers*.

GENERIC ENCODING ALGORITHM.

1. Initialize the local dictionary D with one entry for each character of the input alphabet.
2. **repeat forever**
   - A. {Get the current match pointer p:}
     Use a *match method* to read a string s from the input.
     p = the index of s in D.
     Transmit p using $\text{BITS}(|D| - 1)$ bits.
   - B. {Update D:}
     Add each of the strings specified by an *update method* to D.

GENERIC DECODING ALGORITHM.

1. Initialize the local dictionary D by performing step 1 of the encoding algorithm.
2. **repeat forever**
   - A. {Get the current match string s:}
     Receive $\text{BITS}(|D| - 1)$ bits for the current match pointer p.
     s = the string in D corresponding to p.
     Output s.
   - B. {Update D:}
     Perform step 2B of the encoding algorithm.

The *match heuristic* reads from the input stream a string that is in the dictionary. (It must always be possible to read such a string since all strings of length 1 are always in the dictionary.) After the match has been encoded by the encoder or decoded by the decoder, "learning" consists of modifying the dictionary by adding one or more new strings with an update heuristic. Although it is possible to have the update heuristic employ a deletion heuristic that removes old entries to make room for new ones (see Storer [1988]), we restrict our attention here to update heuristics that do nothing once the dictionary is full so that encoding and decoding continue indefinitely (or until the system is restarted) without any further modification of the dictionary.

The exact choice of the match and update heuristics is not important for the work described here as long as the following axioms are satisfied. (We shall show shortly that these axioms hold for a variety of optimal adaptive lossless data compression methods.)

Succinctness. Except for the dictionary initialization (where entries for each of the characters of the input alphabet are formed), the string (or strings) added by the update heuristic at a given step depend only on the current match and the previous match.

Robustness.

A. There is a constant $1 \leq \rho$ such if the update heuristic adds a new match based on a particular successive pair of matches, it will do so after this successive pair has occurred at most ρ times.

B. There is a constant $1 \leq \alpha$, called the *learning constant*, such that the encoder dictionary reaches size $|D_{\max}|$ after at most $\alpha\rho|D_{\max}|$ entries have been transmitted.

The succinctness axiom addresses how dictionary entries may be formed. The robustness axiom allows the system to be "conservative" and to not form a new entry until it has "proved" itself by appearing a number of times, but not too conservative because this number of times must be bounded by a constant. In addition, the learning constant ensures that progress is made toward filling up the dictionary as data is encoded (and we don't waste too much effort relearning entries that have already been added earlier). Note that condition B of the robustness axiom is somewhat pessimistic because for all of the data-compression methods that we shall consider, if ρ, $\alpha > 1$, then the encoder dictionary reaches size $|D_{\max}|$ after at most $(\alpha + \rho)|D_{\max}|$ entries have been transmitted.

For technical reasons to be discussed shortly, our results will also work for a variation of the succinctness axiom.

Succinctness, version 2. Except for the entries that contain the characters of the input alphabet, the string (or strings) added by the update heuristic at a given step depend only on the current match and the next character.

The second version of the succinctness axiom is similar to the first version except that it is "one step ahead"; it models certain update heuristics that are important for proofs of optimality in applications involving data compression. Both versions have the property that except for the characters of the alphabet, each entry of the dictionary can be constructed from a pair of pointers to two other entries; this is all that will really be needed in our construction (and, as we shall see shortly, suffices for practical data-compression algorithms).

2.2. Applications to data compression. As mentioned earlier, the major application that motivates this work is data compression by textual substitution. In this section, we review a number of textual-substitution algorithms and show that

they can be modified to satisfy the robustness axiom without sacrificing compression. Our definition of "optimal" compression is *optimal in the information-theoretic sense*; that is, for stationary ergodic sources of entropy H, after processing n characters, the method in question sends an average of $H+\varepsilon(n)$ bits per character, where $\varepsilon(n)$ goes to 0 as n goes to ∞. (See Cover and Thomas [1991] or Gallager [1968] for definitions of entropy, etc.) This asymptotic measure of performance for stationary ergodic sources is standard throughout the literature. Furthermore, the methods we discuss here work well in practice for virtually any source. (See Storer [1988] for a detailed presentation of these methods as well as experimental results.)

Typically, the match heuristic used is the *greedy heuristic*; that is, read the longest match possible. Although other heuristics can be used and the greedy match heuristic does not guarantee the best possible compression on finite strings for any of the update heuristics to be discussed below (Storer [1988]), it does perform well in practice and is used in all of the methods that are provably optimal in the information-theoretic sense. Examples of update heuristics that might be used for data compression are as follows.

Uncompressed character (UC). Add the current match concatenated with the next character of the input; the next character of the input is sent along in uncompressed form as part of the current pointer (so that the next match starts after the character following the current match). This heuristic does not fit exactly into the generic encoding and decoding algorithms, but they can easily be modified to accommodate it by not allowing a match to a single character until it has been sent to the decoder. (The cost of dictionary entries "wasted" by characters that have yet to be seen becomes arbitrarily low as the dictionary size increases.) For finite-length strings and dictionaries of bounded size, UC typically does not perform as well in practice as the NC or FC heuristics discussed below, but we include it because it reflects the construction of Ziv and Lempel [1978] that is provably optimal.

Next character (NC). Add the current match concatenated with the next character of the input. This can be viewed as a more practical implementation of UC. It also requires modifications of the generic encoding and decoding algorithms because the next character of the input stream cannot be deduced by the decoder until the following match is received. (For the special "glitch" where the string matched at the current step is the one formed at the previous step, the decoder can deduce that the first and last character of the current match are the same as the first character of the previous match.) This is the heuristic used by Welch [1984] (upon which UNIX "compress" command is based) and Miller and Wegman [1985].

First character (FC). Add the last match concatenated with the first character of the current match. This heuristic performs similarly to NC in practice, but fits cleanly into the generic encoding and decoding algorithms (and does not have the decoding "glitch" mentioned above); it is discussed in Storer [1988].

Current match (CM). Add the last match concatenated with the current match. This heuristic is discussed by Miller and Wegman [1985] and Seery and Ziv [1977, 1978]. A variant of this method is employed by Reif and Storer [1990, 1992] and Royals et al. [1993] in a massively parallel systolic custom VLSI design.

All prefixes (AP). Add the set of strings consisting of the last match concatenated with each of the prefixes of the current match. This heuristic has the fast-growing characteristics of CM but like FC, NC, and UC maintains a dictionary with the prefix property (if a string is in the dictionary, then so are all of its prefixes); it is discussed in Storer [1988].

It is easy to see that the FC, CM, and AP update heuristics satisfy the succinctness axiom and that the UC and NC update heuristics satisfy version 2 of the succinctness axiom.

For part A of the robustness axiom, observe that the learning constant is 1 for the UC and NC heuristics since the greedy match heuristic ensures that the current match together with the next character of the input stream cannot already be in the dictionary. The FC heuristic has a learning constant of ≤ 2 since the greedy match heuristic insures that the only way the last match together with the first character of the current match can already be in the dictionary is when this is the same as the current match, which can only happen once. Similarly, it can be argued that the learning constant for CM and AP is ≤ 2.

Part B of the robustness axiom is satisfied by all of the heuristics listed above for $\rho = 1$. In this paper, we will use $\rho > 1$ in our constructions for error resilience (so that a successive pair of matches may be seen several times before learning takes place). We shall not address how this modification affects the amount of compression achieved by the CM and AP heuristics since they are not provably optimal to start with (Lempel and Ziv [1990]), although they work well in practice, and restrict our attention to the FC, NC, and UC heuristics. The UC heuristic is exactly the algorithm shown optimal in Ziv and Lempel [1978]; a nice proof that it is optimal in the information-theoretic sense appears in Cover and Thomas [1991]. The essence of this proof is to show that any method that parses the input stream into distinct phrases (and sends a number of bits equal to $\varepsilon(n)$ of the current number of phrases to the decoder) must be optimal. Furthermore, as outlined in the appendix, this proof can be modified to show that a *constant-redundant parsing* (one where there is a constant that bounds how many times any phrase may appear in the parsing) is also optimal. Hence a "redundant" version of UC is optimal. In addition, we also show in the appendix that redundant versions of FC and NC are optimal.

2.3. Bounded-size dictionaries. Theoretical proofs of optimality such as presented in Lempel and Ziv [1976] and Ziv and Lempel [1977, 1978] simply assume that the dictionary grows indefinitely as the infinitely long input stream is processed. However, a practical strength of the compression methods outlined earlier is that relatively small bounds on the size of the dictionary (e.g., 2^{12} to 2^{16} entries) provide good performance in practice on virtually all types of data. As mentioned earlier, we take the same approach here and simply assume that once the dictionary fills, it remains fixed for the remainder of the data to be processed (so that the update heuristic is defined to add nothing once the dictionary is full). Although this strategy typically works well on individual files, in practice some method for changing the dictionary over time after it has filled is usually incorporated into the algorithm. (See Storer [1988] for discussion of various strategies.) For example, the UNIX compress command monitors compression and restarts the dictionary-growing process if compression falls off after the dictionary is full. Another simple strategy is a "swapping" arrangement with two dictionaries, where after the dictionary fills for the first time, continue using it to compress data but at the same time start forming new entries in a second dictionary; once the second dictionary is full, it can be used for compression and the first dictionary reset to be empty, and so on. Since restarting or changing of dictionaries occurs infrequently, very secure protocols (e.g., several hundred bits to encode a single bit) can be used to periodically restart the process or to send a swap bit. Such a protocol can also be used to agree that the dictionaries are now "frozen" and will not be modified further. We shall not address these issues further in this

paper.

2.4. A model for error-resilient communication. We use the term *communication channel* to refer to any medium or device over which data is transmitted and received. We assume that pointers sent from the encoder to the decoder over a communication channel are subject to the following errors:

- *add:* An extra pointer, chosen at random, is inserted into the communication stream.
- *delete:* A pointer is deleted from the communication stream.
- *change:* A new pointer, chosen at random from the space of all pointers, replaces a pointer.

We consider the following classes of error distributions:

- *uniform:* Errors occur randomly.
- *arbitrary:* Errors may be arbitrarily correlated.

Note that our results do not apply to the case where an individual bit is inserted or deleted but do allow individual bits to be changed because this is just a special case of a change error. (In fact we are only charging a cost of 1 no matter how many bits of a pointer are changed.)

The goal is to provide guarantees that there is *perfect protection* against error propagation due to a specified number of channel errors; that is, there is no further corruption of the data beyond what is directly due to the channel errors.

3. Protection against k errors. In this section, we present an encoding scheme which, for any fixed integer constant $k \geq 0$, guarantees that k or fewer errors on the channel will not cause *any* error propagation. We will not address how this protocol affects the amount of compression (although it appears reasonable in practice); rather, this protocol will be an important building block for a more general construction, presented in the next section, that does guarantee that compression is not affected asymptotically.

To simplify our presentation, we shall assume that the update heuristic adds at most one new match (e.g., UC, NC, FC, and CM have this property but AP does not); at the end of this section, we describe how this encoding scheme can be generalized to update heuristics that may add more than one new entry. Also, similar modifications as for the generic encoding and decoding algorithms are discussed for the UC and NC heuristics.

The two key ideas are as follows:

- Hashing is used to compute where a new entry is to be placed in the dictionary. For simplicity, throughout this paper, we assume that a truly random hash function is used; in practice, a 2-universal hash function (see Carter and Wegman [1979]) can be used. Hashing has the effect of eliminating the dependence between addresses that is normally present in dynamic dictionary communication so that if a given index is not used right away, it will have no effect on what indices are used in the future.
- Counts are maintained for all pointer pairs seen thus far and a pair is used by the match heuristic only if it "warms up" to be a clear winner over pairs that hash to the same address.

DEFINITION. *Suppose that counts (initially 0) are maintained in the encoder for all pointer pairs sent thus far (i.e., each time a pointer is sent/received, the count of the pair of pointers it represents is incremented) and in the decoder for all pointer pairs received thus far. For a sequence S of pointers sent by the encoder,* $\mathrm{LAG}(S)$ *is the maximum amount that any count in the decoder is incorrectly increased due to errors on the channel plus the maximum amount any count fails to be increased due*

to errors on the channel. For any integer $k \geq 0$, *the* warming value *for* k, $w(k)$, *is the smallest integer such that* $\mathrm{LAG}(S) \leq w(k)$ *over all sequences of pointers with at most* k *errors.*

LAG THEOREM. *For any* $k \geq 1, w(k) \leq 3k$.

Proof. Let

$S = U_0 C_1 U_1 \ldots C_m U_m$ be the sequence of pointers *sent* by the encoder and

$R = U_0 I_1 U_1 \ldots I_m U_m$ be the sequence of pointers *received* by the decoder

where

$U_i, 0 < i < m, |U_i| \geq 1$ are blocks of pointers,

$I_i, 1 \leq i \leq m, |I_i| \geq 0$ are blocks of pointers, and

$|\prod_{i=1}^{m} I_m| \leq k$ (the product denotes string concatenation).

That is, S and R are partitions (which are not necessarily unique) of the input and output streams into blocks of pointers defined as follows. The U's are blocks of pointers that went across the channel *unchanged* and the C's are blocks of pointers that were *correctly* sent to the communication channel but due to errors on the channel were received *incorrectly* by the decoder as the I blocks, where all pointers in a given I block are incorrect.

Now consider a particular block C_i that is received as I_i. Let

$x =$ the number of pointers that are changed,

$y =$ the number of pointers that are deleted, and

$z =$ the number of pointers that are added

and write

$$C_i = q_1 \cdots q_{x+y},$$
$$I_i = r_1 \cdots r_{x+z}.$$

Any way of choosing x, y, and z that corresponds to a way to convert C_i to I_i suffices for our construction. Let p be the last pointer of U_{i-1} and s be the first pointer of U_{i+1}. The only counts that might not be incremented (but should have been) are due to the loss of

$$pq_1, \quad q_1 q_2 \cdots q_{x+y-1} q_{x+y}, \quad q_{x+y} s,$$

which in the worst case can increase the lag by $x + y + 1$. The only counts that might be incorrectly incremented are due to the addition of

$$pr_1, \quad r_1 r_2 \cdots r_{x+z-1} r_{x+z}, \quad r_{x+z} s,$$

which in the worst case can increase the lag by $x + z + 1$. Note that if $i = 1$ and/or $i = m$, then p and/or s may not exist, and this can only lower the number of counts that might not be incremented or might be incorrectly incremented. Thus the total change in lag for the transformation of C_i to I_i is at most $2x + y + z + 2$. In fact, for the case where $y = z = 0$ (there are only change errors), this upper bound can be reduced by observing that if

$$pq_1 = q_1 q_2 = \cdots q_{x+y-1} q_{x+y} = q_{x+y} s,$$

then it follows that $p = q_1 \cdots q_{x+y} = s$, and so it cannot be that $pr_1 = r_1 r_2$ (or that $pr_1 = r_1 s$) or r_1 would not be an incorrect pointer, so the lag can be at most $2x + 1$. Hence we have

$$\mathrm{LAG}(C_i) \leq \left\{ \begin{array}{ll} 2x + 1 & \text{if } (y + z) = 0, \\ 2x + y + z + 2 & \text{if } (y + z) > 0 \end{array} \right\}.$$

Let $e = x + y + z$. For the case where $(y + z) = 0$, assuming that $k > 0$ (otherwise, $\mathrm{LAG}(C_i) = 0$), $2x + 1 \leq 2e + 1 \leq 3e$. For the case where $(y + z) > 0$:

$$\begin{aligned} 2x + y + z + 2 &= 2(e - (y + z)) + (y + z) + 2 \\ &= 2e - (y + z) + 2 \\ &\leq 2e + 1 \\ &\leq 3e. \end{aligned}$$

Hence the warming value for each block of e errors is $\leq 3e$, and the theorem follows.

ENCODING ALGORITHM WITH k-ERROR PROTOCOL.

1. Initialize the *local dictionary* D to have one entry for each character of the input alphabet and to have an empty hash table that is capable of storing pointer pairs; let h denote a hash function that maps pointer pairs to the range $0 \ldots (|D| - 1)$.
2. **repeat forever**
   A. {Get the current match string s and compute the current match pointer p:}
      Read a string s for which there exists pointer pair qr such that
      - s = the string corresponding to the pointer pair qr,
      - $\mathrm{count}(qr) > \mathrm{count}(uv) + w(k)$ for all uv such that $h(uv) = h(qr)$.

      if s does not exist
      then p = the index in D of the next input character
      else begin
      $p = h(qr)$
      $\mathrm{count}(qr) = \mathrm{count}(qr) + 1$
      end
      Transmit p using $\mathrm{BITS}(|D| - 1)$ bits.
   B. {Update D:}
      for each pair xy produced by the update method **do**
      if xy is not already in the dictionary **then** $\mathrm{count}(xy) = 0$

DECODING ALGORITHM WITH k-ERROR PROTOCOL.

1. Initialize the local dictionary D by performing step 1 of the encoding algorithm.
2. **repeat forever**
   A. {Get the current match pointer p and compute the current match string s:}
      Receive $\mathrm{BITS}(|D| - 1)$ bits for the current match pointer p.
      if p represents a single character
      then s = the single character corresponding to p
      else if there is a pointer pair qr such that
      $h(qr) = p$ and $\mathrm{count}(qr)$ is largest among all pairs that hash to p
      then begin
      s = the string corresponding to qr
      $\mathrm{count}(qr) = \mathrm{count}(qr) + 1$
      end
      else s = the empty string
      Output s.
   B. {Update D:}
      Perform step 2B of the encoding algorithm.

Given the lag theorem, the generic encoding and decoding algorithms can be modified, as shown above, to employ the *k-error protocol* to insure perfect protection against any k errors; that is, no bytes are corrupted beyond those corrupted by channel errors.

For correctness of the protocol, observe that for each pointer p in a sequence S of pointers sent by the encoder, the pointer pair qr that p represents is a clear "winner" among all pointer pairs that hash to p; that is, the count of qr is greater by at least $\mathrm{LAG}(S)$ than the count of any other pair uv that hashes to p. Hence when the decoder receives an uncorrupted pointer p, the pointer pair qr with the largest count that hashes to p must be the correct pair for p because k errors cannot cause some other pair uv that hashes to p to have a count equal or greater than that of qr.

As mentioned at the beginning of this section, the encoding and decoding algorithms with the k-error protocol can be adapted to the UC and NC heuristics, and to heuristics such as AP that may add more than one entry. For the UC and NC heuristics, the same modifications as for the generic encoding and decoding algorithms can be used. For the AP heuristic, or any heuristic that may construct more than one match from the last match and the current match, we can hash on triples consisting of the two pointers in question together with an integer that indicates which of the strings corresponding constructed from this pointer pair is being referenced; this integer can be provided by the match heuristic. The lag theorem can still be used to verify the protocol because it is still the case that at most one count is incremented when a pointer is sent by the encoder or received by the decoder.

We leave as an open problem the class of sources for which the k-error protocol produces a constant-redundant parsing for update heuristics that satisfy the succinctness and robustness axioms. All that has been shown here is that k errors will not propagate when using the k-error protocol. Because hash conflicts could in theory cause the counts of two entries that hash to the same location to "race" (so that neither count sufficiently exceeded the other), it is not clear what effect, if any, the protocol has asymptotically on the amount of compression obtained with a given update heuristic (as compared with the same heuristic without the protocol). We conjecture that in practice, performance will not be significantly affected for small values of k and reasonably size dictionaries (e.g., 2^{16} or more entries). In the next section, we make use of the k-error protocol in a way that avoids hash conflicts and insures constant redundancy.

4. High-probability protection against an error rate. In this section, we employ probabilistic analysis to examine more carefully the proof of the lag theorem of the last section. The idea is to show that the k-error protocol actually gives, with high probability, strong protection against a fixed error rate during the period when the dictionary is changing and is vulnerable to error propagation. In addition, by encoding pointers to avoid hash conflicts with high probability, we can guarantee no asymptotic loss of compression.

We employ the Chernoff bound (see Hagerup and Rub [1989]) on the number of heads X in a sequence of independent coin tossings with expected value μ ($e = 2.7182\ldots$ denotes the natural logarithm base defined by $\lim_{n\to\infty}(\frac{n}{n-1})^n$):

$$\text{For } x \geq 0, \quad \mathrm{Prob}(X \geq (1+h)\mu) < \left(\frac{e^h}{(1+h)^{(1+h)}}\right)^{\mu}.$$

To obtain the form of the bound that we shall use here, first we observe that the right-hand side is

$$= \frac{1}{e^{\mu}}\left(\frac{e}{1+h}\right)^{(1+h)u} = \frac{1}{e^{\mu}}\left(\frac{(1+h)\mu}{e\mu}\right)^{-(1+h)\mu}$$

and hence, since $e > 1$, it follows that

$$\text{For } z > e\mu, \quad \text{Prob}(X \geq z) < \frac{1}{e^{\mu}} \left(\frac{z}{e\mu}\right)^{-z} < \left(\frac{z}{e\mu}\right)^{-z}.$$

DAMPING THEOREM. *If a sufficiently large dictionary is employed with a method satisfying the succinctness and robustness axioms together with the k-error protocol of the last section (where $w(k)$ denotes the warming value for k and α denotes the learning constant) on a channel with a uniform independent error probability of $1/r$ (on the average, one error for every r pointers), then the probability that the system looses stability (i.e., the probability that any errors propagate) is*

$$< \frac{1}{\left(\frac{r}{4e\alpha}\right)^{\left(\frac{w(k)}{2}+1\right)}}.$$

In addition, with probability greater than $1 - 1/2^{|D|(1-\varepsilon(|D|))}$, the amount of compression achieved is within a factor of $(1+\varepsilon(|D|))$ of what would be achieved with the same method used on a perfect channel without the protocol, where $\varepsilon(|D|) \to 0$ as $|D| \to \infty$.

Proof. To simplify notation, we assume throughout this proof that all logarithms are base 2.

Although the k-error protocol may work in the presence of hash conflicts, it is difficult to estimate the effect of the hash conflicts on the performance of the protocol for the application in question; for example, for data compression, it is not clear how the compression achieved is affected in the worst case. We avoid this problem by employing a more complicated hashing scheme that has no conflicts with extremely high probability. Although this scheme may increase the number of bits in some pointers, it will have no asymptotic effect on the total number of bits sent.

To store n items in the hash table, we use a table of size $n \log(n)^2$. Since at any time the table contains at most n elements, each time an element is inserted into the table, the probability that it goes to a bucket that already contains an entry is at most $n/(n \log(n)^2) = 1/(\log(n)^2)$, and thus the expected number of hash conflicts after inserting all n elements is bounded above by $n/\log(n)^2$. Hence from the Chernoff bound, with $\mu = n/\log(n)^2$ and $z = n/\log(n)$, it follows that the number of hash conflicts is greater than $n/\log(n)$ with probability

$$\begin{aligned} &< \left(\frac{n/\log(n)}{en/(\log(n)^2)}\right)^{-n/(\log(n)^2)} \\ &= \left(\frac{\log(n)}{e}\right)^{-n/(\log(n)^2)} \\ &< \frac{1}{2^{n/(\log(n)^2)}}. \end{aligned}$$

Thus it follows that

$$\text{Prob}(\text{number of hash conflicts } < n/\log(n)) > 1 - \frac{1}{2^{n/(\log(n)^2)}}.$$

The $n/\log(n)$ conflicting entries can all be hashed again into a table of size $n\log(n)$, the remaining $n/(\log(n)^2)$ conflicting entries can all be hashed again into a table of size n, and so on. In general, each pointer is now being represented by a sequence of indices, each consisting of a number of bits equal to log of the size of the corresponding hash table followed by a bit that is 0 if the index is the last in the pointer or 1 if there is another to follow. By summing the bits sent for a sequence of n pointers, we see that all n pointers use $\log(n\log(n)^2)+1$ bits for the first index, at most $n/\log(n)$ pointers have an additional $\log(n\log(n))+1$ bits for the second index, and so on. The first term is bounded by $(1+\varepsilon(n))n\log(n)$, where $\varepsilon(n)\to 0$ as $n\to\infty$, the second term is $O(n)$, and the remaining terms go down geometrically by a factor of $\log(n)$. Hence the total number of bits sent is asymptotically arbitrarily close to the $n\log(n)$ bits that are sent in any case.

Given that we can assume that there are no hash conflicts for the encoder, the k-error protocol takes care of conflicts at the decoding end. That is, if a pointer is received whose bits are the prefix of the bits of two or more pointers, the one with the largest count wins.

We now bound the average value of the following two quantities:

X_{ij} = the number of times that the count of pair i,j is incorrectly increased by the decoder due to errors on the channel, before the dictionary of the encoder is full;

Y_{ij} = the number of times that the count of a pair i,j fails to be increased when it would otherwise have been increased if there had been no errors on the channel, before the dictionary of the encoder becomes full.

Let

$E(D)$ = the expected number of pointers transmitted by the encoder before its dictionary is full.

From the proof of the lag theorem, we know that each error can cause at most two counts to incorrectly increase, and hence for each of the $E(D)/r$ pointers that are corrupted, at most two counts, which are equally likely to be any of at least $|D_{\max}|$ pairs, can be incorrectly increased. So the expected value of X_{ij}, which we denote by μX_{ij}, is bounded by

$$\mu X_{ij} \le \frac{2E(D)}{r|D_{\max}|}.$$

By the robustness axiom, $E(D)\le \alpha w(k)|D_{\max}|$, and hence

$$\mu X_{ij} \le \frac{2\alpha w(k)}{r}.$$

Since each error can cause at most two counts to incorrectly fail to increase, similar to X_{ij}, we can compute

$$\mu Y_{ij} \le \frac{2\alpha w(k)}{r}.$$

Hence the probability that X_{ij} or Y_{ij} is $\geq (\frac{w(k)}{2}+1)$ is

$$> \left(\frac{\frac{w(k)}{2}+1}{\frac{2e\alpha w(k)}{r}}\right)^{-\left(\frac{w(k)}{2}+1\right)}$$

$$= \frac{1}{\left(\frac{r(w(k)+2)}{4e\alpha w(k)}\right)^{\frac{w(k)}{2}+1}}$$

$$< \frac{1}{\left(\frac{r}{4e\alpha}\right)^{\frac{w(k)}{2}+1}}.$$

From the above bound, the theorem follows since a LAG of more than $w(k)$ cannot occur if no values of X_{ij} or Y_{ij} are more than $w(k)/2$.

Let us now consider the amount of compression achieved. We have already verified that each of the $|D|$ pointers is encoded with only a factor of $1+\varepsilon(|D|)$ more bits, where $\varepsilon(|D|) \to 0$ as $|D| \to \infty$. Hence if the method in question is optimal in the information-theoretic sense, then by the robustness axioms, as $|D| \to \infty$, the same method with the k-error protocol must also be optimal in the information theoretic sense (since it is $w(k)$-redundant).

COROLLARY. *For $\alpha \leq 2$ (which is the case for all data-compression methods that we have considered) and $r \geq 10^{12}$ (a reasonable assumption in practice for a clean channel with a low-overhead error-correction mechanism), choosing $w(k) \geq (2.2509k-2)$ yields a probability that is*

$$< \frac{1}{r^k}.$$

For example, if $k=5$, then by using $w(k)=10$, an error rate of $1/10^{12}$ is effectively "damped" to $1/10^{60}$.

Proof. If we write $w(k) = 2xk-2$, we can solve for the minimum value of x by simplifying the expression above as follows:

$$\leq \frac{1}{r^k\left(\frac{r^{\frac{w(k)}{2}+1-k}}{(8e)^{\frac{w(k)}{2}+1}}\right)}$$

$$= \frac{1}{r^k\left(\frac{r^{x-1}}{(8e)^x}\right)^k}.$$

The expression $\frac{r^{x-1}}{(8e)^x}$, which is monotonically increasing in x, becomes ≥ 1 when $(r/8e)^x \geq r$, which is true when $x \geq \frac{\log(r)}{\log(r)-\log(8e)}$. Assuming $r \geq 10^{12}$, $w(k) \geq 2.2509k-2$ suffices.

5. Practical considerations. The previous section encoded pointers to make the probability of *any* hash conflicts arbitrarily small. Although the protocol may work even in the presence of hash conflicts, this eliminated the need to address the issue of what affect hash conflicts have on performance of the system (e.g., the amount

of compression achieved). We leave it as an open problem whether, in the case of data compression for stationary ergodic sources, performance is affected when hash conflicts are allowed. Here we mention another strategy that can be viewed as a compromise between allowing hash conflicts and rehashing to avoid them. With a small number of extra bits per pointer, we can make the number of hash conflicts small and then simply not use these entries of the dictionary (thus "wasting" a small fraction of the dictionary).

A small constant $c \geq 1$ extra bits are added to each pointer so that the $|D_{\max}|$ dictionary entries are hashed into a space of $|D_{\max}|2^c$ indices. Consider a particular index i that is used for the first time in a sequence of n pairs that are hashed into the dictionary. The probability that all other pairs do *not* hash to index i is

$$
\begin{aligned}
&\geq \left(1 - \frac{1}{n2^c}\right)^{n-1} \\
&> \left(1 - \frac{1}{n2^c}\right)^{n} \\
&= \left(\left(1 - \frac{1}{n2^c}\right)^{n2^c}\right)^{2^{-c}} \\
&\geq \left(\frac{1}{e(n)}\right)^{2^{-c}}, \quad \text{where } e(n) = \left(\frac{n}{n-1}\right)^{n}, \\
&= \frac{\left(\frac{1}{e}\right)^{2^{-c}}}{\left(\frac{e(n)}{e}\right)^{2^{-c}}} \\
&> \frac{1 - (2^{-c})}{\left(\frac{e(n)}{e}\right)^{2^{-c}}}, \quad e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad \text{and for all } 0 < x < 1, \quad \frac{1}{e^x} > 1 - x.
\end{aligned}
$$

Hence the probability that two pairs hash to index i is

$$
\begin{aligned}
&\leq 1 - \frac{1 - (2^{-c})}{\left(\frac{e(n)}{e}\right)^{(2^{-c})}} \\
&< 2^{-c} + \left(1 - \left(\frac{e}{e(n)}\right)^{(2^{-c})}\right) \quad \text{since for all } n > 1,\ e(n) > e \\
&= 2^{-c} + \varepsilon(n) \quad \text{since } e = \lim_{n \to \infty} e(n),
\end{aligned}
$$

where $\varepsilon(n)$ goes to 0 as n goes to ∞. Thus the expected number of hash conflicts is less than

$$n(2^{-c} + \varepsilon(n)),$$

and by the Chernoff bound it follows that the probability that there are more than εn hash conflicts is

$$< \left(\frac{\varepsilon}{e(2^{-c} + \varepsilon(n))}\right)^{-\varepsilon n}.$$

For any constant $d > 1$, this expression can be made less than $d^{-\varepsilon n}$ if c is made sufficiently large (by choosing n sufficiently large).

Given that the probability of more than εn hash conflicts can be made arbitrarily low, a simple approach is the conservative strategy of "throwing out" all indices that correspond to a hash conflict (even though many or all of these entries may not cause any problems for the decoder). This construction adds c extra bits to each pointer, which can be made to have an insignificant effect on compression by using a very large dictionary (and hence a very large pointer size) but will most likely be significant for dictionary sizes that are used in practice. However, this effect may not be unacceptable. For example, if we take $|D_{\max}| = 2^{16}$ (a common and practical choice for data compression) and $\varepsilon = 0.1$ (again, a reasonable value in practice), then we must choose $c \geq 5$; taking $c \geq 5$ and computing the error term $\varepsilon(n)$, we see that the probability that there are more than $\varepsilon(n)$ hash conflicts is $< 2^{-1,500}$. However, the cost for this security against many hash conflicts is an extra five bits for every 16-bit pointer. If we consider a typical example for lossless compression using a dictionary of size 2^{16}, we might expect the compressed data to be 30 percent of the size of the original data, but now with the extra five bits per pointer, the compressed size will be $\frac{21}{16}30 \approx 39$ percent of the original size. Although this cost might be reasonable in practice, it would be nice to avoid it. We conjecture that this cost can be reduced by a tighter analysis that does not throw out all indices with hash conflicts but rather throws out only those that delay the dictionary-filling process due to "racing" and "thrashing" of counts. We leave such analysis (as well as the effect of hash conflicts on application-dependent performance issues such as compression ratio) as a subject for future research.

Appendix. Constant-redundant parsings are optimal. The UC heuristic is exactly the algorithm shown to be optimal in Ziv and Lempel [1978]; a nice proof appears in Cover and Thomas [1991]. The essence of this proof is to show that *any* method that parses the input stream into distinct phrases (and sends a number of bits equal to $\log_2$ of the current number of phrases to the decoder) must be optimal. This appendix notes how this notion can be generalized and still maintain optimality.

DEFINITION. *A* τ-redundant parsing *of the input stream is one with at most* τ *copies of any given phrase.*

LEMMA. τ*-redundant parsings give optimal data compression for the UC model.*

Proof. We refer to the presentation in section 12.10 of Cover and Thomas [1991] and describe how slight modifications can be made to the formulas. Note that here $c(n)$ denotes the total number of phrases (with repetition), whereas in the original presentation phrases are not repeated. Lemmas 12.10.1 and 12.10.2 do not change if the parsing is τ-redundant. Lemma 12.10.3 (Ziv's inequality) gets a factor of τ inside the log on the right side. (The proof is essentially unchanged except that the 1 in the log on the right side in equation (12.286) becomes τ.) Theorem 12.10.1 (the main theorem) still holds for a τ-redundant parsing; the proof is essentially the same, with the following small changes: A factor of τ goes in the denominator inside the log on the right side of equation (12.288), which is equivalent to adding the term $\tau \log(\rho)$ to the right side of equation (12.288). This term of $c \log(\tau)$ is added to the right side of equation (12.292) and subtracted from the right side of equation (12.293). Finally, the error term $\varepsilon_k(n)$ in equation (12.300) is changed to have $(c/n)\log(\tau)$ added to it; since c is $O(n/\log(n))$ by Lemma 12.10.1, the error term still goes to 0 as n goes to ∞.

COROLLARY. *The lemma holds for the FC and NC heuristics as well.*
Proof. These heuristics can use a phrase at most $\tau|\Sigma|$ times.

REFERENCES

J. L. CARTER AND M. N. WEGMAN [1979], *Universal classes of hash functions*, J. Comput. System Sci., 18, pp. 143–154.

T. M. COVER AND J. A. THOMAS [1991], *Elements of Information Theory*, John Wiley, New York.

R. G. GALLAGER [1968], *Information Theory and Reliable Communication*, John Wiley, New York.

T. HAGERUP AND C. RUB [1989], *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33, pp. 305–308.

A. LEMPEL AND J. ZIV [1976], *On the complexity of finite sequences*, IEEE Trans. Inform. Theory, 22, pp. 75–81.

A. LEMPEL AND J. ZIV [1990], private communication.

V. S. MILLER AND M. N. WEGMAN [1985], *Variations on a theme by Lempel and Ziv*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Springer-Verlag, Berlin, pp. 131–140.

J. REIF AND J. A. STORER [1990], *A parallel architecture for high speed data compression*, in Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, IEEE Press, Los Alamitos, CA, pp. 238–243.

J. REIF AND J. A. STORER [1992], *A parallel architecture for high speed data compression*, J. Parallel Distrib. Comput., 13, pp. 222–227.

D. M. ROYALS, T. MARKAS, N. KANAPOULOS, J. H. REIF, AND J. A. STORER [1993], *On the design and implementation of a lossless data compression and decompression chip*, IEEE J. Solid-State Circuits, 28, pp. 948–953.

J. B. SEERY AND J. ZIV [1977], *A universal data compression algorithm: Description and preliminary results*, Technical Memorandum 77-1212-6, Bell Laboratories, Murray Hill, NJ.

J. B. SEERY AND J. ZIV [1978], *Further results on universal data compression*, Technical Memorandum 78-1212-8, Bell Laboratories, Murray Hill, NJ.

J. A. STORER [1988], *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD.

J. A. STORER AND T. G. SZYMANSKI [1978], *The macro model for data compression*, in Proc. 10th Annual ACM Symposium on the Theory of Computing, ACM, New York, pp. 30–39.

T. A. WELCH [1984], *A technique for high-performance data compression*, IEEE Comput., 17, pp. 8–19.

J. ZIV AND A. LEMPEL [1977], *A universal algorithm for sequential data compression*, IEEE Trans. Inform. Theory, 23, pp. 337–343.

J. ZIV AND A. LEMPEL [1978], *Compression of individual sequences via variable-rate coding*, IEEE Trans. Inform. Theory, 24, pp. 530–536.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 950–960, August 1997

003

CONSTANT-TIME RANDOMIZED PARALLEL STRING MATCHING*

MAXIME CROCHEMORE†, ZVI GALIL‡, LESZEK GĄSIENIEC§, KUNSOO PARK¶, AND WOJCIECH RYTTER‖

Abstract. Given a pattern string of length m for the string-matching problem, we design an algorithm that computes deterministic samples of a sufficiently long substring of the pattern in constant time. This problem used to be the bottleneck in the pattern preprocessing for one- and two-dimensional pattern matching. The best previous time bound was $O(\log^2 m/\log\log m)$. We use this algorithm to obtain the following results (all algorithms below are optimal parallel algorithms on a CRCW PRAM):

1. a deterministic string-matching algorithm which takes $O(\log\log m)$ time for preprocessing and constant time for text search, which are the best possible in both preprocessing and text search;

2. a constant-time deterministic string-matching algorithm in the case where the text length n satisfies $n = \Omega(m^{1+\epsilon})$ for a constant $\epsilon > 0$;

3. a simple string-matching algorithm that has constant time with high probability for random input;

4. the main result: a constant-expected-time Las Vegas algorithm for computing the period of the pattern and all witnesses and thus for string matching itself; in both cases, an $\Omega(\log\log m)$ lower bound is known for deterministic algorithms.

Key words. parallel string matching, randomized algorithms, deterministic samples

AMS subject classifications. 68Q22, 68Q25, 68R15

PII. S009753979528007X

1. Introduction.

The string-matching problem is defined as follows: Given pattern $P[0..m-1]$ and text $T[0..n-1]$, find all occurrences of P in T. We study parallel complexity of string matching on a CRCW PRAM. The PRAM (parallel random-access machine) is a shared-memory model of parallel computation which consists of a collection of identical processors and a shared memory. Each processor is a RAM working synchronously and communicating via the shared memory. The CRCW (concurrent-read concurrent-write) PRAM allows both concurrent reads and concurrent writes to a memory location, and it has several variants depending on how concurrent writes are handled. We use the weakest version (called *common* in [7]), in which the only concurrent writes allowed are of the same value, 1. A parallel

* Received by the editors January 11, 1995; accepted for publication (in revised form) August 3, 1995.

http://www.siam.org/journals/sicomp/26-4/28007.html

† Institut Gaspard Monge, Université de Marne-la-Vallée, 2 Rue de la Butte Verte, F-93160 Noisy le Grand, France (mac@univ-mlv.fr). The research of this author was supported in part by PRC "Algorithmique, Modèles, Infographie" and GREG "Motifs dans les séquences."

‡ Department of Computer Science, Columbia University, New York, NY 10027 (galil@cs.columbia.edu) and Tel-Aviv University, Tel-Aviv, Israel. The research of this author was supported in part by NSF grant CCR-90-14605 and CISE Institutional Infrastructure grant CDA-90-24735.

§ Max-Planck Institut für Informatik, Im Stadtwald, Saarbrücken D-66123, Germany (leszek@mpi-sb.mpg.de). The research of this author was done during the author's stay at Instytut Informatyki, Uniwersytet Warszawski, 02-097 Warszawa, Poland and was supported in part by KBN grant 2-11-90-91-01 and EC Cooperative Action IC-1000 (project ALTEC).

¶ Department of Computer Engineering, Seoul National University, Seoul 151-742, Korea (kpark@theory.snu.ac.kr). The research of this author was supported by KOSEF grant 951-0906-069-2.

‖ Instytut Informatyki, Uniwersytet Warszawski, 02-097 Warszawa, Poland (rytter@mimuw.edu.pl) and Department of Computer Science, Liverpool University, United Kingdom. The research of this author was done partly while visiting the University of California at Riverside and was supported in part by KBN grant 8T11C01208.

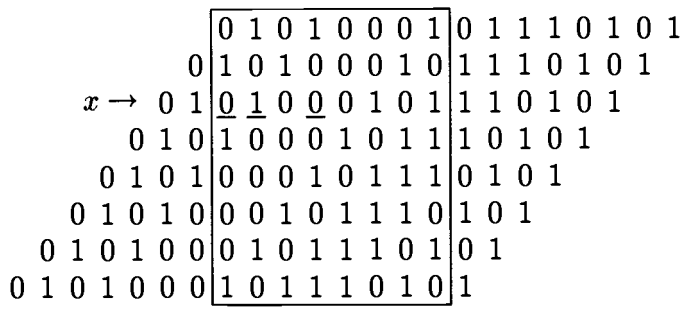


FIG. 1. *A 3-size DS of x for 8 shifts: $A = \{2, 3, 5\}$ and $f = 6$.*

algorithm for a problem is *optimal* if its total work is asymptotically the same as the minimum possible work for the problem. All optimal algorithms in this paper have linear work. Most string-matching algorithms consist of two stages. The first preprocesses the pattern and the second uses the data structure constructed in the first to search the text. For the text search, a constant-time optimal parallel algorithm (following optimal $O(\log^2 m/\log\log m)$-time preprocessing) is known [8]. On the other hand, an $\Omega(\log\log m)$ lower bound with a linear number of processors is known for the entire string-matching problem [3].

Let $x[0..r-1]$ be a string of length r. String $x[0..p-1]$, $1 \le p < r$, is a *period* of x if $x[i] = x[i+p]$ for all $0 \le i < r-p$. The shortest period of x is called *the period* of x. We also use the term period for the *length* of the corresponding string. If the period of x is shorter than a half (fourth) of x, x is called *periodic* (4-*periodic*). A *witness* of x against (the periodicity of) i is a position w such that $x[w] \neq x[w-i]$ [12]. Let p be the period of x. When we say that we compute the period of x, we mean computing $\min(p, r/4)$. When we say that we compute the witnesses of x, we mean computing the witnesses against all nonperiods i, $1 \le i < r/4$. The witnesses of x can be computed in optimal $O(\log\log r)$ time by [2]. (It is possible to compute large periods and witnesses using techniques of [1], but we will not need them here.) Given two strings x and y and a position i of y such that x does not occur at position i of y, a *witness to* nonoccurrence at i is a position w such that $y[w] \neq x[w-i]$. A substring of x of length i is called an i-*block* of x. The positions of all strings in this paper start from 0.

Consider a nonperiodic pattern string x of length m. Align $k \le m/2$ copies of x one on top of the other so that the ith copy starts above the ith symbol of the first copy. A deterministic sample (DS) of x for k shifts is an ordered set A of positions and a number f, $1 \le f \le k$, such that $f-1$ consecutive copies of x to the left and $k-f$ consecutive copies to the right have at least one mismatch with copy number f of x in the positions of the set A. The size of a DS is the size of the ordered set A. See Fig. 1. Vishkin [13] introduced the notion of deterministic samples and proved the existence of a DS of size at most $\log m$ for $m/2$ shifts.

The DS is crucial for very fast optimal parallel search of the pattern in a given text. During the text search, we maintain a subset of text positions, referred to as *candidates*, which can be start positions of pattern occurrences. Assume that we can somehow reduce the number of candidates to one in every $\log m$-block of the text. Then for every candidate, we compare the symbols at the positions of the set A of the DS with the corresponding symbols of the text. If a candidate has mismatches, it is no

longer a candidate for an occurrence of the pattern. On the other hand, if a candidate has matches in all the positions of A, then by the definition of the DS, we can eliminate all other candidates in an $m/2$-block of the text. This method was used in a constant-time optimal text search [8]. Very recently, it was also used in a constant-time two-dimensional text search [5]. However, the optimal algorithm suggested by Vishkin and used in [8] for computing the DS was very expensive, taking $O(\log^2 m/\log\log m)$ time. This resulted in two "best" algorithms for string matching: an optimal $O(\log\log m)$-time algorithm for the entire problem [2] (for which an $\Omega(\log\log m)$ lower bound was also proved [3]) and a constant-time text search with expensive preprocessing that was dominated by the computation time of the DS. Thus the DS computation has been the bottleneck.

In section 2, we present a constant-time deterministic algorithm for computing a DS of logarithmic size. We also show how to compute a constant-size DS for $O(\log\log m)$ shifts in constant time. This constant-size DS will be crucial to our main result in section 4. Since we compute deterministic samples for a part of the pattern, we can use more than a linear number of processors.

Section 3 contains three applications of the constant-time algorithm for DS. The first application is that it allows us to have only one best string-matching algorithm with constant-time search and $O(\log\log m)$-time preprocessing. Our new algorithm achieves the best possible time in both preprocessing and text search. The second application is a deterministic $O(k)$-time string-matching algorithm using n processors for the case where $m = O(n^{1-2^{-k}})$, i.e., a constant-time string-matching algorithm using n processors when $n = \Omega(m^{1+\epsilon})$ for a constant $\epsilon > 0$. The third application is a simple string-matching algorithm that has constant time with high probability (and thus constant expected time) for random input.

In section 4, we describe our main result. We present a constant-expected-time Las Vegas algorithm for computing the witnesses. Together with the constant-time text search, we obtain a constant-expected-time Las Vegas algorithm for string matching including preprocessing, solving the main open problem remaining in parallel string matching. Deterministically, an $\Omega(\log\log m)$ lower bound is known for witness computation and string matching [3]. This algorithm is designed based on the lower-bound argument; randomization is used to kill the argument. In the special case where the pattern is periodic and the period of the pattern has only a constant number of prime divisors, randomization is not needed.

Our algorithms will frequently use without mention the constant-time algorithm that finds the maximum (or minimum) position of a nonzero entry in an array [7]. Our algorithms will use the constant-time deterministic polynomial approximate compaction (PAC) of Ragde [11] and its improvement by Hagerup [9]. A d-PAC is an algorithm that compacts an array of size n with at most m nonzero elements into a prefix of size m^d (assuming that $m^d < n$). Ragde gave a $(4+\epsilon)$-PAC and Hagerup gave a $(1+\epsilon)$-PAC for any $\epsilon > 0$.

In many places where we use quantities such as $r/2$, $\log r$, or $\log\log r$ as integers, we mean that any way of rounding them to the nearest integer will do.

2. Constant-time deterministic sampling. Let x be a nonperiodic string of length r. We construct two kinds of DSs in constant time: a $\log k$-size DS for k shifts, $k \le r/2$, and a constant-size DS for $\log\log r$ shifts. We first show how to construct a $\log k$-size DS of x for k shifts, $k \le r/2$, in constant time using r^3 processors and r^2 space. This log-size DS was introduced by Vishkin [13], but its construction takes $O(\log^2 r/\log\log r)$ time using $O(r)$ operations, which is the bottleneck in the

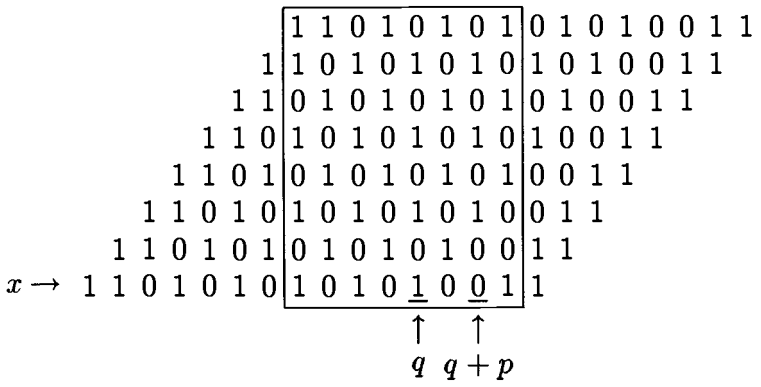


FIG. 2. *A DS with* $A = \{q, q+p\}$ *and* $f = 1$.

preprocessing of string matching [13, 8]. Consider k-blocks starting at positions i for $0 \le i < k$. If two k-blocks are identical, we say that x *has a periodicity*. Note that x has a periodicity if and only if there are i and j for $i < j < k$ (the start positions of the two identical blocks) such that $x[i..j+k-1]$ has a period $p = j - i < k \le r/2$. Using $k^3 < r^3$ processors, the algorithm checks in constant time if x has a periodicity. If it does, the algorithm finds such i, j, and $p = j - i$. p is not necessarily the smallest such period.

Case 1: x has a periodicity in $x[i..j+k-1]$ with period p. Although p is a period of $x[i..j+k-1]$, p cannot be a period of x since x is nonperiodic. That is, the periodicity with period p cannot extend both all the way to the right and all the way to the left of $x[i..j+k-1]$. If it does not extend to the right, let $q > j$ be the smallest position such that $x[q] \ne x[q+p]$ (i.e., end of periodicity). The position q can be found in constant time with r processors. Now the DS is $A = \{q, q+p\}$ and $f = 1$ because in the first copy we have mismatching symbols at positions $q, q+p$ and in the next $k-1$ copies to the right we have matching symbols at the same positions. See Fig. 2, where $k = 8$, $i = 2$, and $j = 4$ ($p = j - i = 2$). If the periodicity extends all the way to the right, let $q < i$ be the largest position such that $x[q] \ne x[p+q]$. The DS is $A = \{q, q+p\}$ and $f = k$.

Case 2: x does not have a periodicity (i.e., all of the k-blocks are distinct). Consider (for discussion only) the compacted prefix tree T of all the blocks (each path from the root of T to a leaf corresponds to a block and every internal node has degree at least 2). Since T has k leaves, there is at least one leaf v of depth $\le \log k$. Let B be the block corresponding to v. The path in T from the root to v hits at most $\log k$ nodes which define at most $\log k$ positions; B is different from each of the other blocks in at least one of these positions. Below we will find such a block B and the (at most $\log k$) positions in B in constant time using r^3 processors and r^2 space. Let b be the start position of B in x. The DS is the derived set of positions in B shifted by b and $f = k - b$. For example, consider Fig. 1. For the block $B = 01000101$, its start position in x is 2 and f is 6.

To find B and the positions in it, we compute a $k \times k$ 0–1 matrix: one row for each block. The matrix is initially set to 0. With r processors per each pair (i, j), $1 \le i, j \le k$, of blocks, find in constant time the smallest index ℓ such that the ith block and the jth block differ at position ℓ (a node in T). Note that we find in this

way exactly all nodes of T (more than once). Set entry ℓ in the two rows i and j to 1. Now we only have to find a row with no more than $s = \log k$ 1's and compress their positions to an array of size s. So we need to solve the following problem for each row of the matrix. Given a 0–1 array of size $k \leq r$ and r^2 processors, find whether it has at most $s = \log k$ 1's and, if it does, compress their positions into an array of size s.

Ragde designed a $(4 + \epsilon)$-PAC [11] and then used it to compress an array of length k with at most s items (= nonzero entries) into an array of size s in time $O(\log s / \log \log k)$. In case the number of items is larger than s, the algorithm fails. Note that when $\log s = O(\log \log k)$, the time is $O(1)$. So to solve the problem above, first the jth processor replaces a 1 in the jth entry with j and then Ragde's compression is applied in constant time. This compression will succeed with at least one of the rows of the matrix and will yield the desired DS.

THEOREM 2.1. *The deterministic algorithm above constructs a* $\log k$*-size DS for* k *shifts,* $k \leq r/2$*, for a nonperiodic string of length* r *in constant time using* r^3 *processors and* r^2 *space.*

Although the DS computation used to be the bottleneck, the algorithm in [8] has another part of the preprocessing (the hitting set) that does not take constant time: the part that enables the algorithm to eliminate all but at most one candidate in every $\log m$-block. The hitting set can be constructed in $O(\log \log m)$ time. Using Theorem 1 and an $O(\log \log m)$-time construction of the hitting set, the algorithm in [8] can be transformed into a string-matching algorithm which takes constant time for text search and $O(\log \log m)$ time for preprocessing. However, in order to derive a randomized constant-time algorithm, we cannot afford to compute a hitting set. Instead of the hitting set, we use the following constant-size DS for $O(\log \log m)$ shifts to design an alternative algorithm called CONST-MATCH in section 3.

We now show how to construct a constant-size DS of x for $\log \log r$ shifts in constant time with $r^2 \log \log r$ processors. This constant-size DS is new and is crucial for constant-time randomized string matching in section 4 as discussed above.

Case 1. If there exists position i in x such that $x[i] \neq x[i + j]$ (or $x[i] \neq x[i - j]$) for every $1 \leq j < \log r$, then we take $A = \{i\}$ and $f = \log r$ (or $A = \{i\}$ and $f = 1$) as the DS for $\log r$ shifts (and therefore for $\log \log r$ shifts as well).

Case 2. Otherwise, every symbol in x occurs very often (with distance shorter than $\log r$ between neighboring ones). So every symbol occurs at least $r/\log r$ times in x, which implies that there are at most $\log r$ different symbols in x. Consider all substrings of length $\log \log r$ in the first half of x. Since there are $(\log r)^{\log \log r}$ different strings of length $\log \log r$ over $\log r$ symbols and since $(\log r)^{\log \log r} < r/(2 \log \log r)$, some substrings of length $\log \log r$ repeat without overlap in the first half of x. Find such a substring y in constant time using $r^2 \log \log r$ processors. Let z be the substring between the two copies of y. The substring yzy has a period $p = |yz| < r/2$. Since x is nonperiodic, period p has a mismatch in x. Let q be the smallest (largest) position such that $x[q] \neq x[q + p]$ to the right (left) of the first copy of y. Then $A = \{q, q + p\}$ and $f = 1$ ($A = \{q, q + p\}$ and $f = \log \log r$) is a constant-size DS for $\log \log r$ shifts.

THEOREM 2.2. *The deterministic algorithm above constructs a constant-size DS for* $\log \log r$ *shifts for a nonperiodic string of length* r *in constant time using* $r^2 \log \log r$ *processors.*

3. Applications of constant-time DS. In this section, we present applications of the constant-time DS computation, and at the same time, we build up procedures which will be used in the constant-time randomized string-matching algorithm in section 4. Thanks to a well-known reduction [2], we can assume without loss of

generality that the pattern in a string matching problem is non-4-periodic.

The text-search algorithm will maintain candidates, which can still be start positions of pattern occurrences. All other positions have gotten witnesses to nonoccurrences. Consider two candidates $i < j$ such that w is a witness against $j - i$ (i.e., $P[w] \neq P[w - j + i]$).

(a) If $T[i + w] \neq P[w]$, $i + w$ is a witness to nonoccurrence at i.
(b) If $T[i + w] \neq P[w - j + i]$, $i + w$ is a witness to nonoccurrence at j.

The two tests above are called a *duel* between i and j [12]. By a duel, we can remove one or both of the candidates. Given $h > 0$, we partition the text T into disjoint h-blocks in the obvious way. If every h-block has at most two candidates, we say that T is *h-good*.

LEMMA 3.1. *If T is h-good and an h-size DS for k shifts is given, then T can be made k-good in optimal constant time.*

Proof. Let A be the ordered set of the h-size DS. For each k-block, there are initially at most $2k/h$ candidates and we have $h/2$ processors per candidate. For each such candidate in the k-block, make h comparisons with the positions of A. If a candidate has a mismatch, it provides a witness against the candidate. Find the leftmost (ls) and rightmost (rs) survivors in the k-block. By the definition of the DS, every survivor i between ls and rs has at least one mismatch in the DS positions of ls or rs. For each such i, make $2h$ comparisons with the DS positions of ls and rs and find a witness against it. □

LEMMA 3.2. *If T is $m^{1/k}$-good for $k > 1$ and the witnesses of the pattern are given, T can be made $m/4$-good in time $O(\log k)$ with $O(n \log k)$ operations.*

Proof. Run $\log k$ rounds of the following until T is $m/4$-good. In a round, we start with at most two candidates per i-block (i-good) and end with at most one per i^2-block (i^2-good) by performing at most $4i^2$ duels in each i^2-block. Duels find witnesses to nonoccurrences. (Note that we actually start each round after the first with at most one candidate per i-block.) □

Given a string x of length r and a number $\ell \leq 2\sqrt{r}$, FIND-SUB finds the first nonperiodic substring z of length ℓ and computes witnesses of z if such z exists, and it otherwise computes the period p of x of length less than $\ell/2$ and witnesses against nonmultiples of p in optimal constant time.

PROCEDURE FIND-SUB:

1. Naïvely check if each of the first $\ell/2$ positions is a period of the prefix of x of length ℓ and compute witnesses against nonperiods.
2. If none is a period, z is the prefix of x of length ℓ. Stop.
3. If there are periods, find the shortest one p. Find the smallest prefix y of x such that p is not the period of y.
4. If p is the period of x (y does not exist), witnesses against nonmultiples of p are easily computed from the witnesses of the first p-block. Stop.
5. Otherwise (y exists, i.e., there is a mismatch with period p), z is the suffix of y of length ℓ. (z is nonperiodic [8].) Naïvely compute the witnesses of z.

The first application of the constant-time DS computation is a simple string-matching algorithm with constant-time search and $O(\log \log m)$-time preprocessing called CONST-MATCH. In order to be used in section 4, CONST-MATCH solves the following problem: Given a (non-4-periodic) pattern P of length m and its witnesses and text T, find all occurrences of P in T and witnesses to nonoccurrences in optimal constant time. Initially, T is 2-good.

PROCEDURE CONST-MATCH:

1. Find the first nonperiodic substring x of P of length $r = m^{1/3}$ and the first nonperiodic substring x' of x of length $2 \log r$ using FIND-SUB, which also computes witnesses of x and x'. Steps 2–4 use Lemma 1.
2. Use the constant-size DS of x for $\log \log r$ shifts to make T $\log \log r$-good.
3. Use the $\log \log r$-size DS of x' for $\log r$ shifts to make T $\log r$-good.
4. Use the $\log r$-size DS of x for $r/2$ shifts to make T $r/2$-good.
5. Perform two rounds of duels to make T $m/4$-good (Lemma 2). Then check the surviving candidates by naïve comparisons.

THEOREM 3.3. *Given the witnesses of the pattern, procedure* CONST-MATCH *performs string matching in optimal constant time.*

COROLLARY 3.4. *Using* CONST-MATCH*, we get a string-matching algorithm with constant-time text search and* $O(\log \log m)$*-time preprocessing.*

The second application is a deterministic $O(k)$-time algorithm for finding a subpattern of length $m^{1-2^{-k}}$ in the text. This application may seem somewhat contrived, but it is useful in the next two applications. We will twice use the case $k = 1$. Let P' be a subpattern of P of length $m^{1-2^{-k}}$. We first compute witnesses of P' in $O(k)$ time by the first k rounds of the preprocessing of [2] and then find all occurrences of P' in the text by CONST-MATCH. (In case P' is 4-periodic, we use the reduction of [2].)

COROLLARY 3.5. *Using* CONST-MATCH*, we get an* $O(k)$*-time algorithm using* n *processors for finding all occurrences of a given subpattern of length* $m^{1-2^{-k}}$ *in the text, or, stated differently, we get a deterministic* $O(k)$*-time algorithm using* n *processors for string matching in case* $m = O(n^{1-2^{-k}})$.

The third application is a simple string-matching algorithm that has constant time with high probability for random input. Let P' be the prefix of the pattern of length $\sqrt{m}$. Compute witnesses of P' in constant time. Find all occurrences of P' in the text by CONST-MATCH. Being able to match P' of length $\sqrt{m}$ in constant time gives us a constant-expected-time parallel algorithm for random text even if we sequentially check the remaining symbols. The probability that the time is larger than some small constant is exponentially small.

COROLLARY 3.6. *Using* CONST-MATCH*, we get a simple string-matching algorithm that has constant time with high probability for random input.*

4. Constant-time randomized string matching. The main application of the constant-time DS computation is a constant-expected-time Las Vegas algorithm for computing the witnesses of the pattern. Together with CONST-MATCH, we obtain a constant-expected-time Las Vegas algorithm for string matching including preprocessing, solving the main open problem remaining in string matching. Thus, randomization is used to "beat" the deterministic $\Omega(\log \log m)$ lower bound for witness computation and string matching [3].

We introduce a notion of *pseudoperiod* (also used in [4]). It has an operational definition: Given a string x of length r, if we compute witnesses against all $i < r/4$ except for multiples of q, we say that q is a pseudoperiod of x. It follows from this definition that if x is 4-periodic, q must divide the period of x. Procedure FIND-PSEUDO computes a large pseudoperiod of x.

PROCEDURE FIND-PSEUDO:

1. Run FIND-SUB with x and $\ell = 2\sqrt{r}$. If the period q of x is $< \sqrt{r}$, stop. Otherwise, FIND-SUB finds the first nonperiodic substring z of x of length $2\sqrt{r}$ and computes witnesses of z.

2. Using CONST-MATCH, find all occurrences of z in x and witnesses to nonoccurrences.
3. Construct the $(r-2\sqrt{r})$-bit binary string x' such that for $0 \le i < r-2\sqrt{r}$, $x'[i] = 1$ if i is an occurrence of z and $x'[i] = 0$ otherwise. Compute the period q of x' in the case where $q < r/4$ and, in addition, all witnesses of x' against nonperiods of x' smaller than $r/4$. Note that if $q < r/4$, all periods of x' smaller than $r/4$ are multiples of q. This computation exploits the special form of x'; it contains at most $\sqrt{r}$ 1's with distance of at least $\sqrt{r}$ between them since z is nonperiodic and of length $2\sqrt{r}$. Thus we can compute witnesses of x' by considering only the 1's.
   3.1. Divide string x' into disjoint $\sqrt{r}$-blocks.
   3.2. There is at most one 1 in every block. Record the position of the 1 in the given block in the first element of that block. (Now every processor can read from the first element of a block the position of 1 in that block.)
   3.3. Let t be the position of the first 1 in x'. For position $i < t$, t is a witness against $t-i$. If $t \ge r/4$ this substep is done. Note that we already have all witnesses against $i < r/4$. Otherwise, consider $i \ge 2t$ (i.e., $i-t \ge t$). If $x'[i] = 0$, then i is a witness against $i-t$. If $x'[i] = 1$, $i-t$ is a potential period of x' since it shifts the first 1 to a 1. Use the $\sqrt{r}$ processors of the block of i to check if $i-t$ is a real period of x' by checking for all k such that $x'[k] = 1$ whether ($x'[k+i-t] = 1$ or $k+i-t \ge r-2\sqrt{r}$) and ($x'[k-i+t] = 1$ or $k-i+t < 0$). If all of these tests succeed, $i-t$ is a period of x'. If the test with k fails, $k+i-t$ or k is a witness against $i-t$. Compute q, the smallest period of x'.
   3.4. From witnesses of x', compute witnesses of x. Let w be the witness of x' against i. Assume that $x'[w] = 0$ and $x'[w-i] = 1$. (The other case is similar.) Since $x'[w] = 0$, w is a nonoccurrence of z in x. Let j be the witness to nonoccurrence of z at w. One can verify that $w+j$ is a witness of x against i.

Procedure FIND-PSEUDO computes q, which satisfies the following:

P1. If $q \le \sqrt{r}$, q is the real period of x.
P2. If $q > \sqrt{r}$, then q is a pseudoperiod of x.

Given q integers, let $\mathrm{LCM}_{k,q}$ be the minimum of $k/4$ and the LCM (least common multiple) of the q numbers. Given a $k \times q$ array B of symbols and for every column c its pseudoperiod $q_c < k/4$ and witnesses against nonmultiples of q_c, procedure FIND-LCM computes $\mathrm{LCM}_{k,q}$ of the pseudoperiods and witnesses against nonmultiples of $\mathrm{LCM}_{k,q}$ smaller than $k/4$ in constant time with kq processors as follows.

Procedure FIND-LCM:

1. Construct a $(k/4-1) \times q$ array B': in the cth column of B', write 1 in the multiples of the pseudoperiod q_c and 0 in other places.
2. For each row that is not all 1's, any 0 entry provides a witness against the row number.
3. If there is a row with all entries 1's, return the smallest such row; otherwise, return $k/4$.

Let p be the period of the pattern P. Recall that the main problem is to compute $\min(p, m/4)$ and the witnesses against all nonperiods i, $1 \le i < m/4$. These nonperiods are exactly all the nonmultiples of p smaller than $m/4$. In the 4-periodic case, witnesses against all i, $1 \le i < p$, are sufficient. The other witnesses against $i < m/4$ can be computed from them in constant time.

We first describe a deterministic $O(\log\log m)$-time algorithm for the main problem. The algorithm consists of rounds and maintains a variable q. The invariant at the end of a round is that q is a pseudoperiod of x. Initially, $q = 1$. We describe one round of the algorithm. A witness against i found during the round is a witness of P against iq.

1. Divide P into blocks of size q and make an array B of $k = m/q$ rows and q columns, where column j contains all $P[i]$ for all $i \equiv j \bmod q$.
2. For each column c of B, find q_c, its pseudoperiod, and witnesses against nonmultiples of q_c using FIND-PSEUDO.
3. If all pseudoperiods are $\leq \sqrt{k}$, all pseudoperiods are real periods. Using FIND-LCM, compute $\mathrm{LCM}_{k,q}$ and witnesses against nonmultiples of $\mathrm{LCM}_{k,q}$ smaller than $k/4$. The period of P that we compute is $q \cdot \mathrm{LCM}_{k,q}$. Stop.
4. Otherwise, choose a column c with $q_c > \sqrt{k}$. Witnesses against nonmultiples of q_c were computed in step 2. $q \leftarrow q \cdot q_c$.
5. If $q < m/4$, then go to the next round; otherwise, stop.

Note that in the first round, we have one column and compute a pseudoperiod of P by FIND-PSEUDO. In subsequent rounds, $q \cdot q_c$ is a pseudoperiod because we compute witnesses for all nonmultiples of $q \cdot q_c$. Since the new value k is at most $\sqrt{k}$, there are at most $O(\log\log m)$ rounds.

This algorithm follows the structure of the lower-bound proof [3]. That proof uses the notion of "possible period length," which is the minimum number that can still be the period of the pattern based on the results of the comparisons so far. The lower-bound argument maintains a possible period length $q \leq m^{1-4^{-i}}$ in round i and forces any algorithm to have at least $\frac{1}{4}\log\log m$ rounds. Here we compute a pseudoperiod q that may not be a period length but must divide it in case the pattern is 4-periodic. $q > m^{1-2^{-i}}$ in round i and the algorithm finds the period in at most $\log\log m$ rounds.

COROLLARY 4.1. *If P is 4-periodic and its period has a constant number of prime divisors, we can compute witnesses and do string matching in optimal deterministic constant time.*

We now describe an $O(1)$-expected-time randomized algorithm for the main problem. Execute the first three rounds of the deterministic algorithm and then execute round 4 below until it stops. At the beginning of round 4, q is a pseudoperiod of x, and B is the $k \times q$ array, $k = m/q$, created by step 1 of the deterministic algorithm. We have $q > m^{7/8}$ and $k = m/q < m^{1/8}$.

Round 4:

1. Randomly choose a multiset of $s = m/k^2$ columns from B, i.e., each of s processors chooses a random column number from the set $\{1, \ldots, q\}$. Find the period of each chosen column naïvely with k^2 processors. Using naïve comparisons, also compute witnesses against nonperiods. Using FIND-LCM, compute $h = \mathrm{LCM}_{k,s}$ and witnesses against nonmultiples of h.
2. If $h = k/4$, the pattern P is not 4-periodic. Stop.
3. Otherwise, check if h is a period of each column of B. If h is a period in all columns, qh is the period of P; stop. Otherwise, let C be the set of columns where h is not a period.
4. Using Hagerup's $(1+\epsilon)$-PAC [9], try to compact C into the set C' of size $m^{3/4}$. If the compaction fails, again try round 4 starting from step 1.
5. If the compaction is successful, compute all periods of columns in C' naïvely (we have enough processors because $m^{3/4}k^2 < m$). Using naïve comparisons, also compute witnesses against nonperiods. Using FIND-LCM, compute $h' =$

$\mathrm{LCM}_{k,m^{3/4}}$ of these periods and witnesses against nonmultiples of h'. The period of P that we compute is $\min(m/4, q \cdot \mathrm{LCM}(h, h'))$.

LEMMA 4.2. *With a very high probability, the size of C in round 4 is smaller than $m^{1/2}$.*

Proof. Let Q be the multiset of the q real periods of the columns of B. We call a period *good* if it appears in Q at least $q^{1/3}$ times; otherwise call it *bad*.

For a good period, the probability that it is not chosen among the s random choices is at most $(1 - q^{1/3}/q)^s < e^{-m^{1/6}}$ since $s = m/k^2 = q^2/m > q^{2/3}m^{1/6}$. Thus the probability that there is some good period that was not chosen is at most $qe^{-m^{1/6}} < me^{-m^{1/6}} \ll 1$.

We showed that with a very high probability only the bad periods will remain in C. Since there are only k different values for periods, the number of occurrences of bad periods in Q is at most $q^{1/3}k = m/q^{2/3} < m/m^{7/12} < m^{1/2}$. □

It follows from Lemma 3 that the PAC will fail (and as a result round 4 will be repeated) with a very small probability and the expected number of rounds is smaller than five.

LEMMA 4.3. *The randomized algorithm is an optimal Las Vegas parallel algorithm for computing the period and the witnesses of the pattern P. It has constant time with high probability.*

THEOREM 4.4. *Together with* CONST-MATCH*, we have a randomized optimal Las Vegas parallel algorithm for string matching, including preprocessing. It has constant time with high probability (and thus constant expected time).*

5. Conclusion. We have shown how to compute deterministic samples for a part of the pattern in constant time, and we obtained a deterministic string-matching algorithm which is probably the best in both preprocessing and text search. We use them to obtain a simple constant-expected-time algorithm for random input and a more sophisticated randomized algorithm for string matching with constant expected time.

The randomized algorithm for string matching leads to constant-expected-time randomized algorithms for several related problems. If we solve $\log m$ (or even more) string-matching problems at the same time, the expected time is still a constant. This converts the algorithms for finding all periods, squares, and palindromes [1] into constant-expected-time randomized algorithms.

We believe that there may be more applications for our superfast deterministic sampling. The DS can be considered as a deterministic fingerprint. No constant-time algorithm is known for the conventional fingerprint computation on a CRCW PRAM. On the EREW PRAM, our algorithm can be translated into an optimal $O(\log m)$-time algorithm [6]; it works for any alphabet since it only performs comparisons on the input symbols. The conventional fingerprint has an optimal randomized $O(\log m)$-time algorithm on the EREW PRAM and it works only when the alphabet is given and is of small size [10]. On the other hand, our DS is only for a subpattern, and while it is computed deterministically, one still needs randomization for computing the witnesses. (The conventional fingerprint, while randomized, is applied deterministically.)

Acknowledgments. We thank Noga Alon and Yossi Matias for helpful suggestions.

REFERENCES

[1] A. Apostolico, D. Breslauer, and Z. Galil, *Optimal parallel algorithms for periods, palindromes and squares*, in Proc. 19th International Colloquium on Automata Languages and Programming, Lecture Notes in Comput. Sci. 623, Springer-Verlag, Berlin, 1992, pp. 296–307.

[2] D. Breslauer and Z. Galil, *An optimal $O(\log \log n)$ time parallel string matching algorithm*, SIAM J. Comput., 19 (1990), pp. 1051–1058.

[3] D. Breslauer and Z. Galil, *A lower bound for parallel string matching*, SIAM J. Comput. 21 (1992), pp. 856–862.

[4] B. S. Chlebus and L. Gasieniec, *Optimal pattern matching on meshes*, in Proc. 11th Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, Berlin, 1994, pp. 213–224.

[5] R. Cole, M. Crochemore, Z. Galil, L. Gasieniec, R. Hariharan, S. Muthukrishnan, K. Park, and W. Rytter, *Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions*, in Proc. 34th IEEE Symposium Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 248–258.

[6] A. Czumaj, Z. Galil, L. Gasieniec, K. Park, and W. Plandowski, *Work-time optimal parallel algorithms for string problems*, in Proc. 27th ACM Symposium on Theory of Computing, ACM, New York, 1994, pp. 713–722.

[7] F. E. Fich, P. Ragde, and A. Wigderson, *Relations between concurrent-write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[8] Z. Galil, *A constant-time optimal parallel string-matching algorithm*, J. Assoc. Comput. Mach., 42 (1995), pp. 908–918.

[9] T. Hagerup, *On a compaction theorem of Ragde*, Inform. Process. Lett., 43 (1992), pp. 335–340.

[10] R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Develop., (1987), pp. 249–260.

[11] P. Ragde, *The parallel simplicity of compaction and chaining*, J. Algorithms, 14 (1993), pp. 371–380.

[12] U. Vishkin, *Optimal parallel pattern matching in strings*, Inform. and Control, 67 (1985), pp. 91–113.

[13] U. Vishkin, *Deterministic sampling–a new technique for fast pattern matching*, SIAM J. Comput., 20 (1991), pp. 22–40.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 961–990, August 1997

004

LEARNING FROM MULTIPLE SOURCES OF INACCURATE DATA*

GANESH BALIGA†, SANJAY JAIN‡, AND ARUN SHARMA§

Abstract. Most theoretical models of inductive inference make the idealized assumption that the data available to a learner is from a *single* and *accurate* source. The subject of inaccuracies in data emanating from a single source has been addressed by several authors. The present paper argues in favor of a more realistic learning model in which data emanates from *multiple* sources, some or all of which may be *inaccurate*. Three kinds of inaccuracies are considered: spurious data (modeled as *noisy* texts), missing data (modeled as *incomplete* texts), and a mixture of spurious and missing data (modeled as *imperfect* texts).

Motivated by the above argument, the present paper introduces and theoretically analyzes a number of inference criteria in which a learning machine is fed data from multiple sources, some of which may be infected with inaccuracies. The learning situation modeled is the identification in the limit of programs from graphs of computable functions. The main parameters of the investigation are: the kind of inaccuracy, the total number of data sources, the number of faulty data sources which produce data within an acceptable bound, and the bound on the number of errors allowed in the final hypothesis learned by the machine.

Sufficient conditions are determined under which, for the same kind of inaccuracy, for the same bound on the number of errors in the final hypothesis, and for the same bound on the number of inaccuracies, learning from multiple texts, some of which may be inaccurate, is equivalent to learning from a single inaccurate text.

The general problem of determining when learning from multiple inaccurate texts is a restriction over learning from a single inaccurate text turns out to be combinatorially very complex. Significant partial results are provided for this problem. Several results are also provided about conditions under which the detrimental effects of multiple texts can be overcome by either allowing more errors in the final hypothesis or by reducing the number of inaccuracies in the texts.

It is also shown that the usual hierarchies resulting from allowing extra errors in the final program (results in increased learning power) and allowing extra inaccuracies in the texts (results in decreased learning power) hold.

Finally, it is demonstrated that in the context of learning from multiple inaccurate texts, spurious data is better than missing data, which in turn is better than a mixture of spurious and missing data.

Key words. inductive inference, machine learning, inaccurate data, multiple sources

AMS subject classifications. 68T05, 68T, 68, 68Qxx, 68Q

PII. S0097539792239461

1. Introduction.

A scenario in which an *algorithmic* learner attempts to learn its environment may be described thusly. At any given time, some finite data about the environment is made available to the learner. The learner reacts to this finite information by conjecturing a hypothesis to explain the behavior of its environment. The availability of additional data may cause the learner to revise its old hypotheses. The learner is said to be successful just in case the sequence of hypotheses it conjectures stabilizes to a final hypothesis which correctly represents its environment.

The above model, generally referred to as *identification in the limit*, originated in the pioneering work of Putnam [19], Gold [12], and Solomonoff [25, 26]. More recently, this model has been the subject of numerous studies in computational learning theory

* Received by the editors October 26, 1992; accepted for publication (in revised form) August 7, 1995.

http://www.siam.org/journals/sicomp/26-4/23946.html

† Computer Science Department, Rowan College of New Jersey, Mullica Hill, NJ 08024 (baliga@gboro.rowan.edu).

‡ Department of Information Systems and Computer Science, National University of Singapore, Singapore, Republic of Singapore 119260 (sanjay@iscs.nus.sg).

§ School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia (arun@cse.unsw.edu.au).

(see, e.g., [13, 20, 10, 27, 1]). A problem with this model is the idealized assumption that the data available to the learner is from a *single* and *accurate* source. The subject of inaccuracies in the data available to a learning machine has been previously studied by Schäfer-Richter [23], Fulk and Jain [11], Osherson, Stob, and Weinstein [17], and Jain [14, 15]. Each of these studies, however, also makes the assumption that the data available to the learner is from a single source. The present paper argues that in realistic learning situations, data available to a learner is from *multiple* sources, some of which may be *inaccurate*. We discuss these issues in the context of a specific learning scenario, namely, scientific inquiry modeled as identification of programs from graphs of computable functions. Although we present our results in the context of this particular learning task, we note that some of our arguments and techniques can be applied to other learning situations as well.

Consider a scientist S investigating a real-world phenomenon F. S performs experiments on F, noting the result of each experiment, while simultaneously conjecturing a succession of candidate explanations for F. A criterion of success is for S to eventually conjecture an explanation which S never gives up and which correctly explains F. Since we never measure a continuum of possibilities, we could treat S as performing discrete experiments x on F and receiving back experimental results $f(x)$. By using a suitable Gödel numbering, we may treat f associated with F as a function from N, the set of natural numbers, into N. Also, assuming a suitable neomechanistic viewpoint about the universe, f is computable. A complete and predictive explanation of F, then, is just a computer program for computing f. Thus algorithmic identification in the limit of programs for computable functions from their graph yields a plausible model for scientific inquiry.

Let us consider some common practices in scientific inquiry. Data is usually collected using different instruments, possibly at different places. (For example, astronomers use data from different telescopes situated at different locations.) In many cases, experimental errors may creep in or the instruments may simply be faulty. In some extreme cases, the same instrument may record conflicting readings at different times. Also, occasionally it may be infeasible to perform experiments (for example, determining the taste of cyanide). Moreover, the experimental findings of one scientist are generally available to others. All of this tends to suggest that a scientist often receives data from multiple sources, many of which are likely to be inaccurate. The present paper incorporates these observations into the standard learning model. We now proceed formally.

Section 2 presents the notation. Section 3 presents the preliminary notions about identification in the limit and inaccurate data. Section 4 introduces the main subject of this paper, viz, learning in the presence of multiple sources of inaccurate data. In this section, we also discuss some of our results informally. Section 5 presents our results with proofs.

2. Notation. The recursion-theoretic concepts not explained below are treated in [22]. N denotes the set of natural numbers, $\{0, 1, 2, 3, \ldots\}$, and N^+ denotes the set of positive integers, $\{1, 2, 3, \ldots\}$. $\in$, $\subseteq$, and $\subset$ denote, respectively, membership, containment, and proper containment for sets.

We let e, i, j, k, l, m, n, r, s, t, u, v, w, x, y, and z, with or without decorations,[1] range over N. We let a, b, and c, with or without decorations, range over $N \cup \{*\}$.

$[m, n]$ denotes the set $\{x \in N \mid m \leq x \leq n\}$. We let S, with or without

[1] Decorations are subscripts, superscripts, primes and the like.

decorations, range over subsets of N and we let A, B, C, and D, with or without decorations, range over finite subsets of N. $\min(S)$ and $\max(S)$, respectively, denote the minimum and maximum element in S ($\max(S)$ is undefined if S contains infinitely many elements). We take $\min(\emptyset)$ to be ∞ and $\max(\emptyset)$ to be 0. $\text{card}(S)$ denotes the cardinality of S. $*$ denotes *unbounded* but *finite*. We let $(\forall n \in N)[n < * < \infty]$. So then "$\text{card}(S) \leq *$" means that $\text{card}(S)$ is finite.

Let $\lambda x, y.\langle x, y\rangle$ denote a fixed pairing function (a recursive, bijective mapping: $N \times N \rightarrow N$) [22]. $\lambda x, y.\langle x, y\rangle$ and its inverses are useful for simulating the effect of having multiple argument functions. π_1 and π_2 are corresponding projection functions, i.e., $(\forall x, y)[\pi_1(\langle x, y\rangle) = x \ \wedge \ \pi_2(\langle x, y\rangle) = y]$.

η and ξ range over partial functions. For $a \in (N \cup \{*\})$, we say that η_1 is an a-variant of η_2 (written $\eta_1 =^a \eta_2$) iff $\text{card}(\{x \mid \eta_1(x) \neq \eta_2(x)\}) \leq a$. Otherwise, we say that η_1 is not an a-variant of η_2 (written $\eta_1 \neq^a \eta_2$). $\text{domain}(\eta)$ and $\text{range}(\eta)$, respectively, denote the domain and range of partial function η. Let $A \subseteq N$ and $c \in N$. We say that $\eta(A) = c$ iff for all $x \in A$, $\eta(x) = c$. If S_1 and S_2 are two sets, then $S_1 \mathbf{\Delta} S_2$ denotes $(S_1 - S_2) \cup (S_2 - S_1)$.

$\mathcal{R}$ denotes the class of all *recursive* functions of one variable, i.e., total computable functions with arguments and values from N. f, g, and h, with or without decorations, range over $\mathcal{R}$. $\mathcal{C}$ and $\mathcal{S}$, with or without decorations, range over subsets of $\mathcal{R}$.

We fix φ to be an *acceptable programming system* [21, 22, 16] for the partial recursive functions: $N \rightarrow N$. φ_i denotes the partial recursive function computed by the φ-program with index i. W_i denotes $\text{domain}(\varphi_i)$. W_i is then the r.e. set/language ($\subseteq N$) accepted (or, equivalently, generated) by the φ-program i. We let Φ be an arbitrary Blum complexity measure [4] associated with an acceptable programming system φ; such measures exist for any acceptable programming system [4]. $W_{i,s}$ denotes the set $\{x \mid x < s \ \wedge \ \Phi_i(x) \leq s\}$.

In some contexts p and q, with or without decorations, range over programs. In other contexts, p and q range over total recursive functions, with the range of p and q being interpreted as (indexes for) programs. In some contexts P, with or without decorations, ranges over programs. In other contexts, P ranges over sets of programs.

For any predicate Q, $\mu n.Q(n)$ denotes the minimum integer n such that $Q(n)$ is true if such an n exists; it is undefined otherwise. For any set A, 2^A denotes the power set of A. A^k denotes the Cartesian product of A with itself k times. The quantifiers "$\overset{\infty}{\forall}$," "$\exists!$," and "$\overset{\infty}{\exists}$" mean "for all but finitely many," "there exists a unique," and "there exist infinitely many," respectively.

3. Preliminaries. The kind of data a scientist handles in the investigation of a phenomenon F is an ordered pair $(x, f(x))$, where f is the function associated with F and $f(x)$ is the result of experiment x on F. At any given time, a scientist conjectures a hypothesis after seeing a finite sequence of such ordered pairs. We let SEQ denote the set of all finite sequences of ordered pairs. Finite sequences are also referred to as initial segments. As already mentioned, a hypothesis is simply a computer program identified by its index in a given fixed acceptable programming system. Based on these observations, we describe a learning machine in Definition 3.1 below. We let σ and τ, with or without decorations, range over SEQ. $\text{content}(\sigma)$ denotes the set of pairs appearing in σ. The length of σ, denoted by $|\sigma|$, is the number of elements in σ. $\sigma \diamond (x, y)$ denotes the *concatenation* of (x, y) at the end of sequence σ.

DEFINITION 3.1. *A* learning machine *is an algorithmic device that computes a mapping from* SEQ *into* N. *We let* $\mathbf{M}$, *with or without decorations, range over learning machines.*

Scientific inquiry is a limiting process. There is no fixed order in which experiments may be performed, and a scientist is never sure if any new evidence would cause a revision of the currently held hypothesis. The notion of a *text* is described in Definition 3.2 to model the infinite sequence of experimental data that a scientist may encounter in the course of investigating a phenomenon.

DEFINITION 3.2.

1. *A* text *is any infinite sequence of ordered pairs. We let T, with or without decorations, range over texts.*
2. *The set of pairs appearing in a text T is denoted by* content(T).
3. *Let a total function $f : N \to N$ and a text T be given. T is for f just in case* content$(T) = \{(x, y) \mid f(x) = y\}$.
4. *The initial finite sequence of T of length n is denoted by $T[n]$.*

Definition 3.3 below describes what it means for a learning machine to converge on a text.

DEFINITION 3.3. *Suppose* $\mathbf{M}$ *is a learning machine and T is a text.* $\mathbf{M}(T)\downarrow$ (*read:* $\mathbf{M}(T)$ converges) $\iff$ $(\exists p)(\overset{\infty}{\forall} n)[\mathbf{M}(T[n]) = p]$. *If* $\mathbf{M}(T)\downarrow$, *then* $\mathbf{M}(T)$ *is defined = the unique p such that* $(\overset{\infty}{\forall} n)[\mathbf{M}(T[n]) = p]$; *otherwise,* $\mathbf{M}(T)$ *is said to diverge* (*written:* $\mathbf{M}(T)\uparrow$).

Definition 3.4 below describes what it means for a learning machine to successfully learn a function.

DEFINITION 3.4. (See [12, 3, 8].)

1. $\mathbf{M}$ $\mathbf{Ex}^a$-identifies *a total function f* (*written:* $f \in \mathbf{Ex}^a(\mathbf{M})$) $\iff$ ($\forall$ *texts T for f*)$(\exists p \mid \varphi_p =^a f)[\mathbf{M}(T)\downarrow = p]$.
2. $\mathbf{Ex}^a = \{\mathcal{S} \mid (\exists \mathbf{M})[\mathcal{S} \subseteq \mathbf{Ex}^a(\mathbf{M})]\}$.

Definition 3.4 above models the situation in which a scientist has access to an *accurate* source of data. Since totally accurate experimental data is seldom available, models of scientific inquiry should accommodate inaccuracies. In the paradigm under discussion, inaccurate data is modeled by inaccurate texts. First, we briefly consider the kinds of inaccuracies that may arise in experimentation. The subject of inaccuracies in the data available to a learning machine has been previously studied by Schäfer-Richter [23], Fulk and Jain [11], Osherson, Stob, and Weinstein [17], and Jain [14, 15].

- *Noisy data*: Experimental error, usually caused by faulty equipment, may result in spurious data that is not representative of the phenomenon under investigation.
- *Incomplete data*: It may not be feasible to perform certain experiments either due to technological limitations or due to ethical considerations. Such situations result in incomplete data.
- *Imperfect data*: In most experimental investigations, the inaccuracies are a mixture of both noisy and incomplete data. Such situations are said to yield imperfect data.

The three kinds of inaccuracies discussed above suggest three natural extensions to the notion of texts defined below.

DEFINITION 3.5. (See [11, 17, 14]; also see [23].) *Let a text T and a function $f \in \mathcal{R}$ be given. Let $a \in N \cup \{*\}$. Then we define the following:*

1. *T is said to be* a-noisy *for f just in case* $\{(x, y) \mid f(x) = y\} \subseteq$ content(T) *and* card(content$(T) - \{(x, y) \mid f(x) = y\}) \leq a$.
2. *T is said to be* a-incomplete *for f just in case* content$(T) \subseteq \{(x, y) \mid f(x) = y\}$ *and* card$(\{(x, y) \mid f(x) = y\} -$ content$(T)) \leq a$.

3. *T is said to be* a-imperfect *for f just in case* $\mathrm{card}(\{(x,y) \mid f(x) = y\} \mathbf{\Delta} \mathrm{content}(T)) \leq a$.

Definition 3.6 below describes what it means for a learning machine to learn a function from inaccurate texts. We give the definition for noisy texts; the corresponding notions for incomplete and imperfect texts (respectively called $\mathbf{In}^a\mathbf{Ex}^b$ and $\mathbf{Im}^a\mathbf{Ex}^b$) can be defined similarly.

DEFINITION 3.6. (See [11, 17, 14]; also see [23].) *Let $a, b \in N \cup \{*\}$.*

1. $\mathbf{M}$ $\mathbf{N}^a\mathbf{Ex}^b$-identifies *a total function f (written: $f \in \mathbf{N}^a\mathbf{Ex}^b(\mathbf{M})$)* $\iff$ $(\forall$ *a-noisy texts T for f*$)(\exists p \mid \varphi_p =^b f)[\mathbf{M}(T)\downarrow = p]$.
2. $\mathbf{N}^a\mathbf{Ex}^b = \{\mathcal{S} \mid (\exists \mathbf{M})[\mathcal{S} \subseteq \mathbf{N}^a\mathbf{Ex}^b(\mathbf{M})]\}$.

4. Multiple inaccurate texts. The previous section described a paradigm that models a scientist receiving data from a *single*, albeit possibly inaccurate, source. In actual scientific practice, a phenomenon is investigated by a number of different scientists, each performing their own experiments. In due course of time, the data of one scientist becomes available to another through scientific journals, word of mouth, personal communications, etc. Thus a scientist conjectures hypotheses based on data coming from a number of different sources. This situation could be modeled in the present paradigm as a machine receiving multiple texts, some or all of which are inaccurate. It should be noted that the presence of inaccuracies in at least some of the texts is essential for this model to be different because learning from multiple texts, each of which are accurate, is equivalent to learning from a single accurate text.

Before we define learning from multiple inaccurate texts, we need to tinker with the definition of a learning machine to account for data coming from more than one source. Definition 4.1 below describes learning machines that receive multiple streams of data.

DEFINITION 4.1. *Let $k \in N^+$. A* learning machine with k streams *is an algorithmic device that computes a mapping from* SEQ^k *into N.*

Again, we let $\mathbf{M}$, with or without decorations, range over learning machines with multiple streams; it will be clear from the context if we mean a learning machine with a single stream. Definition 4.2 below describes what it means for a learning machine to converge on multiple texts.

DEFINITION 4.2. *Let $k \in N^+$. Let $\mathbf{M}$ be a learning machine and $T_1, T_2, \ldots, T_k$ are k texts.* $\mathbf{M}(T_1, T_2, \ldots, T_k)\downarrow$ *(read:* $\mathbf{M}(T_1, T_2, \ldots, T_k)$ *converges)* $\iff$ $(\exists p)(\exists n)(\forall n_1, n_2, \ldots, n_k \mid n_1 \geq n, n_2 \geq n, \ldots, n_k \geq n)[\mathbf{M}(T_1[n_1], T_2[n_2], \ldots, T_k[n_k]) = p]$. *If* $\mathbf{M}(T_1, T_2, \ldots, T_k)\downarrow$, *then* $\mathbf{M}(T_1, T_2, \ldots, T_k)$ *is defined = the unique p such that* $(\exists n)(\forall n_1, n_2, \ldots, n_k \mid n_1 \geq n, n_2 \geq n, \ldots, n_k \geq n)[\mathbf{M}(T_1[n_1], T_2[n_2], \ldots, T_k[n_k]) = p]$; *otherwise,* $\mathbf{M}(T_1, T_2, \ldots, T_k)$ *is said to diverge (written:* $\mathbf{M}(T_1, T_2, \ldots, T_k)\uparrow$*).*

Definition 4.3 below describes what it means for a learning machine to learn a function from multiple inaccurate texts. We give the definition for noisy texts; the corresponding notions for incomplete and imperfect texts may be defined similarly.

DEFINITION 4.3. *Let $j, k \in N^+$. Let $a, b \in N \cup \{*\}$.*

(a) *A learning machine* $\mathbf{M}$ $\mathbf{Mul}^j_k\mathbf{N}^a\mathbf{Ex}^b$-identifies *a total function f (written: $f \in \mathbf{Mul}^j_k\mathbf{N}^a\mathbf{Ex}^b(\mathbf{M})$)* $\iff$ $(\forall$ *k texts $T_1, T_2, \ldots, T_k$ such that at least j out of these k texts are a-noisy for f*$)(\exists p \mid \varphi_p =^b f)[\mathbf{M}(T_1, T_2, \ldots, T_k)\downarrow = p]$.

(b) $\mathbf{Mul}^j_k\mathbf{N}^a\mathbf{Ex}^b = \{\mathcal{S} \mid (\exists \mathbf{M})[\mathcal{S} \subseteq \mathbf{Mul}^j_k\mathbf{N}^a\mathbf{Ex}^b(\mathbf{M})]\}$.

So a machine $\mathbf{Mul}^j_k\mathbf{N}^a\mathbf{Ex}^b$-($\mathbf{Mul}^j_k\mathbf{In}^a\mathbf{Ex}^b$-, $\mathbf{Mul}^j_k\mathbf{Im}^a\mathbf{Ex}^b$-) identifying a function f, upon being fed k texts at least j of which are a-noisy (a-incomplete, a-imperfect) for f, converges to a program for a b-variant for f. Henceforth, when

discussing $\mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^b$, etc., those texts for which the inaccuracy is within the required bound are referred to as *acceptable* texts. We next note that the only interesting cases are those for which the number of acceptable texts is a majority of the total number of texts.

Consider a collection of recursive functions $\mathcal{C}$. Assume $f_1, f_2 \in \mathcal{C}$ to be such that $f_1 \neq^{2a} f_2$. Then for all k, we have $\mathcal{C} \notin \mathbf{Mul}_k^{\lfloor \frac{k}{2} \rfloor}\mathbf{N}^0\mathbf{Ex}^a$. This is because among k texts there may be $\lfloor \frac{k}{2} \rfloor$ accurate texts for both f_1 and f_2. To avoid such problems, we consider only those cases in which a strict majority of the texts are acceptable.[2]

We now discuss our results before presenting them with proofs in the next section.

One of our central aims is to investigate the relationship between identification from a single inaccurate text and identification from multiple inaccurate texts, where the inaccuracy is of the same kind. For noisy texts, for instance, this question can be phrased as follows: How does $\mathbf{N}^a\mathbf{Ex}^b$ relate to $\mathbf{Mul}_k^j\mathbf{N}^c\mathbf{Ex}^d$ for various values of a, b, c, d and j, k? Similar questions can be posed for incomplete and imperfect texts. This general question turns out to be combinatorially very difficult, and only partial results are presented in this paper.

First, the following immediate proposition states that for the same bound on the number of inaccuracies and the same kind of inaccuracy, identification from multiple inaccurate texts cannot be better than identification from a single inaccurate text, that is, collections of functions that can be identified from multiple inaccurate texts can also identified from a single inaccurate text.

PROPOSITION 4.4. *Let $j, k \in N$ such that $k \geq j$. Let $a, b \in N \cup \{*\}$.*

1. $\mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^b \subseteq \mathbf{N}^a\mathbf{Ex}^b$.
2. $\mathbf{Mul}_k^j\mathbf{In}^a\mathbf{Ex}^b \subseteq \mathbf{In}^a\mathbf{Ex}^b$.
3. $\mathbf{Mul}_k^j\mathbf{Im}^a\mathbf{Ex}^b \subseteq \mathbf{Im}^a\mathbf{Ex}^b$.

The natural question is "When does the above proposition yield an equality?" That is, we would like to have sufficient conditions for when identification from multiple texts is no worse than identification from a single inaccurate text. The results in section 5.1 give some conditions such that—for each kind of inaccuracy (noise, incompleteness, and imperfection)—if a collection of functions can be learned from a single inaccurate text, then it can also be learned (with the same bound on the number of errors in the final program) from multiple inaccurate texts, provided the bound on the number of inaccuracies in at least a *majority* of the acceptable texts is the same as the bound on the number of inaccuracies in the single inaccurate text.

The next natural question is under what conditions identification from multiple texts, some of which may be inaccurate, constitutes a restriction on identification from a single inaccurate text. More precisely, we would like to answer the following question: For what relationship between a, b, c, d and j, k is $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_k^j\mathbf{N}^c\mathbf{Ex}^d \neq \emptyset$? Unfortunately, this question turns out to be very difficult, and we are only able to provide partial results for identification from three texts, at least two of which are acceptable (that is, for the case where $j = 2$ and $k = 3$). Theorem 5.4 in section 5.2 is a very general result that gives sufficient conditions for a, b, c, and d such that $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d \neq \emptyset$. Unfortunately, these conditions turn out to be somewhat intricate, but the reader can get a feel for the special cases of these conditions presented

[2] It should be noted that if for all $f_1, f_2 \in \mathcal{C}$, $f_1 =^{2a} f_2$, then all functions in $\mathcal{C}$ are finite variants of any fixed function in the class. We feel that such cases are not interesting because if at least j out of k input texts are acceptable for a function $f \in \mathcal{C}$, then one can easily find a program p (in the limit) such that at least j out of k input texts are acceptable for φ_p (though this φ_p may not be in $\mathcal{C}$).

in Corollaries 5.5, 5.6, 5.7, and 5.8. These results are presented in section 5.2.

Also in section 5.2, we consider the cases where the restrictive effects of learning from multiple inaccurate texts can be compensated. More precisely, we identify the relationship between a, b, c, and d such that $\mathbf{N}^a\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$. This is achieved by either allowing a greater upper bound on the number of errors tolerated in the final program (that is, by making d greater than b) or making the quality of acceptable texts better than the quality of single inaccurate text (that is, by making c smaller than a). An interesting result along these lines is that for each type of inaccuracy, the collections of functions identified from a single inaccurate text can also be identified from three texts, at least two of which are acceptable with the same bound on the number of inaccuracies as the single text, provided we are prepared to tolerate twice the number of errors in the final program. This follows from Theorems 5.9, 5.19, and 5.22.

In section 5.3, we consider the effects of increasing the bound on the number of errors tolerated in the final program-keeping other parameters the same, and the effects of increasing the bound on the number of inaccuracies in the acceptable texts, keeping other parameters the same. We are able to show the expected result that for each inaccuracy type, larger collections of functions become identifiable if we

- increase the bound on the number of errors allowed in the final program (keeping other parameters the same) or
- decrease the bound on the number of inaccuracies allowed in the input texts (keeping other parameters the same).

The results discussed so far have been about the same kind of inaccuracies. Finally, in Section 5.4, we present results that compare one kind of inaccuracy with another in the context of identification from multiple texts. Our findings show that noisy texts are better for learnability than incomplete texts. This observation may be interpreted as saying that spurious data is preferable to missing data. However, we also show that imperfect texts are worse for learnability than incomplete texts, which can be seen as saying that a mixture of spurious and missing data is less desirable than only missing data.

5. Results. We now present results relating various criteria of inference introduced in section 4. Most results in this paper are about the case in which a learning machine is receiving data from three sources, at least two of which are acceptable. Section 5.1 presents results about those cases in which learning from multiple texts is not a restriction on learning power. Section 5.2 presents results in which learning from multiple texts is a restriction on learning power and also results from cases in which the deleterious effects of learning from multiple inaccurate texts can be compensated by either allowing extra errors in the final program or improving the quality of the acceptable texts. Section 5.3 presents the hierarchy results and section 5.4 presents some results about the interaction between different kinds of inaccuracies.

From a purely technical point of view, the results presented in section 5.2 are the most difficult results of this paper. Also, we are able to establish more results for noisy texts than for incomplete and imperfect texts.

5.1. When identification from multiple texts is not restrictive. We consider cases when learning from multiple inaccurate texts is equivalent to learning from a single inaccurate text. Parts 1, 2, and 3 of the next result show that for each kind of inaccuracy, the collections of functions that can be identified from a single text with a finite number of inaccuracies is exactly the same as the collections of functions that can be identified (with the same bound on the number of errors allowed in the final

program) from multiple texts, at least a majority of which have only a finite number of inaccuracies.

THEOREM 5.1. *Let $j, k \in N$, $k \geq j > \lfloor \frac{k}{2} \rfloor$. Let $a \in N \cup \{*\}$.*

1. $\mathbf{Mul}_k^j\mathbf{N}^*\mathbf{Ex}^a = \mathbf{N}^*\mathbf{Ex}^a$.
2. $\mathbf{Mul}_k^j\mathbf{In}^*\mathbf{Ex}^a = \mathbf{In}^*\mathbf{Ex}^a$.
3. $\mathbf{Mul}_k^j\mathbf{Im}^*\mathbf{Ex}^a = \mathbf{Im}^*\mathbf{Ex}^a$.
4. $\mathbf{Mul}_k^j\mathbf{N}^0\mathbf{Ex}^a = \mathbf{Mul}_k^j\mathbf{In}^0\mathbf{Ex}^a = \mathbf{Mul}_k^j\mathbf{Im}^0\mathbf{Ex}^a = \mathbf{Ex}^a$.

Proof. Given k texts $T_1, T_2, \ldots, T_k$, construct a text T such that $(x, y) \in \text{content}(T) \iff \text{card}(\{i \mid (x, y) \in \text{content}(T_i)\}) \geq \lfloor \frac{k}{2} \rfloor + 1$. It is easy to see that if at least $\lfloor \frac{k}{2} \rfloor + 1$ of the k texts are $*$-noisy ($*$-incomplete, $*$-imperfect, accurate) for f, then so is T. The theorem follows. □

We now introduce a technical notion that is used in later proofs. Let P be a finite set of programs. Define the program $\mathbf{Unify}(P)$ as follows:

begin $\varphi_{\mathbf{Unify}(P)}(x)$
  Search for $i \in P$ such that $\varphi_i(x)\downarrow$.
  If and when such an i is found, output $\varphi_i(x)$ (for the first such i found).
end $\varphi_{\mathbf{Unify}(P)}(x)$

The next result shows that for $a \in N \cup \{*\}$, the collections of functions for which an exact program can be identified from a single text with the number of inaccuracies bounded by a is exactly the same as the collections of functions for which an exact program can be identified from multiple texts, at least a majority of which have at most a inaccuracies.

THEOREM 5.2. *Let $j, k \in N$, $k \geq j > \lfloor \frac{k}{2} \rfloor$. Let $a \in N \cup \{*\}$.*

1. $\mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex} = \mathbf{N}^a\mathbf{Ex}$.
2. $\mathbf{Mul}_k^j\mathbf{In}^a\mathbf{Ex} = \mathbf{In}^a\mathbf{Ex}$.
3. $\mathbf{Mul}_k^j\mathbf{Im}^a\mathbf{Ex} = \mathbf{Im}^a\mathbf{Ex}$.

Proof. 1. Proposition 4.4 yields $\mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex} \subseteq \mathbf{N}^a\mathbf{Ex}$. We now show that $\mathbf{N}^a\mathbf{Ex} \subseteq \mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}$.

Let $\mathbf{M}$ $\mathbf{N}^a\mathbf{Ex}$-identify $\mathcal{C}$. We show how to $\mathbf{Ex}$-identify $f \in \mathcal{C}$ from k texts, at least j of which are a-noisy for f. Let n, $S = \{i_1, i_2, \ldots, i_j\} \subseteq \{1, 2, \ldots, k\}$, and $P = \{p_{i_1}, p_{i_2}, \ldots, p_{i_j}\}$ be such that

(i) for all $l \in S$ and $m \geq n$, $\mathbf{M}(T_l[m]) = p_l$,
(ii) if $p, q \in P$, then $\text{card}(\{x \mid \varphi_p(x)\downarrow \neq \varphi_q(x)\downarrow\}) = 0$, and
(iii) $\text{card}(S) = j$.

Clearly, such an n, S, and P exist. (Let $S \subseteq \{i \mid T_i \text{ is } a\text{-noisy for } f\}$ be a set of cardinality j; let n be such that for all $l \in S$, $\mathbf{M}$ converges on T_l after seeing at most n inputs; let $P = \{\mathbf{M}(T_l) \mid l \in S\}$. Then n, S, P satisfy (i), (ii), and (iii).) Clearly, a machine $\mathbf{M}'$, given texts $T_1, \ldots, T_k$, can find such an n, S, and P in the limit. Now since $\mathbf{M}$ on $T_l, l \in S$, converges to p_l and $\text{card}(S) > k - j$, there exists an $l \in S$ such that $\varphi_{p_l} = f$. This along with (ii) above implies that $\mathbf{Unify}(P)$ is a program for f. This proves part 1.

Parts 2 and 3 can be proved similarly. □

The next result shows that for $a \in N \cup \{*\}$, the collections of functions for which a $*$-error program can be identified from a single text with the number of inaccuracies bounded by a is exactly the same as the collections of functions for which a $*$-error program can be identified from multiple texts, at least a majority of which have at most a inaccuracies.

THEOREM 5.3. *Let $j, k \in N$, $k \geq j > \lfloor \frac{k}{2} \rfloor$. Let $a \in N \cup \{*\}$.*

1. $\mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^* = \mathbf{N}^a\mathbf{Ex}^*$.
2. $\mathbf{Mul}_k^j\mathbf{In}^a\mathbf{Ex}^* = \mathbf{In}^a\mathbf{Ex}^*$.
3. $\mathbf{Mul}_k^j\mathbf{Im}^a\mathbf{Ex}^* = \mathbf{Im}^a\mathbf{Ex}^*$.

Proof. 1. Proposition 4.4 yields $\mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^* \subseteq \mathbf{N}^a\mathbf{Ex}^*$. We now show that $\mathbf{N}^a\mathbf{Ex}^* \subseteq \mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^*$.

Let $\mathbf{M}$ $\mathbf{N}^a\mathbf{Ex}^*$-identify $\mathcal{C}$. We show how to $\mathbf{Ex}^*$-identify $f \in \mathcal{C}$ from k texts, at least j of which are a-noisy for f. Let n, $S = \{i_1, i_2, \ldots, i_j\} \subseteq \{1, 2, \ldots, k\}$, and $P = \{p_{i_1}, p_{i_2}, \ldots, p_{i_j}\}$ be such that

(i) for all $l \in S$ and $m \geq n$, $\mathbf{M}(T_l[m]) = p_l$,

(ii) if $p, q \in P$, then $\mathrm{card}(\{x \mid \varphi_p(x)\downarrow \neq \varphi_q(x)\downarrow\}) \leq n$, and

(iii) $\mathrm{card}(S) = j$.

Clearly, such an n, S, and P exist. (Let $S \subseteq \{i \mid T_i \text{ is } a\text{-noisy for } f\}$ be a set of cardinality j; let $P = \{\mathbf{M}(T_l) \mid l \in S\}$; let n be so large that for all $l \in S$, $\mathbf{M}$ converges on T_l after seeing at most n inputs and $f =^{n/2} \varphi_p$ for all $p \in P$. Then n, S, and P satisfy (i), (ii), and (iii).) Clearly, a machine $\mathbf{M}'$, given texts $T_1, \ldots, T_k$, can find such an n, S, P in the limit. Now since $\mathbf{M}$ on $T_l, l \in S$, converges to p_l and $\mathrm{card}(S) > k - j$, there exists an $l \in S$ such that $\varphi_{p_l} =^* f$. This along with (ii) above implies that $\mathbf{Unify}(P)$ is a program for a finite variant of f. This proves part 1.

Parts 2 and 3 can be proved similarly. □

5.2. When identification from multiple texts is restrictive. In this section we initially tackle the question of when, for the same kind of inaccuracy, learning from multiple texts, some of which may be inaccurate, is a restriction over learning from a single inaccurate text. That is, we consider the question: For which values of a, b, c, d and j, k is $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_k^j\mathbf{N}^c\mathbf{Ex}^d \neq \emptyset$? As already noted, this turns out to be a very difficult question. For the case where $j = 2$ and $k = 3$, we are able to provide significant partial answers. These results follow from a very complex diagonalization argument.

In this section, we then consider the related question of when the restriction of identification from multiple texts can be compensated. That is, for what values of a, b, c, d is $\mathbf{N}^a\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$. These results use simulation arguments.

We first present results for noisy texts (section 5.2.1), followed by results for incomplete texts (section 5.2.2). Only a few results are presented for imperfect texts.

5.2.1. Noisy texts. We present a general theorem (Theorem 5.4) that covers numerous cases in which identification from multiple noisy texts is a restriction over identification from a single noisy text. Unfortunately, the conditions in the theorem turn out to be very complex, and the reader is advised to see the four corollaries presented immediately after the statement of the theorem for cases in which multiple texts pose a restriction. However, we first try to present the intuitive motivation of the theorem. Consider the question of trying to compare identification from a single noisy text and identification from three texts, at least two of which are noisy. Furthermore, let

- a be the bound on the noise level of the single text,
- b be the bound on the number of errors allowed in programs inferred from the single text,
- c be the bound on the noise level of the acceptable texts in the case of identification from multiple texts, and
- d be the bound on the number of errors allowed in programs inferred from multiple texts.

We would like to know when

$$\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d \neq \emptyset.$$

Now let us see the effects of altering each of the parameters. Increasing a or decreasing b makes the class $\mathbf{N}^a\mathbf{Ex}^b$ smaller. Similarly, decreasing c and increasing d makes the class $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$ larger. The following theorem tells us how much we can "stretch" the parameters a, b, c, and d such that there are still collections of functions that can be identified from single noisy text but not from three texts, at least two of which are acceptable.

THEOREM 5.4. *Let $a, b, c, d \in N$ be given. If there exist $r, \alpha \in N$ such that*

- $r - b \leq \alpha \leq \min(\{b, r\})$,
- $r \leq c$,
- $2r > a$, *and*
- $d < \max(\{b + \alpha - \frac{r}{2}, b + \frac{\alpha}{3}\})$,

then $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d \neq \emptyset$.

Before giving a proof of the above result, we present corollaries. The first corollary says that for given $a, b, c \in N$ such that a is greater, but not too much greater, than c ($c \leq a \leq 2c - 1$) and c is no greater than b, there are collections of functions for which a b-error program can be identified from an a-error text but for which even a $(b + \lceil \frac{c}{2} \rceil - 1)$-error program cannot be identified from three texts, at least two of which are c-noisy for the function being learned.

COROLLARY 5.5. *Let $a, b, c \in N$ be such that $c \leq a \leq 2c - 1$, $c \leq b$. Then* $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil - 1} \neq \emptyset$.

Proof. Take $r = \alpha = c$ in Theorem 5.4. □

The next three corollaries are more variations on this theme and give insight into how much the parameters a, b, c, and d can be stretched.

COROLLARY 5.6. *Let $a, b, c \in N$ be such that $c \leq a \leq 2c - 1$, $\frac{c}{2} \leq b \leq \frac{3}{4}\lceil \frac{a+1}{2} \rceil$. Then* $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{\lceil \frac{4b}{3} \rceil - 1} \neq \emptyset$.

Proof. Take $r = c$ and $\alpha = b$ in Theorem 5.4. □

COROLLARY 5.7. *Let $a, b, c \in N$ be such that $c \leq a \leq 2c - 1$, $\frac{a}{2} < b \leq c$. Then* $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{\lceil \frac{3b}{2} \rceil - 1} \neq \emptyset$.

Proof. Take $r = \alpha = b$ in Theorem 5.4. □

COROLLARY 5.8. *Let $a, b, c \in N$ be such that $c \leq a \leq 2c-1$, $\max(\{\frac{c}{2}, \frac{3}{4}\lceil \frac{a+1}{2} \rceil\}) \leq b \leq \frac{a}{2}$. Then* $\mathbf{N}^a\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{\lceil 2b - \frac{a+2}{4} \rceil - 1} \neq \emptyset$.

Proof. Take $r = \lceil \frac{a+1}{2} \rceil$ and $\alpha = b$ in Theorem 5.4. □

We now give a proof of Theorem 5.4. The proof turns out to be somewhat complex because we have to come up with a collection of functions that can be identified from a single noisy text but which cannot be identified from three texts, at least two of which are noisy. To design this collection of functions, we employ a technique from the study of identification from single inaccurate texts [11]. We then employ the operator recursion theorem [5] to construct a diagonalization argument to show that this class cannot be identified from three texts, at least two of which are noisy. The diagonalization technique is an adaptation of the techniques used elsewhere in the study of inductive inference; we direct the reader to a survey of such techniques [6].

Proof of Theorem 5.4. Assume the hypothesis of the theorem and fix r and α satisfying the hypothesis.

Let $A = \{\langle 0, x \rangle \mid 0 \leq x < r\}$.

Consider the following class of functions.

$\mathcal{C} = \{f \in \mathcal{R} \mid$ the following conditions hold:
(1) $f(A) = f(\langle 0,0\rangle) \in \{1,2,3\}$;
(2) $(\forall i \in \{1,2,3\} - \{f(\langle 0,0\rangle)\})[(\max(\{f(\langle i,\langle x,0\rangle\rangle) \mid x \in N\})$ exists) $\wedge$
$(\varphi_{\max(\{f(\langle i,\langle x,0\rangle\rangle)\mid x\in N\})} =^b f)]$;
(3) $(\forall i \in \{1,2,3\})(\forall x,y,z)[f(\langle i,\langle x,y\rangle\rangle) = f(\langle i,\langle x,z\rangle\rangle)]\}$.

Intuitively, A is the set of points where some coding is done. (A is the only place where inaccuracies will matter.) a-noise is not enough to spoil the coding in A for $\mathbf{N}^a\mathbf{Ex}^b$-identification. However, r-noise is enough to spoil the coding for $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$-identification.

It is easy to show that $\mathcal{C} \in \mathbf{N}^a\mathbf{Ex}^b$. To see this, suppose that T is an a-noisy text for $f \in \mathcal{C}$. Since $2r > a$, there exist an $i \in \{1,2,3\}$ and $x \in A$ such that $(x,i) \notin \text{content}(T)$. This along with clause (2) in the definition of $\mathcal{C}$ implies that $\varphi_{\max(\{f(\langle i,\langle x,0\rangle\rangle)\mid x\in N\})} =^b f$. Now $\max(\{f(\langle i,\langle x,0\rangle\rangle) \mid x \in N\})$ can be determined in the limit from T due to cylindrification in clause (3). It follows that $\mathcal{C} \in \mathbf{N}^a\mathbf{Ex}^b$.

We now show that $\mathcal{C} \notin \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$.

The essential idea for diagonalization is that a machine cannot $\mathbf{Ex}^d$-identify $\mathcal{C}$ if the coding in A (due to clause (1) in definition of $\mathcal{C}$) is spoiled. So suppose by way of contradiction that machine $\mathbf{M}$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$-identifies $\mathcal{C}$.

Then by the operator recursion theorem [5], there exists a recursive, 1–1, and increasing p, with $p(0) > 1$, defined as follows in stages.

We define $\varphi_{p(i)}$ in stages $s \geq 3$. Let x_3^s denote the least x such that for all y, $\varphi_{p(1)}(\langle 3,\langle x,y\rangle\rangle)$ is not defined before stage s.

For $x < \alpha$, let $\varphi_{p(1)}(\langle 0,x\rangle) = 2$. For $\alpha \leq x < r$, let $\varphi_{p(1)}(\langle 0,x\rangle) = 3$. Let $\varphi_{p(1)}(\langle 1,\langle 0,0\rangle\rangle) = p(1)$ and $\varphi_{p(1)}(\langle 2,\langle 0,0\rangle\rangle) = p(2)$.

For $x < \alpha$, let $\varphi_{p(2)}(\langle 0,x\rangle) = 1$. For $\alpha \leq x < r$, let $\varphi_{p(2)}(\langle 0,x\rangle) = 3$. Let $\varphi_{p(2)}(\langle 1,\langle 0,0\rangle\rangle) = p(1)$ and $\varphi_{p(2)}(\langle 2,\langle 0,0\rangle\rangle) = p(2)$.

Let σ_1^3, σ_2^3 and σ_3^3 be such that

$$\text{content}(\sigma_1^3) = \{(x,2) \mid x \in A\} \cup \{(x,3) \mid x \in A\} \cup \{(\langle 1,\langle 0,0\rangle\rangle, p(1)), (\langle 2,\langle 0,0\rangle\rangle, p(2))\},$$
$$\text{content}(\sigma_2^3) = \{(x,1) \mid x \in A\} \cup \{(x,3) \mid x \in A\} \cup \{(\langle 1,\langle 0,0\rangle\rangle, p(1)), (\langle 2,\langle 0,0\rangle\rangle, p(2))\},$$
$$\text{content}(\sigma_3^3) = \{(x,1) \mid x \in A\} \cup \{(x,2) \mid x \in A\} \cup \{(\langle 1,\langle 0,0\rangle\rangle, p(1)), (\langle 2,\langle 0,0\rangle\rangle, p(2))\}.$$

Intuitively, we will form three (nearly identical) texts T_1, T_2, and T_3 with the initial segments σ_1^3, σ_2^3, and σ_3^3 as defined above. Note that these three texts spoil the coding in A (since all possible values of the coding are present in two of these texts). Now the basic idea in the diagonalization below is to either

(1) force infinitely many mind changes by $\mathbf{M}$ (see step 4 in the construction)—in this case T_1 and T_2 form the c-noisy texts for the diagonalizing function—or

(2) based on what the final program $\mathbf{M}(T_1,T_2,T_3)$ does on the inputs from A (see step 2 in the construction), do an appropriate diagonalization of the same form as done in $\mathbf{Ex}$-hierarchy theorem in [8] (see steps 5a, 5c, and 6a in the construction). The diagonalization is done so as to maximize the errors of the final program under the constraint of keeping the diagonalizing function within the class (see the comments at the end of step 2 in the construction). Steps 5b and 6b in the construction are needed since one cannot effectively determine whether program $\mathbf{M}(T_1,T_2,T_3)$ halts on inputs from the set A (we can do it only in the limit).

Let $\varphi_{p(i)}^s$ denote the part of $\varphi_{p(i)}$ defined before stage s. Go to stage 3.

Begin Stage s

1. Let $q = \mathbf{M}(\sigma_1^s, \sigma_2^s, \sigma_3^s)$.

2. Let $D_1 = \{x \mid x \in A \ \wedge \ \Phi_q(x) \leq s \ \wedge \ \varphi_q(x) = 1\}$.
Let $D_2 = \{x \mid x \in A \ \wedge \ \Phi_q(x) \leq s \ \wedge \ \varphi_q(x) = 2\}$.
Let $D_3 = \{x \mid x \in A \ \wedge \ \Phi_q(x) \leq s \ \wedge \ \varphi_q(x) \in N - \{1, 2\}\}$.
Let $err_{1,2} = r - \text{card}(D_3) + b - \alpha$.
Let $err_{2,3} = r - \text{card}(D_1) + b - r + \alpha$.
Let $err_{1,3} = r - \text{card}(D_2) + b - r + \alpha$.
If $err_{1,2} \leq \max(\{err_{1,3}, err_{2,3}\})$, then let $extra\_err = b - r + \alpha$; otherwise, let $extra\_err = b - \alpha$. If $err_{1,3} > err_{2,3}$, then let $l = 1$ and $l' = 2$; otherwise, let $l = 2$ and $l' = 1$.
Let $X^s \subset \{\langle 4, x\rangle \mid x \geq s\}$ denote the (lexicographically least) set of $extra\_err$ elements such that for all $x \in X^s$, $\varphi_{p(1)}(x)$ is not defined before stage s.
(*Note.* Let $T_i = \bigcup_s \sigma_i^s$. D_1 (D_2, D_3) denote the points in A for which φ_q seems to be correct if T_2 and T_3 (T_1 and T_3, T_1 and T_2) are the acceptable texts for the function being learned and $err_{i,j}$ denotes the error that we can force if we chose a function such that T_i and T_j are correct texts for the function. Informally, an attempt is made at every stage to try and make those two texts correct so as to maximize the number of errors made by $\mathbf{M}$'s current output program.)

3. For $x \in A$, let $\varphi_{p(s)}(x) = l'$. Let $\varphi_{p(s)}(\langle 3, \langle x_3^s, 0\rangle\rangle) = \varphi_{p(1)}(\langle 3, \langle x_3^s, 0\rangle\rangle) = \varphi_{p(2)}(\langle 3, \langle x_3^s, 0\rangle\rangle) = p(s)$. For $x \in \text{domain}(\varphi_{p(1)}^s) - A$, let $\varphi_{p(s)}(x) = \varphi_{p(1)}(x)$.

4. For $i \in \{1, 2, 3\}$, let σ_i' be an extension of σ_i^s such that $\text{content}(\sigma_i') = \text{content}(\sigma_i^s) \cup \{(\langle 3, \langle x_3^s, 0\rangle\rangle, p(s))\}$.

5. Dovetail steps 5a, 5b, and 5c until (if ever) one of steps 5a or 5b succeeds and at least one iteration of the loop at step 5c is completed. If step 5a succeeds (before step 5b succeeds (if ever)), then complete the then current iteration of the repeat loop in step 5c and go to step 6a. If step 5b succeeds (before step 5a succeeds (if ever)), then complete the then current iteration of the repeat loop in step 5c and go to step 6b.

5a. Search for $\sigma_i'' \supseteq \sigma_i'$, $i \in \{1, 2, 3\}$, such that $\mathbf{M}(\sigma_1^s, \sigma_2^s, \sigma_3^s) \neq \mathbf{M}(\sigma_1'', \sigma_2'', \sigma_3'')$ and for all $j, j' \in \{1, 2, 3\}$, for all $x, y, z \in N$, the following four conditions hold:

- $\text{content}(\sigma_j'') - \text{content}(\sigma_j') = \text{content}(\sigma_{j'}'') - \text{content}(\sigma_{j'}')$.
- for $x \notin A$, if $(x, y), (x, z) \in \text{content}\ (\sigma_j'')$ then $y = z$.
- If $(\langle j, \langle x, y\rangle\rangle, v) \in \text{content}(\sigma_1'')$ and $(\langle j, \langle x, z\rangle\rangle, w) \in \text{content}(\sigma_1'')$, then $v = w$.
- If $(x, y) \in (\text{content}(\sigma_1'') - \text{content}(\sigma_1'))$, then $[[y = 0 \ \wedge \ x \notin A]$ OR $x \in X^s$ OR [for some $i \in \{1, 2, 3\}$, $u, v \in N$: $x = \langle i, \langle u, v\rangle\rangle$ and $\varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle)$ has been defined until now and $\varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle) = y]]$.

(*Remark.* This substep looks for one set of suitable extensions of the currently defined initial segments of the three texts that makes $\mathbf{M}$ change its mind.)

5b. Search for $x \in A - (D_1 \cup D_2 \cup D_3)$ such that $\varphi_q(x)\downarrow$.

5c. Let $Addednew = \emptyset$.
(*Note.* We use *Addednew* in order to identify the set of elements for which we extend $\varphi_{p(1)}$ in step 5c. *Addednew* will be used (in steps 6a and 6b), provided at least one of steps 5a or 5b succeeds).
Repeat
Let x be the least point not in X^s such that $\varphi_{p(1)}(x)$ is not defined until now. If for some $i \in \{1,2,3\}$, x is of the form $\langle i, \langle u, v\rangle\rangle$ and $\varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle)$ is defined until now, then let $\varphi_{p(1)}(x) = \varphi_{p(2)}(x) = \varphi_{p(s)}(x) = \varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle)$; otherwise, let $\varphi_{p(1)}(x) = \varphi_{p(2)}(x) = \varphi_{p(s)}(x) = 0$.
Let $Addednew = Addednew \cup \{x\}$.
Forever

6a. For all x and y such that $(x,y) \in \text{content}(\sigma_1'') - \text{content}(\sigma_1^s)$, let $\varphi_{p(1)}(x) = \varphi_{p(2)}(x) = y$. Let σ_i^{s+1} be an extension of σ_i'' such that $\text{content}(\sigma_i^{s+1}) - \text{content}(\sigma_i'') = \{(x, \varphi_{p(1)}(x)) \mid x \in Addednew\}$. Go to stage $s+1$.

6b. Let σ_i^{s+1} be an extension of σ_i' such that $\text{content}(\sigma_i^{s+1}) - \text{content}(\sigma_i') = \{(x, \varphi_{p(1)}(x)) \mid x \in Addednew\}$. Go to stage $s+1$.

End stage s
Now consider the following cases.
Case 1. All stages terminate.
Let

$$f(x) = \begin{cases} 3 & \text{if } x \in A; \\ \varphi_{p(1)}(x) & \text{otherwise.} \end{cases}$$

Clearly, $f \in \mathcal{C}$ (since $\alpha \le b$, $\varphi_{p(1)} =^b f$, and $\varphi_{p(2)} =^b f$). Also, $\mathbf{M}$ on f makes infinitely many mind changes since step 5b can succeed at most finitely many times before step 5a succeeds.

Case 2. Stage s (≥ 3) is the least stage which starts but never terminates.

Let q, $err_{1,2}$, $err_{2,3}$, $err_{1,3}$, D_1 , D_2, and D_3 be as defined in steps 1 and 2 in stage s. Now $\max(\{err_{1,2}, err_{2,3}, err_{1,3}\}) \ge \frac{err_{1,2}+err_{2,3}+err_{1,3}}{3} \ge \frac{3b+\alpha}{3}$. Thus $\max(\{err_{1,2}, err_{2,3}, err_{1,3}\}) \ge b + \frac{\alpha}{3}$. Also, since $err_{1,2} \le r + b - \alpha$, we have $\max(\{err_{1,3}, err_{2,3}\}) \ge \frac{2b-r+2\alpha}{2}$. Thus $\max(\{err_{1,2}, err_{2,3}, err_{1,3}\}) \ge \max(\{b + \alpha - \frac{r}{2}, b + \frac{\alpha}{3}\})$.

Case 2a. In step 2 of stage s it was found that $err_{1,2} \le \max(\{err_{1,3}, err_{2,3}\})$.

Let l and l' be as found in step 2. In this case $\text{domain}(\varphi_{p(s)}) = N - X^s$. Let f be any fixed total extension of $\varphi_{p(s)}$ such that for all $x \in X^s$, we have $f(x) \ne \varphi_q(x)$.

Clearly, $f \in \mathcal{C}$ (since $\varphi_{p(l)} =^b f$, $\varphi_{p(s)} =^b f$, and $f(\langle 0,0\rangle) = l'$). Let T_1, T_2, and T_3 be extensions of σ_1', σ_2', and σ_3' (as defined in step 4 in stage s), respectively, such that $\text{content}(T_1) - \text{content}(\sigma_1^s) = \{(x,y) \mid f(x) = y \ \wedge \ (\forall z)[(x,z) \notin \text{content}(\sigma_1^s)]\}$ and for all j, the $[|\sigma_1'| + j]$th element of T_1, the $[|\sigma_2'| + j]$th element of T_2, and the $[|\sigma_3'| + j]$th element of T_3 are all the same. Clearly, $\mathbf{M}(T_1, T_2, T_3) = q$ (otherwise, step 5a would succeed).

Now for all $x \in A - D_{l'}$, $f(x) \ne \varphi_q(x)$ (since that was the way $D_{l'}$ was chosen in step 2 of stage s). Thus $\varphi_q \ne^{\max(\{err_{1,3}, err_{2,3}\})-1} f$.

Case 2b. In step 2 of stage s it was found that $err_{1,2} > \max(\{err_{1,3}, err_{2,3}\})$.

In this case, $\text{domain}(\varphi_{p(1)}) = N - X^s$. Let f be a fixed total function such that the following three conditions hold:

- $f(A) = 3$,
- $f(x) = \varphi_{p(1)}(x)$ for $x \in \text{domain}(\varphi_{p(1)}) - A$, and
- $(\forall x \in X^s)[f(x) \neq \varphi_q(x)]$.

Clearly, $f \in \mathcal{C}$ (since $\varphi_{p(1)} =^b f$, $\varphi_{p(2)} =^b f$, and $f(\langle 0,0\rangle) = 3$). Let T_1, T_2, and T_3 be extensions of σ_1', σ_2', and σ_3' respectively, such that $\text{content}(T_1) - \text{content}(\sigma_1^s) = \{(x,y) \mid f(x) = y \ \wedge \ (\forall z)[(x,z) \notin \text{content}(\sigma_1^s)]\}$ and for all j, the $[|\sigma_1'| + j]$th element of T_1, the $[|\sigma_2'| + j]$th element of T_2, and the $[|\sigma_3'| + j]$th element of T_3 are all the same. Also, it is clear that $\mathbf{M}(T_1, T_2, T_3) = q$ (otherwise, step 5a would succeed). Now for all $x \in A - D_3$, $f(x) \neq \varphi_q(x)$ (since that was the way D_3 was chosen in step 2 of stage s). Thus $\varphi_q \neq^{err_{1,2}-1} f$.

From the above cases, it follows that $\mathcal{C} \notin \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$. □

The results established so far identified cases where multiple noisy texts are a restriction over learning from a single noisy text. The natural question that arises is "What are the cases where the restriction of learning from multiple noisy texts can be compensated in some way." That is, for which values of a, b, c, and d is $\mathbf{N}^a\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$? A quick look at the effects of the various parameters yields at least two different ways in which the restrictive impact of multiple texts can be compensated. These are

- by allowing extra errors in the program inferred from multiple texts (that is, by making d larger than b) and
- by having a reduced bound on the number of noise elements in the multiple texts (that is, by making c smaller than a).

Theorems 5.9 and 5.10 use the first technique; Theorems 5.12, 5.13, and 5.14 use the second technique; and Theorem 5.11 can be seen as using a combination of the two techniques to compensate for the deleterious effects of learning from multiple texts. The technique used in these simulation arguments is to first collect all the input points where there is noise in some text. Then construct a program which judiciously chooses the output on these inputs based on the input texts and outputs of the converging programs.

Theorem 5.9 below shows that the collections of functions for which a b-error program can be identified from a single c-noisy text can also be learned from three texts, at least two of which are c-noisy, provided we are prepared to tolerate up to twice the number ($2b$) of errors in the final program.

THEOREM 5.9. *Let $b, c \in N$. Then $\mathbf{N}^c\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{2b}$.*

Proof. Suppose machine $\mathbf{M}$ $\mathbf{N}^c\mathbf{Ex}^b$-identifies $\mathcal{C}$. We construct $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{2b}$-identifies $\mathcal{C}$.

Let T_1, T_2, and T_3 be the three given texts out of which at least two are c-noisy for $f \in \mathcal{C}$.

Let i and j be distinct and such that $\mathbf{M}(T_i)\downarrow$ and $\mathbf{M}(T_j)\downarrow$, $P_i = \mathbf{M}(T_i)$ and $P_j = \mathbf{M}(T_j)$, and $\text{card}(\{x \mid \varphi_{P_i}(x)\downarrow \neq \varphi_{P_j}(x)\downarrow\}) \leq 2b$. Let $CLASH = \{x \mid \varphi_{P_i}(x)\downarrow \neq \varphi_{P_j}(x)\downarrow\}$. Let $S \subseteq CLASH$ be a set of cardinality $\lfloor \frac{\text{card}(CLASH)}{2} \rfloor$.

Note that we can easily determine all of the sets defined above and P_i and P_j as above in the limit from the given texts.

$\mathbf{M}'$ in the limit outputs a program p such that

$$\varphi_p(x) = \begin{cases} \varphi_{P_i}(x) & \text{if } x \in S; \\ \varphi_{P_j}(x) & \text{if } x \in CLASH - S; \\ \varphi_{P_i}(x) & x \in N - CLASH \ \wedge \ x \in \text{domain}(\varphi_{P_i}); \\ \varphi_{P_j}(x) & x \in N - CLASH \ \wedge \ x \in \text{domain}(\varphi_{P_j}); \\ \uparrow & \text{otherwise.} \end{cases}$$

Since at least one of P_i and P_j is a b-error program for f, it is clear that p is a $2b$-error program for f. Thus $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{2b}$-identifies f. □

In order to prove the rest of the results in this section (Theorems 5.10–5.14), it is useful to define certain sets and texts. We will require these sets to satisfy certain assumptions which can be assumed *without loss of generality*. We state these definitions and assumptions next, and we assume them without explicitly stating them in the proofs.

Suppose a machine is trying to $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$-identify f (where $c \in N$). Suppose the three texts provided as input to the machine are T_1, T_2, and T_3, such that at least two of these texts are c-noisy for f.

DEFINITIONS. *Let T_1, T_2, and T_3 be texts.*

1. T_{UNION} *is a text such that* $\text{content}(T_{UNION}) = \{(x,y) \mid (x,y) \in \text{content}(T_1) \ \vee \ (x,y) \in \text{content}(T_2) \ \vee \ (x,y) \in \text{content}(T_3)\}$.
2. $NOISE\_POINTS = \{x \mid (\exists y, z)[y \neq z \ \wedge \ (x,y) \in \text{content}(T_{UNION}) \ \wedge \ (x,z) \in \text{content}(T_{UNION})]\}$.
3. $NOISE_{1,2} = \{x \mid \text{card}(\{y \mid (x,y) \in \text{content}(T_1)\}) = \text{card}(\{y \mid (x,y) \in \text{content}(T_2)\}) \geq 2 \ \wedge \ \text{card}(\{y \mid (x,y) \in \text{content}(T_3)\}) = 1\}$.
4. $NOISE_{1,3} = \{x \mid \text{card}(\{y \mid (x,y) \in \text{content}(T_1)\}) = \text{card}(\{y \mid (x,y) \in \text{content}(T_3)\}) \geq 2 \ \wedge \ \text{card}(\{y \mid (x,y) \in \text{content}(T_2)\}) = 1\}$.
5. $NOISE_{2,3} = \{x \mid \text{card}(\{y \mid (x,y) \in \text{content}(T_2)\}) = \text{card}(\{y \mid (x,y) \in \text{content}(T_3)\}) \geq 2 \ \wedge \ \text{card}(\{y \mid (x,y) \in \text{content}(T_1)\}) = 1\}$.
6. $NOISE_{1,2,3} = NOISE\_POINTS - (NOISE_{1,2} \cup NOISE_{1,3} \cup NOISE_{2,3})$.

Assumptions. We make the following assumptions about the notions defined above.

1. For all $i \in \{1,2,3\}$, $\text{card}(\{x \mid (\exists y, z)[y \neq z \ \wedge \ (x,y) \in \text{content}(T_i) \ \wedge \ (x,z) \in \text{content}(T_i)]\}) \leq c$ (since if the cardinality of noise points is greater than c, then we know which two texts are c-noisy texts for f).
2. For all $i, j \in \{1,2,3\}$ and for all $x \in NOISE\_POINTS$, $\text{card}(\{y \mid (x,y) \in \text{content}(T_i) \cap \text{content}(T_j)\}) \geq 1$. (This is so because otherwise at least one of T_i and T_j is not a c-noisy text for f, which implies that the remaining text is a c-noisy text for f.)
3. For all x and y, $\text{card}(\{i \mid (x,y) \in \text{content}(T_i)\}) \neq 1$. (Otherwise, surely $f(x) \neq y$ and we can thus drop (x,y) from consideration.)

Note that if the above assumptions do not hold, then we can convert the texts (in the limit) to T_1', T_2', and T_3' such that at least two of these texts are c-noisy for f and T_1', T_2', and T_3' satisfy the assumptions given above.

The next theorem is a variation on Theorem 5.9 in which we compensate for multiple texts by allowing extra errors in the final program. In this case, instead of allowing twice the number of errors in the final program, the upper bound on the number of errors is expressed in terms of the error bound for single text, b, and the noise level of the single text, c. We would like to note that the simulation carried out in Theorem 5.10 below is optimal in the following sense. If we keep the bound on the

number of inaccuracies in the texts the same for the single text and for the acceptable texts in the multiple case and if $c \leq b$, then for all of the collections of functions for which a b-error program can be identified from a single c-noisy text, a $b + \lceil \frac{c}{2} \rceil$-error program can also be identified from three texts, at least two of which are only c-noisy. The optimality of the result is in the sense that this simulation fails if we reduce the bound on the number of errors allowed in the final program (of the multiple case) by even one. This was the diagonalization result described in Corollary 5.5.

THEOREM 5.10. *Let $b, c \in N$. Then $\mathbf{N}^c\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil}$.*

Proof. Suppose machine $\mathbf{M}$ $\mathbf{N}^c\mathbf{Ex}^b$-identifies $\mathcal{C}$. We construct $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil}$-identifies $\mathcal{C}$. Let T_1, T_2, and T_3 be three given texts out of which at least two are c-noisy for f.

Let $m = \min(\{\text{card}(NOISE_{1,2}), \text{card}(NOISE_{1,3}), \text{card}(NOISE_{2,3})\})$.

Let S be a set of cardinality $3m$ such that $\text{card}(S \cap NOISE_{1,2}) = \text{card}(S \cap NOISE_{1,3}) = \text{card}(S \cap NOISE_{2,3})$.

Let $X \subseteq NOISE\_POINTS - S$ be a set of cardinality $\lceil \frac{\text{card}(NOISE\_POINTS)-3m}{2} \rceil$.

Let i and j be distinct and such that $\mathbf{M}(T_i)\downarrow$ and $\mathbf{M}(T_j)\downarrow$, $P_i = \mathbf{M}(T_i)$ and $P_j = \mathbf{M}(T_j)$, and $\text{card}(\{x \mid \varphi_{P_i}(x)\downarrow \neq \varphi_{P_j}(x)\downarrow \ \wedge \ x \notin NOISE\_POINTS\}) \leq 2b$. Let $CLASH = \{x \mid \varphi_{P_i}(x)\downarrow \neq \varphi_{P_j}(x)\downarrow \ \wedge \ x \notin NOISE\_POINTS\}$.

Note that we can easily determine all the sets defined above and P_i and P_j as above in the limit from the given texts. $\mathbf{M}'$ in the limit outputs a program p such that

$$\varphi_p(x) = \begin{cases} \varphi_{P_i}(x) & \text{if } x \in X; \\ \varphi_{P_j}(x) & \text{if } x \in NOISE\_POINTS - (X \cup S); \\ y & \text{if } x \in CLASH \ \wedge \ (x, y) \in \text{content}(T_{UNION}); \\ \varphi_{P_i}(x) & \text{if } x \in N - (NOISE\_POINTS \cup CLASH) \text{ and} \\ & x \in \text{domain}(\varphi_{P_i}); \\ \varphi_{P_j}(x) & \text{if } x \in N - (NOISE\_POINTS \cup CLASH) \text{ and} \\ & x \in \text{domain}(\varphi_{P_j}); \\ y & \text{if } x \in S \ \wedge \ (x, y) \in \text{content}(T_1) \cap \text{content}(T_2) \cap \text{content}(T_3); \\ \uparrow & \text{otherwise.} \end{cases}$$

Since at least one of P_i and P_j computes a b-variant of f, the number of errors committed by the φ program $p \leq b + m + \lceil \frac{\text{card}(NOISE\_POINTS-S)}{2} \rceil$, which is at most $b + \lceil \frac{c}{2} \rceil$. ∎

The next theorem uses a combination of increasing the bound on the number of errors in the final program and having a smaller noise level than in the case of single text to compensate for the restrictive effects of multiple texts. The theorem shows that given $b, c, d \in N$ such that b is smaller, but not too much smaller, than c (that is, $\frac{c}{2} \leq b \leq c$), all of the collections of functions for which a b-error program can identified from a single $(2c-1)$-noisy text can also be identified from three texts, at least two of which are c-noisy, provided we are prepared to tolerate a bound of $\geq \max(\{\frac{4b}{3}, 2b - \frac{c}{2}\})$ on the number of errors in the final program. (Compare the following theorem with Corollary 5.6.)

THEOREM 5.11. *Let $b, c, d \in N, \frac{c}{2} \leq b \leq c$, be given. Then $\mathbf{N}^{2c-1}\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$ if $d \geq \max(\{\frac{4b}{3}, 2b - \frac{c}{2}\})$.*

Proof. Suppose that machine $\mathbf{M}$ $\mathbf{N}^{2c-1}\mathbf{Ex}^b$-identifies $\mathcal{C}$. We construct machine $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$-identifies $\mathcal{C}$. Let T_1, T_2, and T_3 be three texts given for a function $f \in \mathcal{C}$ such that at least two of the three texts are c-noisy for f.

Note that T_{UNION} is a noisy text for f. If T_{UNION} is a $(2c-1)$-noisy text for f, then $\mathbf{M}'$ can just output the programs output by $\mathbf{M}$ on T_{UNION} to identify f with at most d errors. Thus assume that T_{UNION} so formed contains more than $2c-1$ noisy elements. A simple calculation shows that the amount of noise in T_{UNION} is bounded by $\frac{3c+\mathrm{card}(NOISE_{1,2,3})}{2}$. If $\mathrm{card}(NOISE_{1,2,3}) < c$, then T_{UNION} is a $(2c-1)$-noisy text for f. Thus $\mathrm{card}(NOISE_{1,2,3}) = c$. Also, for all $x \in NOISE_{1,2,3}$, we can assume that there exist distinct y_1^x, y_2^x, and y_3^x such that $(x, y_1^x) \in \mathrm{content}(T_2) \cap \mathrm{content}(T_3)$, $(x, y_2^x) \in \mathrm{content}(T_1) \cap \mathrm{content}(T_3)$, and $(x, y_3^x) \in \mathrm{content}(T_1) \cap \mathrm{content}(T_2)$. (Otherwise, T_{UNION} contains at most $2c-1$ noisy elements.)

Let i, j be distinct and such that $\mathbf{M}(T_i)\downarrow$ and $\mathbf{M}(T_j)\downarrow$, $P_i = \mathbf{M}(T_i)$ and $P_j = \mathbf{M}(T_j)$, and $\mathrm{card}(\{x \mid \varphi_{P_i}(x)\downarrow \neq \varphi_{P_j}(x)\downarrow \ \wedge \ x \notin NOISE\_POINTS\}) \leq 2b$. Let $CLASH = \{x \mid [\varphi_{P_i}(x)\downarrow \neq \varphi_{P_j}(x)\downarrow] \wedge [x \notin NOISE\_POINTS]\}$. For $m \in \{i, j\}$, let $\delta_m = \mathrm{card}(\{x \mid x \in NOISE\_POINTS \wedge [(x, \varphi_{P_m}(x)) \in \mathrm{content}(T_i) \cap \mathrm{content}(T_j)]\})$. Let $\delta = \min(\delta_i, \delta_j)$. We can suppose without loss of generality that $c - b \leq \delta \leq b$. (Otherwise, we can determine a text which is guaranteed to be c-noisy for f. For example, suppose that $\delta = \delta_i < c - b$. Then clearly one of T_i and T_j is not a c-noisy text for f. This implies that the remaining text is a c-noisy text for f. Other cases are similar. In any of these cases, $\mathbf{M}'$ can simply input the correct c-noisy text obtained by the above analysis into $\mathbf{M}$ and arrive (in the limit) at a program which is at most b ($\leq d$) variant of f.)

We will pick a $\mu \in N$, $\mu \leq c$, depending on δ. Let $S \subseteq NOISE\_POINTS$ be a set of cardinality μ. Let $X \subseteq NOISE\_POINTS - S$ be a set of cardinality $\lceil \frac{c-\mu}{2} \rceil$.

$\mathbf{M}'$ in the limit outputs a program p_μ such that the following holds:

$$\varphi_{p_\mu}(x) = \begin{cases} y_k^x & \text{if } x \in S \ \wedge \ k \in \{1,2,3\} - \{i,j\}; \\ y_i^x & \text{if } x \in X; \\ y_j^x & \text{if } x \in NOISE\_POINTS - (X \cup S); \\ y & \text{if } x \in CLASH \ \wedge \ (x,y) \in \mathrm{content}(T_{UNION}); \\ \varphi_{P_i}(x) & \text{if } x \in N - (NOISE\_POINTS \cup CLASH) \text{ and} \\ & x \in \mathrm{domain}(\varphi_{P_i}); \\ \varphi_{P_j}(x) & \text{if } x \in N - (NOISE\_POINTS \cup CLASH) \text{ and} \\ & x \in \mathrm{domain}(\varphi_{P_j}); \\ \uparrow & \text{otherwise.} \end{cases}$$

Now the number of errors committed by $\mathbf{M}'$ on f is bounded by $maxerr = \max(\{b-\mu+\delta, b-\delta+\mu+\lceil\frac{c-\mu}{2}\rceil\}) = \lceil\max(\{b-\mu+\delta, b-\delta+\mu+\frac{c-\mu}{2}\})\rceil$. By choosing $\mu = \max(\{0, \lfloor\frac{4\delta-c}{3}\rfloor\})$, we have $maxerr = \lceil\max(\{b+\frac{c-\delta}{3}, b-\delta+\frac{c}{2}\})\rceil$. The value of $maxerr$ is maximized for $\delta = c-b$. Thus $maxerr = \lceil\max(\{\frac{4b}{3}, 2b-\frac{c}{2}\})\rceil$. Thus if $d \geq \lceil\max(\{\frac{4b}{3}, 2b-\frac{c}{2}\})\rceil$, then $\mathbf{N}^{2c-1}\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^d$. □

We now present three theorems that use the technique of making the bound on the noise level of multiple texts smaller than the bound on the noise level of the single text to compensate for the restrictive effect of multiple texts.

The result below says that for $b, c \in N$ such that b is fairly small compared to c (that is, $b < \lceil\frac{c}{2}\rceil$), if for some collection of functions a b-error program can be identified from a single $(2c-1)$-noisy text, then a b-error program can also be identified from three texts, at least two of which are c-noisy.

THEOREM 5.12. *For* $b < \lceil\frac{c}{2}\rceil$, $\mathbf{N}^{2c-1}\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^b$.

Proof. Suppose machine $\mathbf{M}$ $\mathbf{N}^{2c-1}\mathbf{Ex}^b$-identifies $\mathcal{C}$. We describe $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^b$-identifies $\mathcal{C}$. Let T_1, T_2, and T_3 be the three given texts out of which at least two are c-noisy for $f \in \mathcal{C}$.

As in Theorem 5.11, it can be shown that either T_{UNION} is a $(2c-1)$-noisy for f or else there exists a set X of cardinality c and distinct y_1^x, y_2^x, and y_3^x for each $x \in X$ such that $(\forall x \in X)(\forall i \in \{1,2,3\})(\forall y \in \{y_1^x, y_2^x, y_3^x\} - \{y_i^x\})[(x,y) \in \text{content}(T_i)]$. Now let $i \in \{1,2,3\}$ be such that $\mathbf{M}(T_i)\downarrow$. Let $\mathbf{M}(T_i) = P_i$. For $j \in \{1,2,3\} - \{i\}$, let $D_j = \{x \in X \mid \varphi_{P_i}(x) = y_j^x\}$. Let $j \in \{1,2,3\} - \{i\}$ be such that $\text{card}(D_j) \leq \frac{c}{2}$. Note that such a j exists. Now for all $x \in X$, $f(x) \neq y_j^x$ (since otherwise T_i is a c-noisy text for f and thus φ_{P_i} should be a b-variant of f). Hence T_j is a c-noisy text for f. Thus $\mathbf{M}(T_j)\downarrow$ and $\varphi_{\mathbf{M}(T_j)}$ is a b-variant of f. Clearly, in all the above cases, $\mathbf{M}'$ can in the limit easily output a program which is a b-variant of f. □

Theorem 5.13 is like Theorem 5.12, but it does not require that error bound on the final program, b, be fairly small compared to the bound on the noise level of the multiple texts, c. It says that for $b, c \in N$, if for some collection of functions a b-error program can be identified from a single $(2c)$-noisy text, then a b-error program can also be identified from three texts, at least two of which are c-noisy.

THEOREM 5.13. *Let $b, c \in N$. $\mathbf{N}^{2c}\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^b$.*

Proof. Suppose machine $\mathbf{M}$ $\mathbf{N}^{2c}\mathbf{Ex}^b$-identifies $\mathcal{C}$. We construct $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^b$-identifies $\mathcal{C}$.

Let T_1, T_2, and T_3 be three texts given for a function $f \in \mathcal{C}$ such that at least two of the three texts are c-noisy for f. A simple calculation shows that the amount of noise in T_{UNION} (with respect to the graph of f) is bounded by $\frac{3c+\text{card}(NOISE_{1,2,3})}{2}$. Noting that $\text{card}(NOISE_{1,2,3})$ is bounded by c, it follows that T_{UNION} is a $2c$-noisy text for f. $\mathbf{M}'$ outputs in the limit whatever is output by $\mathbf{M}$ in the limit on the text T_{UNION}. Thus $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^b$-identifies f. This proves the theorem. □

Using a proof technique similar to the above, we can also establish the following variation of the above result. Here the bound on the noise level of the single text is expressed in terms of both the bound in the noise level of the multiple text (c) and the bound on the number of errors in the final program (b). It says that for $b, c \in N$, if for some collection of functions a b-error program can be identified from a single $\frac{3c+2b}{2}$-noisy text, then a b-error program can also be identified from three texts, at least two of which are c noisy. We omit the proof.

THEOREM 5.14. *Let $b, c \in N$. Then $\mathbf{N}^{\frac{3c+2b}{2}}\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{N}^c\mathbf{Ex}^b$.*

5.2.2. Incomplete texts. In this section, we consider the effects of learning from multiple texts on incomplete data. We first show a result that gives cases in which identification from multiple incomplete texts is a restriction on identification from a single incomplete text. We then give results which show the cases where the restrictive effects of multiple inaccurate texts can be compensated. Unfortunately, our results about incomplete texts are not as extensive as those for noisy texts.

The following theorem shows that given b and c such that $b \geq c > 1$, there are collections of functions for which a b-error program can be identified from a single $(\lfloor \frac{3c}{2} \rfloor - 1)$-incomplete text but for which even a $(b + \lceil \frac{c}{2} \rceil - 1)$-error program cannot be identified from three texts, at least two of which are only c-incomplete for the function being learned.

THEOREM 5.15. *Let $b, c \in N, b \geq c > 1$. Then $\mathbf{In}^{\lfloor \frac{3c}{2} \rfloor - 1}\mathbf{Ex}^b - \mathbf{Mul}_3^2\mathbf{In}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil - 1} \neq \emptyset$.*

Proof. The proof uses ideas similar to those used in the proof of Theorem 5.4. Suppose b and c are given as in the hypothesis. Let $A = \{\langle 0, x\rangle \mid 0 \leq x < \lfloor \frac{3c}{2} \rfloor\}$, $A_1 = \{\langle 0, x\rangle \mid 0 \leq x < \lfloor \frac{c}{2} \rfloor\}$, $A_2 = \{\langle 0, x\rangle \mid \lfloor \frac{c}{2} \rfloor \leq x < 2\lfloor \frac{c}{2} \rfloor\}$, $A_3 = \{\langle 0, x\rangle \mid 2\lfloor \frac{c}{2} \rfloor \leq x < 3\lfloor \frac{c}{2} \rfloor\}$, and $A_4 = \{\langle 0, x\rangle \mid 3\lfloor \frac{c}{2} \rfloor \leq x < \lfloor \frac{3c}{2} \rfloor\}$.

Note that $\mathrm{card}(A_1) = \mathrm{card}(A_2) = \mathrm{card}(A_3) = \lfloor \frac{c}{2} \rfloor$ and $\mathrm{card}(A_4) = 1$ if c is odd and 0 otherwise.

Consider the following class of functions:

$\mathcal{C} = \{f \in \mathcal{R} \mid$ the following conditions hold:

(1) $(\forall i \in \{1,2,3\})(\forall x \in A_i)[f(x) = f(\langle 0, (i-1)\lfloor \frac{c}{2} \rfloor \rangle) \in \{0,1\}]$;

(2) $(\exists! i \in \{1,2,3\})[f(A_i) = 1]$;

(3) $(\forall i \in \{1,2,3\} \mid f(A_i) = 0)[(\max(\{f(\langle i, \langle x, 0\rangle\rangle) \mid x \in N\})$ exists) $\wedge$ $(\varphi_{\max(\{f(\langle i,\langle x,0\rangle\rangle)\mid x\in N\})} =^b f)]$;

(4) $(\forall i \in \{1,2,3\})(\forall x,y,z)[f(\langle i, \langle x, y\rangle\rangle) = f(\langle i, \langle x, z\rangle\rangle)]$;

(5) $(\forall x \in A_4)[\varphi_{f(x)} =^b f]$ $\}$

Clearly, $\mathcal{C} \in \mathbf{In}^{\lfloor \frac{3c}{2} \rfloor - 1}\mathbf{Ex}^b$. We now show that $\mathcal{C} \notin \mathbf{Mul}_3^2\mathbf{In}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil - 1}$.

Suppose by way of contradiction that machine $\mathbf{M}$ $\mathbf{Mul}_3^2\mathbf{In}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil - 1}$-identifies $\mathcal{C}$. Then by the operator recursion theorem [5], there exists a recursive 1–1 increasing p, $p(0) > 1$, defined below in stages.

We define $\varphi_{p(i)}$ in stages $s \geq 3$.

Let x_3^s denote the least x such that for all y, $\varphi_{p(1)}(\langle 3, \langle x, y\rangle\rangle)$ is not defined before stage s.

Let $\varphi_{p(1)}(x) = 0$ for $x \in A_1 \cup A_3$; $\varphi_{p(1)}(x) = 1$ for $x \in A_2$; $\varphi_{p(1)}(\langle 1, \langle 0,0\rangle\rangle) = p(1)$; $\varphi_{p(1)}(\langle 2, \langle 0,0\rangle\rangle) = p(2)$; $\varphi_{p(1)}(x) = p(1)$ for $x \in A_4$.

Let $\varphi_{p(2)}(x) = 0$, for $x \in A_2 \cup A_3$; $\varphi_{p(2)}(x) = 1$ for $x \in A_1$; $\varphi_{p(2)}(\langle 1, \langle 0,0\rangle\rangle) = p(1)$; $\varphi_{p(2)}(\langle 2, \langle 0,0\rangle\rangle) = p(2)$; $\varphi_{p(2)}(x) = p(2)$ for $x \in A_4$.

Let σ_1^3, σ_2^3, and σ_3^3 be initial segments such that

$$\mathrm{content}(\sigma_1^3) = \{(x,0) \mid x \in A_1\} \cup \{(\langle 1, \langle 0,0\rangle\rangle, p(1)), (\langle 2, \langle 0,0\rangle\rangle, p(2))\}.$$
$$\mathrm{content}(\sigma_2^3) = \{(x,0) \mid x \in A_2\} \cup \{(\langle 1, \langle 0,0\rangle\rangle, p(1)), (\langle 2, \langle 0,0\rangle\rangle, p(2))\}.$$
$$\mathrm{content}(\sigma_3^3) = \{(x,0) \mid x \in A_3\} \cup \{(\langle 1, \langle 0,0\rangle\rangle, p(1)), (\langle 2, \langle 0,0\rangle\rangle, p(2))\}.$$

Let $\varphi_{p(i)}^s$ denote the part of $\varphi_{p(i)}$ defined before stage s. Go to stage 3.

Begin stage s

1. Let $q = \mathbf{M}(\sigma_1^s, \sigma_2^s, \sigma_3^s)$.

2. Let $D_1 = \{x \mid \Phi_q(x) \leq s \wedge ((x \in A_1 \wedge \varphi_q(x) = 0) \vee (x \in A_2 \wedge \varphi_q(x) = 1) \vee (x \in A_4 \wedge \varphi_q(x) = p(1)))\}$. Let $D_2 = \{x \mid \Phi_q(x) \leq s \wedge ((x \in A_1 \wedge \varphi_q(x) = 1) \vee (x \in A_2 \wedge \varphi_q(x) = 0) \vee (x \in A_4 \wedge \varphi_q(x) = p(2)))\}$. If $\mathrm{card}(D_1) \geq \mathrm{card}(D_2)$, then let $r = 1$ and $r' = 2$; otherwise, let $r = 2$ and $r' = 1$. Let $X^s \subset \{\langle 4, x\rangle \mid x \geq s\}$ denote the (lexicographically least) set of b elements such that for all $x \in X^s$, $\varphi_{p(1)}(x)$ is not defined before stage s.

3. Let $\varphi_{p(s)}(x) = 0$ for $x \in A_3 \cup A_{r'}$; $\varphi_{p(s)}(x) = 1$ for $x \in A_r$; $\varphi_{p(s)}(x) = p(r')$ for $x \in A_4$; $\varphi_{p(s)}(\langle 3, \langle x_3^s, 0\rangle\rangle) = \varphi_{p(1)}(\langle 3, \langle x_3^s, 0\rangle\rangle) = \varphi_{p(2)}(\langle 3, \langle x_3^s, 0\rangle\rangle) = p(s)$. For $x \in \mathrm{domain}(\varphi_{p(1)}^s) - A$, let $\varphi_{p(s)}(x) = \varphi_{p(1)}(x)$.

4. Let σ_i' be an extension of σ_i^s such that $\mathrm{content}(\sigma_i') = \mathrm{content}(\sigma_i^s) \cup \{(\langle 3, \langle x_3^s, 0\rangle\rangle, p(s))\}$.

5. Dovetail steps 5a, 5b, and 5c until (if ever) one of steps 5a and 5b succeeds and at least one iteration of the loop at step 5c is completed. If step 5a succeeds, (before step 5b succeeds (if ever)), then complete the then current iteration of the repeat loop in step 5c and go to step 6a. If step 5b succeeds

(before step 5a succeeds (if ever)), then complete the then current iteration of the repeat loop in step 5c and go to step 6b.

5a. Search for $\sigma_i'' \supseteq \sigma_i'$, $i \in \{1,2,3\}$, such that $\mathbf{M}(\sigma_1^s, \sigma_2^s, \sigma_3^s) \neq \mathbf{M}(\sigma_1'', \sigma_2'', \sigma_3'')$ and for all $j, j' \in \{1,2,3\}$, for all $x, y, z \in N$, the following hold:

- $\text{content}(\sigma_j'') - \text{content}(\sigma_j') = \text{content}(\sigma_{j'}'') - \text{content}(\sigma_{j'}')$.
- If $(x,y), (x,z) \in \text{content}\ (\sigma_j'')$, then $y = z$.
- If $(\langle j, \langle x, y\rangle\rangle, v) \in \text{content}(\sigma_1'')$ and $(\langle j, \langle x, z\rangle\rangle, w) \in \text{content}(\sigma_1'')$, then $v = w$.
- If $(x,y) \in (\text{content}(\sigma_1'') - \text{content}(\sigma_1'))$, then $[x \notin A] \wedge [y = 0$ OR $x \in X^s$ OR [for some $i \in \{1,2,3\}$, $u, v \in N$: $x = \langle i, \langle u, v\rangle\rangle$ and $\varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle)$ has been defined until now and $\varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle) = y]]$.

Intuitively, this substep attempts to look for a mind change.

5b. Search for $x \in (A_1 \cup A_2) - (D_1 \cup D_2)$ such that $\varphi_q(x)\downarrow$.

5c. Let $Addednew = \emptyset$.
(*Note:* We use *Addednew* in order to identify the set of elements for which we extended $\varphi_{p(1)}$ in this step. *Addednew* will be used (in steps 6a and 6b), provided at least one of steps 5a or 5b succeeds).
Repeat
Let x be the least point not in X^s such that $\varphi_{p(1)}(x)$ is not defined until now. If for some $i \in \{1,2,3\}$, x is of the form $\langle i, \langle u, v\rangle\rangle$ and $\varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle)$ is defined until now, then let $\varphi_{p(1)}(x) = \varphi_{p(2)}(x) = \varphi_{p(s)}(x) = \varphi_{p(1)}(\langle i, \langle u, 0\rangle\rangle)$; otherwise, let $\varphi_{p(1)}(x) = \varphi_{p(2)}(x) = \varphi_{p(s)}(x) = 0$.
Let $Addednew = Addednew \cup \{x\}$.
Forever

6a. For all x and y such that $(x,y) \in \text{content}(\sigma_1'') - \text{content}(\sigma_1^s)$, let $\varphi_{p(1)}(x) = \varphi_{p(2)}(x) = y$. Let σ_i^{s+1} be an extension of σ_i'' such that $\text{content}(\sigma_i^{s+1}) - \text{content}(\sigma_i'') = \{(x, \varphi_{p(1)}(x)) \mid x \in Addednew\}$. Go to stage $s+1$.

6b. Let σ_i^{s+1} be an extension of σ_i' such that $\text{content}(\sigma_i^{s+1}) - \text{content}(\sigma_i') = \{(x, \varphi_{p(1)}(x)) \mid x \in Addednew\}$. Go to stage $s+1$.

End stage s
Now consider the following cases.
Case 1. All stages terminate.
Let

$$f(x) = \begin{cases} 0 & \text{if } x \in A_1 \cup A_2; \\ 1 & \text{if } x \in A_3; \\ \varphi_{p(1)}(x) & \text{otherwise.} \end{cases}$$

Clearly, $f \in \mathcal{C}$ (since $c \leq b$, $\varphi_{p(1)} =^b f$, and $\varphi_{p(2)} =^b f$). Also, $\mathbf{M}$ on f makes infinitely many mind changes since Step 5b can succeed at most finitely many times before Step 5a succeeds.

Case 2. Stage s (≥ 3) is the least stage which never terminates.

In this case, $\text{domain}(\varphi_{p(s)}) = N - X^s$. Let q, D_1, D_2, r, and r' be as defined in steps 1 and 2 of stage s. Consider any total extension f of $\varphi_{p(s)}$ such that $(\forall x \in X^s)[\varphi_q(x) \neq f(x)]$.

Clearly, $f \in \mathcal{C}$ (since $\varphi_{p(r')} =^b f$ and $\varphi_{p(s)} =^b f$). Let T_1, T_2, and T_3 be extensions of σ'_1, σ'_2, and σ'_3 (as defined in step 4 in stage s), respectively, such that $\text{content}(T_1) - \text{content}(\sigma^s_1) = \{(x, y) \mid f(x) = y \ \wedge \ x \notin A \ \wedge \ (\forall z)[(x, z) \notin \text{content}(\sigma^s_1)]\}$ and for all j, the $[|\sigma'_1| + j]$th element of T_1, the $[|\sigma'_2| + j]$th element of T_2, and the $[|\sigma'_3| + j]$th element of T_3 are all same. First, notice that for all such f and corresponding T_1, T_2, and T_3, $\mathbf{M}(T_1, T_2, T_3) = q$. (Otherwise, step 5a would succeed.) Second, notice that for all $x \in (A_1 \cup A_2) - (D_1 \cup D_2)$, $\varphi_q(x)\uparrow$. (Otherwise, step 5b would succeed.) Third, from the choice of r and r', it follows that program q makes at least $\lceil \frac{c}{2} \rceil$ errors on $A_1 \cup A_2 \cup A_4$. Also, from the way f has been chosen, we have $(\forall x \in X^s)[f(x) \neq \varphi_q(x)]$. Thus $\varphi_q \neq^{b+\lceil \frac{c}{2} \rceil - 1} f$.

From the above cases, it follows that $\mathcal{C} \notin \mathbf{Mul}^2_3\mathbf{In}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil - 1}$. □

Theorem 5.15 demonstrated cases in which identification from multiple inaccurate texts is a restriction over identification from a single inaccurate text. As in the case with noisy texts, we would like to explore situations in which the deleterious effects of learning from multiple texts can be compensated. Again as in the case of noisy texts, this can be achieved in two different ways:

- by increasing the bound on the number of errors in the final program allowed in identification from multiple texts;
- by increasing the bound on missing data in the single incomplete text (this has the same effect as decreasing the bound on missing data in the multiple texts).

Theorem 5.16 below uses the second technique and Theorems 5.19 and 5.20 use the first technique to compensate for the effects of multiple texts. The technique used in these simulation arguments is to first collect the input points on which the converging programs differ, then to construct a program which judiciously chooses the output on these inputs based on the input texts and outputs of the converging programs.

THEOREM 5.16. $\mathbf{In}^{c+\lfloor \frac{c}{2} \rfloor}\mathbf{Ex}^b \subseteq \mathbf{Mul}^2_3\mathbf{In}^c\mathbf{Ex}^b$.

Theorem 5.16 says that given b and c, if for a collection of functions a b-error program can be identified from a single $(c + \lfloor \frac{c}{2} \rfloor)$-incomplete text, then a b-error program can also be identified from three texts, at least two of which are c-incomplete for the function being learned. A proof of this result requires the notion of a stabilizing sequence, which we introduce next. Let L range over sets of ordered pairs.

DEFINITION 5.17. (See [9].) *σ is said to be a stabilizing sequence for $\mathbf{M}$ on L* $\iff$ $[\text{content}(\sigma) \subseteq L$ *and* $(\forall \tau \mid \sigma \subset \tau \ \wedge \ \text{content}(\tau) \subseteq L)[\mathbf{M}(\tau) = \mathbf{M}(\sigma)]]$.

The following lemma is based on a similar lemma by Blum and Blum [3] (see also Osherson and Weinstein [18]).

LEMMA 5.18 (based on a similar lemma in [3, 18]). *Suppose $\mathbf{M}$ $\mathbf{In}^a\mathbf{Ex}^b$-identifies f. Let L be such that $L \subseteq \{(x, y) \mid f(x) = y\} \ \wedge \ \text{card}(\{(x, y) \mid f(x) = y\} - L) \leq a$. Then there exists a stabilizing sequence for $\mathbf{M}$ on L. Moreover, if σ is a stabilizing sequence for $\mathbf{M}$ on L, then $\varphi_{\mathbf{M}(\sigma)} =^b f$.*

We now give a proof of Theorem 5.16.

Proof of Theorem 5.16. Suppose machine $\mathbf{M}$ $\mathbf{In}^{c+\lfloor \frac{c}{2} \rfloor}\mathbf{Ex}^b$-identifies $\mathcal{C}$. We construct $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}^2_3\mathbf{In}^c\mathbf{Ex}^b$-identifies $\mathcal{C}$.

Let T_1, T_2, and T_3 be three given texts out of which at least two are c-incomplete for $f \in \mathcal{C}$. Let T_{UNION} be a text such that $\text{content}(T_{UNION}) = \text{content}(T_1) \cup$

content$(T_2) \cup$ content(T_3). Without loss of generality, we assume that $(\forall x)[\text{card}(\{y \mid (x,y) \in \text{content}(T_{UNION})\}) \leq 1]$. (If there are x, y, z, i, and j such that $(x,y) \in \text{content}(T_i)$, $(x,z) \in \text{content}(T_j)$, and $y \neq z$, then at least one of T_i and T_j is not a c-incomplete text for f. Thus the third text is a c-incomplete text for f.) We can also assume without loss of generality that $(\forall x)[\text{card}(i \in \{1,2,3\} \mid (\exists y)[(x,y) \in \text{content}(T_i)]) \neq 2]$. (Otherwise, for such x's the value of $f(x)$ is known and thus without loss of generality we can put (x,y) into all three texts.)

In the following, we try to locate a stabilizing sequence for $\mathbf{M}$ on an appropriate subset of $\{(x, f(x)) \mid x \in N\}$. If one of the texts has a "high amount of incompleteness," then this may not be possible (see condition (iii) below); however, in this case, it is possible to determine the text which has a high amount of incompleteness.

Let $n', n'' \in N$, $X \subset \{0, 1, \ldots, n'\}$, and $\sigma \in \text{SEQ}$ be such that (i), (ii), and (iii) below are satisfied.

(i) $(\forall i \in \{1,2,3\})(\forall y \in N)(\forall x \in \{0,1,\ldots,n'\})[(x,y) \in \text{content}(T_i) \Rightarrow (x,y) \in \text{content}(T_i[n''])]$.

(ii) $(\forall x \in X)(\forall y \in N)[\text{card}(\{i \mid (x,y) \in \text{content}(T_i)\}) \leq 1]$.

(iii)

(A) $[(\exists j \in \{1,2,3\})[\text{card}(\{x \mid (x \in X) \ \wedge \ (\forall y \in N)[(x,y) \notin \text{content}(T_j)]\}) > c]]$

OR

(B) $[[[\text{content}(\sigma) \subseteq [\text{content}(T_{UNION}) \cap \{(x,y) \mid (y \in N) \ \wedge \ (x \leq n')\}] - [\{(x,y) \mid (x \in X) \ \wedge \ (y \in N)\}]]$ and σ is a stabilizing sequence for $\mathbf{M}$ on $L = [\text{content}(T_{UNION}) - \{(x,y) \mid (x \in X) \ \wedge \ (y \in N)\}]]$ AND $[(\forall x \leq n')[x \notin X \Rightarrow (\exists y)[\text{card}(\{i \mid (x,y) \in \text{content}(T_i)\}) = 3]]]]$.

It is easy to see that if such n', n'', X, and σ exist, then $\mathbf{M}'$ can find them in the limit. If in (iii) (A) holds, then clearly $\mathbf{M}'$ can know the two texts which are c-incomplete for f and thus output in the limit a b-error program for f. If in (iii) (B) holds, then we claim that $\mathbf{M}(\sigma)$ is a b-error program for f. This would be the case if $\text{card}(\{(x,y) \mid f(x) = y\} - L) \leq c + \lfloor \frac{c}{2} \rfloor$. To prove this, let $\delta_i = \text{card}(\{x \in X \mid (\exists y)[(x,y) \in \text{content}(T_i)]\})$. Let δ be the median of δ_1, δ_2, and δ_3. Now $\text{card}(\{(x,y) \mid f(x) = y\} - L) \leq c - (\text{card}(X) - \delta) + \text{card}(X) \leq c + \delta$ and $\delta \leq \frac{c}{2}$.

To complete the proof, we have to show that such n', n'', X, and σ indeed exist.

Case 1. There exists an $i \in \{1,2,3\}$ such that $\text{card}(\{x \mid (\forall y)[(x,y) \notin \text{content}(T_i)]\}) > c$.

Let n' be such that there exists an i, $\text{card}(\{x \leq n' \mid (\forall y)[(x,y) \notin \text{content}(T_i)]\}) = c + 1$. Let n'' be such that (i) above is satisfied. Let $X = \{x \leq n' \mid \text{card}(\{i \mid (\exists y)[(x,y) \in \text{content}(T_i)]\}) \leq 1\}$. Then it is easy to see that (i), (ii), and (iii) above are satisfied.

Case 2. There does not exist an $i \in \{1,2,3\}$ such that $\text{card}(\{x \mid (\forall y)[(x,y) \notin \text{content}(T_i)]\}) > c$.

Let $X = \{x \mid \text{card}(\{i \mid (\exists y)[(x,y) \in \text{content}(T_i)]\}) \leq 1\}$. Now let $L = \{(x,y) \mid f(x) = y \ \wedge \ x \notin X\}$ and let $\delta_i = \text{card}(\{x \in X \mid (\exists y)[(x,y) \in \text{content}(T_i)]\})$. Let δ be the median of δ_1, δ_2, and δ_3. Now $\text{card}(\{(x,y) \mid f(x) = y\} - L) \leq c - (\text{card}(X) - \delta) + \text{card}(X) \leq c + \delta$ and $\delta \leq \frac{c}{2}$. Thus there exists a stabilizing sequence σ for $\mathbf{M}$ on L. Let $n' = 1 + \max(X \cup \{x \mid (\exists y)[(x,y) \in \text{content}(\sigma)]\})$. Let n'' be chosen so as to satisfy (i) above. Clearly, n', n'', X, and σ satisfy (i), (ii), and (iii).

The above cases prove the existence of σ, X, n', and n'' satisfying (i), (ii), and (iii). □

We now present two results that compensate for the restrictive effects of multiple incomplete texts by allowing a larger bound on the number of errors allowed in the

final program.

THEOREM 5.19. *Let $b, c \in N$. Then* $\mathbf{In}^c\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{In}^c\mathbf{Ex}^{2b}$.

This result, which is a counterpart of Theorem 5.9, says that the collections of functions for which a b-error program can be identified from a c-incomplete text can also be identified from three texts, at least two of which are c-incomplete, provided we are prepared to tolerate twice ($2b$) the number of errors in the final program. This theorem can be proved using a technique similar to the one used in the proof of Theorem 5.9. We omit the details.

The next result gives a different bound on the number of errors allowed in the final program.

THEOREM 5.20. *Let $b, c \in N, b \geq c$ be given. Then* $\mathbf{In}^c\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{In}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil}$.

Proof. Suppose the hypothesis of the theorem holds. Suppose machine $\mathbf{M}$ $\mathbf{In}^c\mathbf{Ex}^b$-identifies $\mathcal{C}$. We construct $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{In}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil}$-identifies $\mathcal{C}$.

Let T_1, T_2, and T_3 be three texts given for a function $f \in \mathcal{C}$ such that at least two of the three texts are c-incomplete for f. Without loss of generality, we can assume that $(\forall x)[\mathrm{card}(\{y \mid (x,y) \in \mathrm{content}(T_1) \cup \mathrm{content}(T_2) \cup \mathrm{content}(T_3)\}) \leq 1]$. (If there is such an x, then let y, z, i, and j be such that $(x,y) \in \mathrm{content}(T_i), (x,z) \in \mathrm{content}(T_j)$, and $y \neq z$. Then at least one of T_i and T_j is not a c-incomplete text for f. Thus the third text is a c-incomplete text for f.)

Let T_{UNION} be a text such that $\mathrm{content}(T_{UNION}) = \{(x,y) \mid (x,y) \in \mathrm{content}(T_1) \vee (x,y) \in \mathrm{content}(T_2) \vee (x,y) \in \mathrm{content}(T_3)\}$. Let i and j be distinct and such that $\mathbf{M}(T_i)\downarrow$ and $\mathbf{M}(T_j)\downarrow$, $P_i = \mathbf{M}(T_i)$ and $P_j = \mathbf{M}(T_j)$, and $\mathrm{card}(CLASH = \{x \mid [\varphi_{P_i}(x)\downarrow \neq \varphi_{P_j}(x)\downarrow] \vee (\exists y)[[(x,y) \in \mathrm{content}(T_{UNION})] \wedge [[\varphi_{P_i}(x)\downarrow \neq y] \vee [\varphi_{P_j}(x)\downarrow \neq y]]]\}) \leq 3b$.

Let $VIS\_IN = CLASH \cap \{x \mid \mathrm{card}(\{i \mid (\exists y)(x,y) \in \mathrm{content}(T_i)\}) \leq 1\}$. For $i \in \{1,2,3\}$, let $IN_i = VIS\_IN \cap \{x \mid (\exists y)[(x,y) \in \mathrm{content}(T_i)]\}$. Let $m = \min(\mathrm{card}(IN_1), \mathrm{card}(IN_2), \mathrm{card}(IN_3))$. Let $S \subseteq VIS\_IN$ be a set of cardinality $3m$ such that $\mathrm{card}(S \cap IN_1) = \mathrm{card}(S \cap IN_2) = \mathrm{card}(S \cap IN_3)$. Let $X \subseteq VIS\_IN - S$ be a set of cardinality $\lceil \frac{\mathrm{card}(VIS\_IN - S)}{2} \rceil$.

Machine $\mathbf{M}'$ outputs (in the limit) program p as follows:

$$\varphi_p(x) = \begin{cases} \varphi_{P_i}(x) & \text{if } x \notin CLASH \ \wedge \ x \in \mathrm{domain}(\varphi_{P_i}); \\ \varphi_{P_j}(x) & \text{if } x \notin CLASH \ \wedge \ x \in \mathrm{domain}(\varphi_{P_j}); \\ y & \text{if } x \in ((CLASH - VIS\_IN) \cup S) \text{ and} \\ & (x,y) \in \mathrm{content}(T_{UNION}) \ ; \\ \varphi_{P_i}(x) & \text{if } x \in X; \\ \varphi_{P_j}(x) & \text{if } x \in VIS\_IN - (S \cup X); \\ \uparrow & \text{otherwise.} \end{cases}$$

Now the number of errors committed by $\mathbf{M}'$ on f is bounded by $m + \lceil \frac{\mathrm{card}(VIS\_IN - S)}{2} \rceil + b$. Observing that $c \geq \mathrm{card}(VIS\_IN) - m$, it follows that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{In}^c\mathbf{Ex}^{b+\lceil \frac{c}{2} \rceil}$-identifies $\mathcal{C}$. □

We end this section by presenting a somewhat surprising result which says that for $a \in N$ the collection of functions for which a 1-error program can be identified from a single a-incomplete text is exactly the same as the collection of functions for which a 1-error program can be identified from three texts, at least two of which are a-incomplete for the function being learned.

THEOREM 5.21. *Let $a \in N$. Then,* $\mathbf{In}^a\mathbf{Ex}^1 = \mathbf{Mul}_3^2\mathbf{In}^a\mathbf{Ex}^1$.

Proof. The case where $a = 1$ follows from Theorem 5.16. Suppose $a > 1$. Suppose $\mathbf{M}$ $\mathbf{In}^a\mathbf{Ex}^1$-identifies $\mathcal{C}$. We give an $\mathbf{M}'$ such that $\mathbf{M}'$ $\mathbf{Mul}_3^2\mathbf{In}^a\mathbf{Ex}^1$-identifies $\mathcal{C}$. Let

T_1, T_2, and T_3 be the three given texts out of which at least two are a-incomplete for $f \in \mathcal{C}$. Let T_{UNION} denote a text such that content(T_{UNION}) = content$(T_1) \cup$ content$(T_2) \cup$ content(T_3). Without loss of generality, assume that for all x, card$(\{y \mid (x,y) \in \text{content}(T_{UNION})\}) \leq 1$. (Otherwise, let y and z, $y \neq z$, be such that $(x,y) \in$ content(T_i) and $(x,z) \in$ content(T_j) (for $i,j \in \{1,2,3\}$). Clearly, at least one of T_i and T_j is not an a-incomplete text for f. Thus the remaining text is an a-incomplete text for f; in this case, $\mathbf{M}'$ can simply input the a-incomplete text for f to $\mathbf{M}$ to obtain a program (in the limit) which is 1-variant of f.) Also, assume that $(\forall x,y)[\text{card}(\{i \in \{1,2,3\} \mid (x,y) \in \text{content}(T_i)\}) \neq 2]$ (since otherwise $f(x) = y$ and we can assume that (x,y) is in all of the texts). Let i and j be distinct and such that $\mathbf{M}(T_i)\downarrow$ and $\mathbf{M}(T_j)\downarrow$. (Clearly, for $f \in \mathcal{C}$, such i and j exist and $\mathbf{M}'$ can determine i and j in the limit.) Without loss of generality, let $i = 1$ and $j = 2$. (Otherwise, simply rename the texts.) Let $\mathbf{M}(T_1) = P_1$ and $\mathbf{M}(T_2) = P_2$.

Let $CLASH12 = \{x \mid [\varphi_{P_1}(x)\downarrow \neq \varphi_{P_2}(x)\downarrow] \vee [\varphi_{P_1}(x)\downarrow \wedge (\exists y)[(x,y) \in \text{content}(T_2) \wedge y \neq \varphi_{P_1}(x)]] \vee [\varphi_{P_2}(x)\downarrow \wedge (\exists y)[(x,y) \in \text{content}(T_1) \wedge y \neq \varphi_{P_2}(x)]]\}$. Let $CLASH1T3 = \{x \mid [\varphi_{P_1}(x)\downarrow \wedge (\exists y)[(x,y) \in \text{content}(T_3) \wedge y \neq \varphi_{P_1}(x)]]\}$. Let $CLASH2T3 = \{x \mid [\varphi_{P_2}(x)\downarrow \wedge (\exists y)[(x,y) \in \text{content}(T_3) \wedge y \neq \varphi_{P_2}(x)]]\}$.

Now consider the following cases. (Note that $\mathbf{M}'$ can also determine in the limit which of the following cases holds.) In each of the following cases, we either give a correct text for f or a program for a 1-variant of f. For each of the case numbers $i > 1$, we assume without stating that all cases $< i$ do not hold.

Case 1. card$(CLASH12) > 2$.

In this case, T_3 is an a-incomplete text for f.

Case 2. card$(CLASH1T3) > 1$.

In this case, T_2 is an a-incomplete text for f.

Case 3. card$(CLASH2T3) > 1$.

In this case, T_1 is an a-incomplete text for f.

Case 4. card$(CLASH12) = 0$.

In this case, let p be such that $\varphi_p = \varphi_{P_1} \cup \varphi_{P_2}$. Then φ_p is a program for a 1-variant of f.

For Cases 5–7, assume without loss of generality that for all $x \in CLASH12 \cup CLASH1T3 \cup CLASH2T3$, $\varphi_{P_1}(x)\downarrow$ and $\varphi_{P_2}(x)\downarrow$. (Since $CLASH12 \cup CLASH1T3 \cup CLASH2T3$ is finite, we can check in the limit if $\varphi_{P_1}(x), \varphi_{P_2}(x)$ is defined or not. If it is undefined, then we can assume that it converges to an arbitrary value.) Also, for all $x \in CLASH12 \cup CLASH1T3 \cup CLASH2T3$, we can assume that $(\forall y)[[(x,y) \in \text{content}(T_1) \Rightarrow y = \varphi_{P_1}(x)] \wedge [(x,y) \in \text{content}(T_2) \Rightarrow y = \varphi_{P_2}(x)]]$. (Otherwise we can assume that y is the value the program converges to.) Note that this assumption may cause the values of $CLASH12, CLASH1T3$ and $CLASH2T3$, as defined above, to change. It may also cause one of Cases 1, 2, 3, or 4 to hold; however, in this case, the above analysis also holds.

Case 5. card$(CLASH1T3) =$ card$(CLASH2T3) = 1$.

Then for all $x \in N - CLASH12 \cup CLASH1T3 \cup CLASH2T3$ $(\exists i \in \{1,2\})[\varphi_{P_i}(x)\downarrow = f(x)]$. A simple calculation shows that card$(CLASH12 \cup CLASH1T3 \cup CLASH2T3) \leq 3$. If $a \geq$ card$(CLASH12 \cup CLASH1T3 \cup CLASH2T3)$, then text T such that content$(T) = \{(x,y) \mid x \in N - (CLASH12 \cup CLASH1T3 \cup CLASH2T3) \wedge (\exists i \in \{1,2,3\})[\varphi_{P_i}(x) = y]\}$ is an a-incomplete text for f. Otherwise, $a = 2$ and card$(CLASH12 \cup CLASH1T3 \cup CLASH2T3) = 3$. In this case, if there exists $i \in \{1,2,3\}$ such that $(\forall x \in CLASH12 \cup CLASH1T3 \cup CLASH2T3)(\forall y)[(x,y) \notin \text{content}(T_i)]$, then each of the remaining texts are a-

incomplete for f. Otherwise, let p be a program such that

$$\varphi_p(x) = \begin{cases} U(x) & \text{if } x \in N - (CLASH12 \cup CLASH13 \cup CLASH23); \\ y & \text{if } x \in CLASH12 \cup CLASH13 \cup CLASH23 \\ & \quad \wedge (x,y) \in \text{content}(T_{UNION}), \end{cases}$$

where $U(x)$ denotes $(\varphi_{P_1} \cup \varphi_{P_2})(x)$. Then p is program for a 1-variant of f.

Case 6. $\text{card}(CLASH1T3) = 1$ and $\text{card}(CLASH2T3) = 0$. (The case where $\text{card}(CLASH2T3) = 1$ and $\text{card}(CLASH1T3) = 0$ is similar.)

Let p be a program such that:

$$\varphi_p(x) = \begin{cases} U(x) & \text{if } x \in N - CLASH12; \\ \varphi_{P_2}(x) & \text{otherwise,} \end{cases}$$

where $U(x)$ denotes $(\varphi_{P_1} \cup \varphi_{P_2})(x)$. Then p is program for a 1-variant of f. Since if T_2 is an a-incomplete text for f, then this is the case. Otherwise, the number of errors committed by p is $\leq \text{card}(CLASH12 - CLASH1T3) \leq 1$.

Case 7. $\text{card}(CLASH1T3) = \text{card}(CLASH2T3) = 0$.

In this case, for all $(x,y) \in \text{content}(T_3)$, $f(x) = y$. This is the case since if T_3 is the correct text, then it is certainly so; otherwise, for all $x \in N - CLASH12$, $\varphi_{P_1}(x) = f(x) \vee \varphi_{P_2}(x) = f(x)$. Thus if $(x,y) \in \text{content}(T_3)$, then $f(x) = y$. Thus $[\mathbf{M}(T_3)\downarrow]$ or $[\text{card}(\{x \mid (\forall y)(x,y) \notin \text{content}(T_3)\}) > a]$. Note that we can determine in the limit one of the clauses above which holds.

Case 7a. $\text{card}(\{x \mid (\forall y)(x,y) \notin \text{content}(T_3)\}) > a$.

In this case, both T_1 and T_2 are a-incomplete texts for f.

Case 7b. $\mathbf{M}(T_3)\downarrow = P_3$.

Let $CLASH13 = \{x \mid [\varphi_{P_1}(x)\downarrow \neq \varphi_{P_3}(x)\downarrow]\}$. Let $CLASH23 = \{x \mid [\varphi_{P_2}(x)\downarrow \neq \varphi_{P_3}(x)\downarrow]\}$.

Case 7b.1. $\text{card}(CLASH13) > 2$.

In this case, T_2 is an a-incomplete text for f.

Case 7b.2. $\text{card}(CLASH23) > 2$.

In this case, T_1 is an a-incomplete text for f.

For Cases 7b.3–7b.5, assume without loss of generality that for all $x \in CLASH12 \cup CLASH13 \cup CLASH23$, $\varphi_{P1}(x)\downarrow$, $\varphi_{P_2}(x)\downarrow$, and $\varphi_{P_3}(x)\downarrow$. (Since $CLASH12 \cup CLASH13 \cup CLASH23$ is finite, we can check in the limit if $\varphi_{P_1}(x), \varphi_{P_2}(x), \varphi_{P_3}(x)$ is defined or not. If it is not defined, then we can assume that it converges to an arbitrary value.) Also, for all $x \in CLASH12 \cup CLASH13 \cup CLASH23$, we can assume that $(\forall y)[[(x,y) \in \text{content}(T_1) \Rightarrow y = \varphi_{P_1}(x)] \wedge [(x,y) \in \text{content}(T_2) \Rightarrow y = \varphi_{P_2}(x)]]$. (Otherwise, we can assume that y is the value the program converges to.) Note that this assumption may cause the values of $CLASH12$, $CLASH13$, $CLASH23$, $CLASH1T3$, and $CLASH2T3$, as defined above, to change. It may also cause one of Cases 1, 2, 3, 4 , 5, 6, 7b.1, or 7b.2 to hold; however, in this case, the above analysis also holds.

Case 7b.3. $\text{card}(CLASH13) = 0$.

In this case, let p be such that $\varphi_p = \varphi_{P_1} \cup \varphi_{P_3}$. Then φ_p is a program for a 1-variant of f.

Case 7b.4. $\text{card}(CLASH23) = 0$.

In this case, let p be such that $\varphi_p = \varphi_{P_2} \cup \varphi_{P_3}$. Then φ_p is a program for a 1-variant of f.

Case 7b.5. $0 < \text{card}(CLASH13) < 3$ and $0 < \text{card}(CLASH23) < 3$.

Then $(\forall x \in N - CLASH12 \cup CLASH13 \cup CLASH23)$ $(\exists i \in \{1,2,3\})$ $[\varphi_{P_i}(x)\downarrow = f(x)]$. A simple calculation shows that $\text{card}(CLASH12 \cup CLASH13 \cup CLASH23) \leq 3$. If $a \geq \text{card}(CLASH12 \cup CLASH13 \cup CLASH23)$, then text T such that $\text{content}(T) = \{(x,y) \mid x \in N - (CLASH12 \cup CLASH13 \cup CLASH23) \wedge (\exists i \in \{1,2,3\})[\varphi_{P_i}(x) = y]\}$ is an a-incomplete text for f. Otherwise, $a = 2$ and $\text{card}(CLASH12 \cup CLASH13 \cup CLASH23) = 3$. In this case, if there exists $i \in \{1,2,3\}$ such that $(\forall x \in CLASH12 \cup CLASH13 \cup CLASH23)(\forall y)[(x,y) \notin \text{content}(T_i)]$, then each of the remaining texts are a-incomplete texts for f. Otherwise, let p be a program such that

$$\varphi_p(x) = \begin{cases} U(x) & \text{if } x \in N - (CLASH12 \cup CLASH13 \cup CLASH23); \\ y & \text{if } x \in CLASH12 \cup CLASH13 \cup CLASH23 \\ & \wedge\ (x,y) \in \text{content}(T_{UNION}), \end{cases}$$

where $U(x)$ denotes $(\varphi_{P_1} \cup \varphi_{P_2} \cup \varphi_{P_3})(x)$. Then p is a program for a 1-variant of f. □

5.2.3. Imperfect texts. The study of imperfect texts turns out to be very complex. We only present a simulation result which says that for $b, c \in N$, the collection of functions for which a b-error program can be identified from a single c-imperfect text can also be learned from three texts, at least two of which are c-imperfect, provided we are prepared to tolerate up to twice the number of errors in the final program. A proof of this result can be worked out along similar lines to our proof of Theorem 5.9; we omit the details.

THEOREM 5.22. *Let $b, c \in N$.* $\mathbf{Im}^c\mathbf{Ex}^b \subseteq \mathbf{Mul}_3^2\mathbf{Im}^c\mathbf{Ex}^{2b}$.

5.3. Hierarchy results. We now turn our attention to hierarchy results for each of the inaccuracy types. In particular, we show for each inaccuracy type that larger collections of functions become identifiable if we

- increase the bound on the number of errors allowed in the final program (keeping other parameters the same) or
- decrease the bound on the number of inaccuracies allowed in the input texts (keeping other parameters the same).

The following two theorems, which follow from results about identification from single inaccurate texts [11], yield as corollaries the above hierarchies.

THEOREM 5.23. *Let $b, j, k \in N$, $k \geq j > \lfloor \frac{k}{2} \rfloor$.*

(a) $\mathbf{Mul}_k^j\mathbf{Im}^*\mathbf{Ex}^{b+1} - \mathbf{Ex}^b \neq \emptyset$.

(b) $\mathbf{Mul}_k^j\mathbf{Im}^*\mathbf{Ex}^* - \bigcup_{b \in N} \mathbf{Ex}^b \neq \emptyset$.

Proof. The proof follows from Theorem 1 in [11] and Theorem 5.1. □

THEOREM 5.24. $(\forall b, j, k \in N \mid k \geq j > \lfloor \frac{k}{2} \rfloor)$ $[\mathbf{Mul}_k^j\mathbf{Im}^b\mathbf{Ex} - [\mathbf{N}^{b+1}\mathbf{Ex}^* \cup \mathbf{In}^{b+1}\mathbf{Ex}^*] \neq \emptyset]$.

Proof. The proof follows from Theorem 6 in [11] and Theorem 5.2. □

It can be shown that the above two results yield the following two corollaries.

COROLLARY 5.25. *Let $a \in N \cup \{*\}$. Let $j, k \in N$ such that $k \geq j > \lfloor \frac{k}{2} \rfloor$. Then*

1. $\mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^0 \subset \mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^1 \subset \cdots \subset \mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^*$;
2. $\mathbf{Mul}_k^j\mathbf{In}^a\mathbf{Ex}^0 \subset \mathbf{Mul}_k^j\mathbf{In}^a\mathbf{Ex}^1 \subset \cdots \subset \mathbf{Mul}_k^j\mathbf{In}^a\mathbf{Ex}^*$;
3. $\mathbf{Mul}_k^j\mathbf{Im}^a\mathbf{Ex}^0 \subset \mathbf{Mul}_k^j\mathbf{Im}^a\mathbf{Ex}^1 \subset \cdots \subset \mathbf{Mul}_k^j\mathbf{Im}^a\mathbf{Ex}^*$.

COROLLARY 5.26. *Let $a \in N \cup \{*\}$. Let $j, k \in N$ such that $k \geq j > \lfloor \frac{k}{2} \rfloor$. Then*

1. $\mathbf{Mul}_k^j\mathbf{N}^0\mathbf{Ex}^a \supset \mathbf{Mul}_k^j\mathbf{N}^1\mathbf{Ex}^a \supset \cdots \supset \mathbf{Mul}_k^j\mathbf{N}^*\mathbf{Ex}^a$;

2. $\mathbf{Mul}_k^j\mathbf{In}^0\mathbf{Ex}^a \supset \mathbf{Mul}_k^j\mathbf{In}^1\mathbf{Ex}^a \supset \cdots \supset \mathbf{Mul}_k^j\mathbf{In}^*\mathbf{Ex}^a$;
3. $\mathbf{Mul}_k^j\mathbf{Im}^0\mathbf{Ex}^a \supset \mathbf{Mul}_k^j\mathbf{Im}^1\mathbf{Ex}^a \supset \cdots \supset \mathbf{Mul}_k^j\mathbf{Im}^*\mathbf{Ex}^a$.

5.4. Comparison of different types of inaccuracies. Finally, we present results which shed light on how learning from one kind of inaccuracy compares with learning from another kind of inaccuracy.

First, we show a very interesting result which implies that in the context of identification from multiple inaccurate texts, noisy texts are better than incomplete texts. This parallels previous findings about identification from single inaccurate texts. The result below shows that the collections of functions that can be identified from multiple incomplete texts can also be identified from multiple noisy texts, provided the bound on the number of errors allowed in the final program and the bound on the number of inaccuracies are the same in both cases (and also provided the cardinality of multiple texts and of acceptable texts is the same in both cases).

THEOREM 5.27. *Let* $a, b \in N \cup \{*\}$ *and* $j, k \in N$, $k \geq j > \lfloor \frac{k}{2} \rfloor$. *Then* $\mathbf{Mul}_k^j\mathbf{In}^a\mathbf{Ex}^b \subseteq \mathbf{Mul}_k^j\mathbf{N}^a\mathbf{Ex}^b$.

Proof. Let $T_1, T_2, \ldots, T_k$ be the k input texts for f such that at least j of the texts are a-noisy for f. Note that if $f(x) = y$, then at least j of the k texts contain (x, y). Let T_i' be the text formed from T_i such that $\text{content}(T_i') = \{(x, y) \mid \text{card}(\{l \mid (x, y) \in \text{content}(T_l)\}) \geq j \ \wedge \ \text{card}(\{z \mid (x, z) \in \text{content}(T_i)\}) \leq 1 \ \wedge \ (x, y) \in \text{content}(T_i)\}$. Thus we can easily obtain T_i''s from T_i's in the limit. Also, if T_i was an a-noisy text for f, then T_i' is an a-incomplete text for f. The theorem follows. □

As a contrast to the above result, the next result implies that the collections of functions which can be identified from multiple noisy texts cannot always be identified from a single incomplete text (and hence from multiple incomplete texts). The result shows that there are collections of functions for which a 0-error program can be identified from any k texts, at least $j(> \lfloor \frac{k}{2} \rfloor)$ of which are only $*$-noisy, but for which even a $*$-error program cannot be identified from single 1-incomplete text.

THEOREM 5.28. *Let* $j, k \in N$, $k \geq j > \lfloor \frac{k}{2} \rfloor$. *Then* $\mathbf{Mul}_k^j\mathbf{N}^*\mathbf{Ex} - \mathbf{In}^1\mathbf{Ex}^* \neq \emptyset$.

Proof. The proof follows from Theorem 10 in [11] and Theorem 5.1. □

The next two results compare identification from multiple incomplete texts and multiple imperfect texts. As expected, imperfect texts turn out to be worse for identification than incomplete texts.

The next theorem shows that for $i \in N$, there are collections of functions for which a 0-error program (best program) can be identified from any k texts, at least $j(> \lfloor \frac{k}{2} \rfloor)$ of which are $(3i - 1)$-incomplete, but for which even a $*$-error program (worst acceptable program) cannot be identified from any single text which is only $2i$-imperfect (a smaller bound on inaccuracies).

THEOREM 5.29. *Let* $j, k \in N, k \geq j > \lfloor \frac{k}{2} \rfloor$. *Then* $(\forall i \in N)[\mathbf{Mul}_k^j\mathbf{In}^{3i-1}\mathbf{Ex} - \mathbf{Im}^{2i}\mathbf{Ex}^* \neq \emptyset]$.

Proof. The proof follows from Theorem 58 in [14] and Theorem 5.2. □

Our final theorem is along similar lines to the previous theorem and shows the detrimental effects of imperfect text over incomplete texts. The result says that there are collections of functions for which a 0-error program (best possible) can be identified from any k texts, at least $j(> \lfloor \frac{k}{2} \rfloor)$ of which are $*$-incomplete (worst kind of missing data), but for which even a $*$-error program (worst acceptable program) cannot be identified from any single text which is only $*$-imperfect (least harmless imperfection).

THEOREM 5.30. *Let* $j, k \in N$, $k \geq j > \lfloor \frac{k}{2} \rfloor$. *Then* $\mathbf{Mul}_k^j\mathbf{In}^*\mathbf{Ex} - \mathbf{Im}^*\mathbf{Ex}^* \neq \emptyset$.

Proof. The proof follows from Theorem 60 in [14] and Theorem 5.2. □

6. Conclusion. We presented arguments against the idealized assumption in Gold's paradigm that a learning agent receives data from a single and accurate source. Gold's paradigm was suitably extended to account for the possibility that a learning agent may receive data from multiple sources, some of which may be inaccurate. Results were presented for the learning task of scientific inquiry modeled as identification in the limit of computer programs for computable functions from their graphs.

For each kind of inaccuracy, we established sufficient conditions when, for the same bound on the number of inaccuracies in both the multiple inaccurate texts and the single inaccurate text, and for the same bound on the number of errors allowed in the final hypothesis for the multiple and single cases, identification from multiple sources is not a restriction. We provided significant partial results for the difficult problem of determining when identification from multiple inaccurate texts is a restriction over identification from a single inaccurate text. We also demonstrated cases under which the detrimental effects of multiple inaccurate texts can be overcome by either allowing more errors in the final program or by decreasing the bound on the number of inaccuracies. We also established the usual hierarchies for each kind of inaccuracy in which

- keeping all of the other parameters fixed and increasing the bound on the number of errors allowed in the final program improves learnability;
- keeping all of the other parameters fixed and increasing the bound on the number of inaccuracies in the acceptable texts restricts learnability.

Finally, we were able to demonstrate that learning from noisy texts is preferable to learning from incomplete texts, which in turn is preferable to learning from imperfect texts.

It should be noted that several of the results presented in section 5.2 can easily be generalized to the case in which a machine is fed k texts and for at least j of these texts the inaccuracies are acceptable. Also, the results in this are about a simple criterion of success, viz, **Ex**-identification. We can also prove corresponding results for a more general criterion of learning, viz, **Bc**-identification (see [8] for a definition; this is also known as GN^{∞} [2] in the Russian literature). Additionally, we would like to note that similar results can also be shown to hold for another learning task, viz, first-language acquisition modeled as identification in the limit of grammars for recursively enumerable languages from a text of these languages. The criterion of success for language acquisition is known as **TxtEx**-identification (see [7] for a definition).

Finally, we end this paper on a speculatory note pointing to future research directions. Scientific success is often not limited to the success of a single scientist receiving data from multiple, possibly inaccurate, sources. In actual practice, a number of scientists are simultaneously investigating a phenomenon, each receiving data from multiple, possibly inaccurate, sources. Scientific success is achieved if any one of these scientists is successful. This scenario could be modeled as a "team" of learning machines, each member of the team receiving multiple inaccurate texts. The team is successful if at least one member of the team converges to a correct program for the function being learned (see Smith [24] for discussion of team identification from a *single* and *accurate* text). However, given our experience with the combinatorial difficulty of the subject matter of this paper, it is quite likely that a study incorporating teams and multiple inaccurate environments may turn out to be very complex, and it is not clear at this stage what additional insights such an investigation may provide.

Acknowledgments. We would like to thank John Case, Mark Fulk, and Errol Lloyd for encouragement and advice. Helpful discussions were provided by Sudhir Jha, Lata Narayanan, and Rajeev Raman. We would also like to thank Andrea Lobo for careful reading of an earlier draft of the paper.

During the initial stages of this investigation, Ganesh Baliga was in the Department of Computer and Information Sciences at the University of Delaware; Sanjay Jain was in the Department of Computer and Information Sciences at the University of Delaware and at the Institute of Systems Science at the National University of Singapore; and Arun Sharma was in the Department of Brain and Cognitive Sciences at the Massachusetts Institute of Technology.

We would also like to express our gratitude to Prof. S. N. Maheshwari of the Department of Computer Science and Engineering at IIT-Delhi for making the facilities of his department available to us during the preparation of an early draft of this paper.

Finally, we would like to thank the referees for valuable comments.

REFERENCES

[1] *Proc. 5th Annual ACM Workshop on Computational Learning Theory*, ACM, New York, 1992.
[2] J. M. Barzdin, *Two theorems on the limiting synthesis of functions*, in Theory of Algorithms and Programs 210, Latvian State University, Riga, Latvia, 1974, pp. 82–88 (in Russian).
[3] L. Blum and M. Blum, *Toward a mathematical theory of inductive inference*, Inform. and Control, 28 (1975), pp. 125–155.
[4] M. Blum, *A machine independent theory of the complexity of recursive functions*, J. Assoc. Comput. Mach., 14 (1967), pp 322–336.
[5] J. Case, *Periodicity in generations of automata*, Math. Systems Theory, 8 (1974), pp. 15–32.
[6] J. Case, *Infinitary self-reference in learning theory*, J. Exper. Theoret. Artif. Intell., 4 (1993), pp 281–293; also appears in special issue on Analogical and Inductive Inference, 6 (1994), pp. 3–17.
[7] J. Case and C. Lynes, *Machine inductive inference and language identification*, in Proc. 9th International Colloquium on Automata, Languages and Programming, M. Nielsen and E. M. Schmidt, eds., Lecture Notes in Computer Science 140, Springer-Verlag, Berlin, 1982, pp. 107–115.
[8] J. Case and C. Smith, *Comparison of identification criteria for machine inductive inference*, Theoret. Comput. Sci., 25 (1983), pp 193–220.
[9] M. Fulk, *A study of inductive inference machines*, Ph.D. thesis, State University of New York at Buffalo, Buffalo, New York, 1985.
[10] M. Fulk and J.Case, eds., *Proc. 3rd Annual Workshop on Computational Learning Theory*, Morgan Kaufmann, San Mateo, CA, 1990.
[11] M. A. Fulk and S. Jain, *Learning in the presence of inaccurate information*, in Proc. 2nd Annual Workshop on Computational Learning Theory, R. Rivest, D. Haussler, and M. K. Warmuth, eds., Morgan Kaufmann, San Mateo, CA, 1989, pp. 175–188.
[12] E. M. Gold, *Language identification in the limit*, Inform. and Control, 10 (1967), pp. 447–474.
[13] D. Haussler and L. Pitt, eds., *Proc.* 1988 *Workshop on Computational Learning Theory*, Morgan Kaufmann, San Mateo, CA, 1988.
[14] S. Jain, *Learning in the presence of additional information and inaccurate information*, Ph.D. thesis, University of Rochester, Rochester, NY, 1990.
[15] S. Jain, *Program synthesis in the presence of infinite number of inaccuracies*, in Proc. 5th International Workshop on Algorithmic Learning Theory, S. Arikawa and K. P. Jantke, eds., Reinhardsbrunn Castle, Germany, 1994, Lecture Notes in Artificial Intelligence 872, Springer-Verlag, Berlin, 1994, pp. 333–348.
[16] M. Machtey and P. Young, *An Introduction to the General Theory of Algorithms*, North–Holland, New York, 1978.
[17] D. Osherson, M. Stob, and S. Weinstein, *Systems that Learn, An Introduction to Learning Theory for Cognitive and Computer Scientists*, MIT Press, Cambridge, MA, 1986.
[18] D. Osherson and S. Weinstein, *A note on formal learning theory*, Cognition, 11 (1982), pp. 77–88.

[19] H. Putnam, *Reductionism and the nature of psychology*, Cognition, 2 (1973), pp. 131–146.
[20] R. Rivest, D. Haussler, and M. K. Warmuth, eds., *Proc. 2nd Annual Workshop on Computational Learning Theory*, Morgan Kaufmann, San Mateo, CA, 1989.
[21] H. Rogers, *Gödel numberings of partial recursive functions*, J. Symbolic Logic, 23 (1958), pp. 331–341.
[22] H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw–Hill, New York, 1967; reprint, MIT Press, Cambridge, MA, 1987.
[23] G. Schäfer-Richter, *Some results in the theory of effective program synthesis-learning by defective information*, Lecture Notes in Comput. Sci., 215 (1986), pp. 219–225.
[24] C. Smith, *The power of pluralism for automatic program synthesis*, J. Assoc. Comput. Mach., 29 (1982), pp. 1144–1165.
[25] R. J. Solomonoff, *A formal theory of inductive inference, part* I, Inform. and Control, 7 (1964), pp. 1–22.
[26] R. J. Solomonoff, *A formal theory of inductive inference, part* II, Inform. and Control, 7 (1964), pp. 224–254.
[27] L. G. Valiant and M. K. Warmuth, eds., *Proc. 4th Annual Workshop on Computational Learning Theory*, Morgan Kaufmann, San Mateo, CA, 1991.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 991–1005, August 1997

005

SINGULAR AND PLURAL NONDETERMINISTIC PARAMETERS*

MICHAL WALICKI† AND SIGURD MELDAL†

Abstract. The article defines algebraic semantics of singular (call-time-choice) and plural (run-time-choice) nondeterministic parameter passing and presents a specification language in which operations with both kinds of parameters can be defined simultaneously. Sound and complete calculi for both semantics are introduced. We study the relations between the two semantics and point out that axioms for operations with plural arguments may be considered as axiom schemata for operations with singular arguments.

Key words. algebraic specification, many-sorted algebra, nondeterminism, sequent calculus

AMS subject classifications. 68Q65, 68Q60, 68Q10, 68Q55, 03B60, 08A70

PII. S00975397264317

1. Introduction.

The notion of *nondeterminism* arises naturally in describing concurrent systems. Various approaches to the theory and specification of such systems, for instance, CCS [16], CSP [9], process algebras [1], and event structures [26], include the phenomenon of nondeterminism. But nondeterminism is also a natural concept in describing sequential programs, either as a means of indicating a "don't care" attitude as to which among a number of computational paths will actually be utilized in a particular computation (e.g., [3]) or as a means of increasing the level of abstraction [14, 25]. The present work proceeds from the theory of *algebraic specifications* [4, 27] and generalizes the theory so that it can be applied to describing nondeterministic operations.

In deterministic programming the distinction between call-by-value and call-by-name semantics of parameter passing is well known. The former corresponds to the situation where the actual parameters to function calls are evaluated and passed as values. The latter allows parameters which are function expressions, passed by a kind of Algol copy rule [21], and which are evaluated whenever a need for their value arises. Thus call-by-name will terminate in many cases when the value of a function may be determined without looking at (some of) the actual parameters, i.e., even if these parameters are undefined. Call-by-value will, in such cases, always lead to undefined result of the call. Nevertheless, the call-by-value semantics is usually preferred in the actual programming languages since it leads to clearer and more tractable programs.

Following [20], we call the nondeterministic counterparts of these two notions *singular* (call-by-value) and *plural* (call-by-name) parameter passing. Other names applied to this, or closely related distinction, are *call-time-choice* vs. *run-time-choice* [2, 8] or *inside-out* (IO) vs. *outside-in* (OI), which reflect the substitution order corresponding to the respective semantics [5, 6]. In the context where one allows nondeterministic parameters, the difference between the two semantics becomes quite apparent even without looking at their termination properties. Let us suppose that

*Received by the editors March 9, 1994; accepted for publication (in revised form) August 7, 1995. The research of the first author was partially supported by the Architectural Abstraction project under NFR (Norway), the CEC under ESPRIT-II Basic Research Working Group 6112 COMPASS, DARPA under ONR contracts N00014-92-J-1928 and N00014-93-1-1335, and Air Force Office of Scientific Research grant AFOSR-91-0354.

http://www.siam.org/journals/sicomp/26-4/26431.html

†Department of Informatics, University of Bergen, HiB, 5020 Bergen, Norway (michal.walicki@ii.uib.no, sigurd.meldal@ii.uib.no).

we have defined operation $g(x)$ as "**if** $x = 0$ **then** a **else** (**if** $x = 0$ **then** b **else** c)" and that we have a nondeterministic choice operation $\sqcup$ returning an arbitrary element from the argument set. The singular interpretation will satisfy the formula $\boldsymbol{\phi}$: $g(x) = (\textbf{if } x = 0 \textbf{ then } a \textbf{ else } c)$, whereas the plural interpretation need not satisfy this formula. For instance, under the singular interpretation $g(\sqcup.\{0,1\})$ will yield either a or c, whereas the set of possible results of $g(\sqcup.\{0,1\})$ under the plural interpretation will be $\{a,b,c\}$. (Notice that in a deterministic environment both semantics would yield the same results.) The fact that the difference between the two semantics occurs already in very trivial examples of terminating nondeterministic operations motivates our investigation.

We discuss the distinction between the singular and the plural passing of nondeterministic parameters in the context of algebraic semantics, focusing on the associated reasoning systems. The singular semantics is given by *multialgebras*, that is, algebras where functions are set valued and where these values correspond to the sets of possible results returned by nondeterministic operations. Thus, if f is a nondeterministic operation, $f(t)$ will denote the set of possible results returned by f when applied to t. We introduce the calculus NEQ which is sound and complete with respect to this semantics.

Although terms may denote sets, the variables in the language range only over individuals. This is motivated by the interest in describing unique results returned by each particular application of an operation (execution of the program). It gives us the possibility of writing instead of a formula $\boldsymbol{\Phi}(f(t))$, which expresses something about the whole set of possible results of $f(t)$, the formula corresponding to $x \in f(t) \Rightarrow \boldsymbol{\Phi}(x)$, which express something about each particular result x returned by $f(t)$. Unfortunately, this poses the main problem of reasoning in the context of nondeterminism—the lack of general substitutivity. From the fact that $h(x)$ is deterministic (for each x has a unique value) we cannot conclude that so is $h(t)$ for an arbitrary term t. If t is nondeterministic, $h(t)$ may have several possible results. The calculus NEQ is designed so that it appropriately restricts the substitution of terms for singular variables.

Although operations in multialgebras are set valued, their carriers are usual sets. Thus operations map individuals to sets. This is not sufficient to model plural arguments. Such arguments can be understood as sets being passed to the operation. The fact that, under plural interpretation, $g(x)$ as defined above need not satisfy $\boldsymbol{\phi}$ results from the two occurrences of x in the body of g. Each of these occurrences corresponds to a repeated application of choice from the argument set x, that is, potentially, to a different value. In order to model such operations we take as the carrier of the algebra a (subset of the) power set—operations map sets to sets. In this way we obtain *power algebra* semantics. The extension of the semantics is reflected at the syntactic level by introduction of *plural variables* ranging over sets rather than over individuals. The sound and complete extension of NEQ is obtained by adding one new rule which allows for usual substitution of arbitrary terms for plural variables.

The structure of the paper is as follows. In sections 2 and 3 we introduce the language for specifying nondeterministic operations and explain the intuition behind its main features. In section 4 we define *multialgebraic* semantics for singular specifications and introduce a sound and complete calculus for such specifications. In section 5 the semantics is generalized to *power algebras* capable of modeling plural parameters, and the sound and complete extension of the calculus is obtained by introducing one additional rule. A comparison of both semantics in section 6 is guided

by the similarity of the respective calculi. We identify the subclasses of multimodels and power models which may serve as equivalent semantics of one specification. We also highlight the increased complexity of the power algebra semantics reflecting the problems with intuitive understanding of plural arguments.

Proofs of the theorems are merely indicated in this presentation. It reports some of the results from [24] where the full proofs and other details can be found.

2. The specification language. A specification is a pair (Σ, Π), where the signature Σ is a pair (**S**,**F**) of sorts **S** and operation symbols **F** (with argument and result sorts in **S**). The set of terms over a signature Σ and variable set X is denoted by $W_{\Sigma,X}$. We always assume that, for every sort S, the set of ground words of sort S, S^{W_Σ}, is not empty.[1]

Π is a set of sequents of atomic formulas written as $a_1, \ldots, a_n \mapsto e_1, \ldots, e_m$. The left-hand side (LHS) of $\mapsto$ is called the *antecedent* and the right-hand side (RHS) the *consequent*, and both are to be understood as sets of atomic formulas (i.e., the ordering and multiplicity of the atomic formulas do not matter). In general, we allow either antecedent or consequent to be empty, though $\emptyset$ is usually dropped in the notation. A sequent with exactly one formula in the consequent ($m = 1$) is called a *Horn formula*, and a Horn formula with empty antecedent ($n = 0$) is a *simple formula* (or a *simple sequent*).

All variables occurring in a sequent are implicitly universally quantified over the whole sequent. A sequent is satisfied if, for every assignment to the variables, one of the antecedents is false or one of the consequents is true (it is valid iff the formula $a_1 \wedge \cdots \wedge a_n \Rightarrow e_1 \vee \cdots \vee e_m$ is valid).

For any term (formula set of formulas) ξ, $\mathcal{V}[\xi]$ will denote the set of variables in ξ. If the variable set is not mentioned explicitly, we may also write $x \in \mathcal{V}$ to indicate that x is a variable.

An atomic formula in the consequent is either an *equation*, $t = s$, or an *inclusion*, $t \prec s$, of terms t, $s \in W_{\Sigma,X}$. An atomic formula in the antecedent, written $t \frown s$, will be interpreted as nonempty intersection of the (result) sets corresponding to t and s. For a given specification $\mathrm{SP} = (\Sigma, \Pi)$, $\mathcal{L}(\mathrm{SP})$ will denote the above language over the signature Σ.

The above conventions will be used throughout the paper. The distinction between the singular and the plural parameters (introduced in the section 5) will be reflected in the notation by the superscript *: a plural variable will be denoted by x^*, the set of plural variables in a term t by $\mathcal{V}^*[t]$, a specification with plural arguments SP^*, the corresponding extension of the language $\mathcal{L}$ by $\mathcal{L}^*$, etc.

3. A note on the intuitive interpretation. Multialgebraic semantics [10, 13] interprets specifications in some form of power structures where the (nondeterministic) operations correspond to *set-valued functions*. This means that a (ground) term is interpreted as a set of *possibilities*; it denotes the set of *possible* results of the corresponding operation. We, on the other hand, want our formulas to express *necessary* facts, i.e., facts which have to hold in *every evaluation* of a program (specification). This is achieved by interpreting terms as *applications* of the respective operations. Every two syntactic occurrences of a term t will refer to *possibly distinct* applications of t. For nondeterministic terms this means that they may denote two distinct values.

[1] This restriction is motivated by the fact (pointed out in [7]) that admitting empty carriers requires additional mechanisms (explicit quantification) in order to obtain sound logic. We conjecture that a similar solution can be applied in our case.

Typically, equality is interpreted in a multialgebra as *set equality* [13, 23, 12]. For instance, the formula $\mapsto t = s$ means that *the sets corresponding to all possible results* of the operations t and s are equal. This gives a model which is mathematically plausible but which does not correspond to our operational intuition. The (set) equality $\mapsto t = s$ does not guarantee that the result returned by some particular application of t will actually be equal to the result returned by an application of s. It merely tells us that *in principle* (in all possible executions) any result produced by t can also be produced by s and vice versa.

Equality in our view should be a *necessary* equality which must hold in every evaluation of a program (specification). *It does not correspond to set equality but to identity of one-element sets.* Thus the simple formula $\mapsto t = s$ will hold in a multistructure M iff both t and s are interpreted in M as one and the same set which, in addition, *has only one element.* Equality is then a *partial equivalence relation*, and terms t for which $\mapsto t = t$ holds are exactly the deterministic terms, denoted by $D_{\boldsymbol{SP},\mathrm{X}}$. This last equality indicates that arbitrary two applications of t have to return the same result.

If it is possible to produce a computation where t and s return different results—and this is possible when they are nondeterministic—then the terms are not equal but, at best, *equivalent.* They are equivalent if they are capable of returning the same results, i.e., if they are interpreted as the same set. This may be expressed using the inclusion relation: $s \prec t$ holds iff the set of possible results of s is included in the set of possible results of t, and $s \asymp t$ if each is included in the other.

Having introduced inclusion one might expect that a nondeterministic operation can be specified by a series of inclusions, each defining one of its possible results. However, such a specification gives only a "lower bound" on the admitted nondeterminism. Consider the following example.

Example 3.1.

S: {Nat},

F:

0:	$\to$ Nat	(zero)
$s\_$:	Nat $\to$ Nat	(successor)
$\_\sqcup\_$:	Nat $\times$ Nat $\to$ Nat	(binary nondeterministic choice)

Π:

(1) $\mapsto 0 = 0$		
(2) $\mapsto s(x) = s(x)$		
(3) $1 \frown 0 \mapsto$		(As usual), we abbreviate $s^n(0)$ as n.)
(4) $\mapsto 0 \prec 0 \sqcup 1$	$\mapsto 1 \prec 0 \sqcup 1$	

The first two axioms make zero and successor deterministic. A limited form of negation is present in $\mathcal{L}$ in the form of sequents with empty consequent. Axiom (3) makes 0 distinct from 1. Axioms (4) make then $\sqcup$ a nondeterministic choice with 0 and 1 among its possible results. This, however, ensures only that in every model both 0 and 1 can be returned by $0 \sqcup 1$. In most models all other kinds of elements may be among its possible results as well, since no extension of the result set of $0 \sqcup 1$ will violate the inclusions of (4). If we are satisfied with this degree of precision, we may stop here and use only the Horn formula. All the results in the rest of the paper apply to this special case. But to specify an "upper bound" of nondeterministic operations we need *disjunction*, the multiple formulas in the consequents. Now, if we write the axiom

$$(5) \quad \mapsto 0 \sqcup 1 = 0, \quad 0 \sqcup 1 = 1,$$

the two occurrences of $0 \sqcup 1$ refer to two arbitrary applications and, consequently, we obtain that either any application of $0 \sqcup 1$ equals 0 or else it equals 1, i.e., that $\sqcup$ is not really nondeterministic but merely underspecified. Since axioms (4) require that both 0 and 1 be among the results of t, the addition of (5) will actually make the specification inconsistent.

What we are trying to say with the disjunction of (5) is that *every application* of $0 \sqcup 1$ returns either 0 or 1; i.e., we need a means of identifying two occurrences of a nondeterministic term as referring to one and the same application. This can be done by *binding* both occurrences to a variable. The appropriate axiom will be

$$(5') \quad x \frown 0 \sqcup 1 \mapsto x = 0,\ x = 1.$$

The axiom says: whenever $0 \sqcup 1$ returns x, then x equals 0 or x equals 1. Notice that such an interpretation presupposes that the variable x refers to a *unique*, *individual* value. Thus bindings have the intended function only if they involve *singular* variables. (Plural variables, on the other hand, will refer to sets and not individuals, and so the axiom

$$(5'') \quad x^* \frown 0 \sqcup 1 \mapsto x^* = 0,\ x^* = 1$$

would have a completely different meaning.) The singular semantics is the most common in the literature on algebraic semantics of nondeterministic specification languages [2, 8, 11], in spite of the fact that it prohibits unrestricted substitution of terms for variables. Any substitution must now be guarded by the check that the substituted term yields a unique value, i.e., is deterministic. We return to this point in the subsection on reasoning, where we introduce a calculus which does not allow one, for instance, to conclude $0 \sqcup 1 = 0 \sqcup 1 \mapsto 0 \sqcup 1 = 0,\ 0 \sqcup 1 = 1$ from the axiom $(5')$ (though it could be obtained from $(5'')$).

4. The singular case: Semantics and calculus. This section defines the multialgebraic semantics of specifications with singular arguments and introduces a sound and complete calculus.

4.1. Multistructures and multimodels.

DEFINITION 4.2 (Multistructures). *Let* Σ *be a signature.* M *is a* Σ-multistructure *if*

(1) *its carrier* $|\mathrm{M}|$ *is an* $\mathbf{S}$*-sorted set,*
(2) *for every* $f\colon \mathrm{S}_1 \times \cdots \times \mathrm{S}_\mathrm{n} \to \mathrm{S}$ *in* $\mathbf{F}$*, there is a corresponding function* $f^M\colon \mathrm{S}_1^\mathrm{M} \times \cdots \times \mathrm{S}_\mathrm{n}^\mathrm{M} \to \mathcal{P}^+(\mathrm{S}^\mathrm{M})$.

A function $\mathbf{\Phi}\colon \mathrm{A} \to \mathrm{B}$ *(i.e., a family of functions* $\mathbf{\Phi}_\mathrm{S}\colon \mathrm{S}^\mathrm{A} \to \mathrm{S}^\mathrm{B}$ *for every* $\mathrm{S} \in \mathbf{S}$*) is a* multihomomorphism *from a* Σ*-multistructure* A *to* B *if*

(H1) *for each constant symbol* $c \in \mathbf{F}$, $\mathbf{\Phi}(c^A) \subseteq c^B$,
(H2) *for every* $f\colon \mathrm{S}_1 \times \cdots \times \mathrm{S}_\mathrm{n} \to \mathrm{S}$ *in* $\mathbf{F}$ *and* $\underline{a}_1 \cdots \underline{a}_n \in \mathrm{S}_1^\mathrm{A} \times \cdots \times \mathrm{S}_\mathrm{n}^\mathrm{A}$: $\mathbf{\Phi}(f^\mathrm{A}(\underline{a}_1 \cdots \underline{a}_n)) \subseteq f^\mathrm{B}(\mathbf{\Phi})(\underline{a}_i) \cdots \mathbf{\Phi}(\underline{a}_n))$.

If all inclusions in H1 *and* H2 *are (set) equalities the homomorphism is* tight*; otherwise it is strictly loose (or just* loose*).*

$\mathcal{P}^+(S)$ denotes the set of nonempty subsets of the set S. Operations applied to sets refer to their unique pointwise extensions. Notice that for a constant $c\colon \to S(2)$ indicates that c^M can be a *set* of several elements of sort S.

Since multihomomorphisms are defined on individuals and not sets they preserve singletons and are $\subseteq$-monotonic. We denote the class of Σ-multistructures by

$\mathrm{MStr}(\Sigma)$. It has the distinguished word structure MW_Σ defined in the obvious way, where each ground term is interpreted as a singleton set. We will treat such singleton sets as terms rather than one-element sets (i.e., we do not take special pains to distinguish MW_Σ and W_Σ). MW_Σ is not an initial Σ-structure since it is deterministic and there can exist several homomorphisms from it to a given multistructure. We do not focus on the aspect of initiality and merely register the useful fact from [11].

LEMMA 4.3. *M is a Σ-multistructure iff for every set of variables X and assignment β: $X \to |\mathrm{M}|$, there exists a unique function $\boldsymbol{\beta}[\_]$: $\mathrm{W}_{\Sigma,\mathrm{X}} \to \mathcal{P}^+(|\mathrm{M}|)$ such that*

(1) $\boldsymbol{\beta}[x] = \{\boldsymbol{\beta}(x)\}$,
(2) $\boldsymbol{\beta}[c] = c^M$,
(3) $\boldsymbol{\beta}[f(t_i)] = \cup\{f^M(y_i) \mid y_i \in \boldsymbol{\beta}[t_i]\}$.

In particular, for $X = \emptyset$ there is a unique interpretation function (not a multihomomorphism) $\mathcal{I}$: $W_\Sigma \to \mathcal{P}^+(|M|)$ satisfying the last two points of this definition.

As a consequence of the definition of multistructures, all operations are $\subseteq$-monotonic, i.e., $\boldsymbol{\beta}[s] \subseteq \boldsymbol{\beta}[t] \Rightarrow \boldsymbol{\beta}[f(s)] \subseteq \boldsymbol{\beta}[f(t)]$. Notice also that assignment in the lemma (and in general whenever it is an assignment of elements from a multistructure) means assignment of individuals, not sets.

Next we define the class of *multimodels* of a specification.

DEFINITION 4.4 (Satisfiability). *A Σ-multistructure* M satisfies *an $\mathcal{L}(\Sigma)$ sequent* π

$$t_i \frown s_i \mapsto p_j = r_j, m_k \prec n_k,$$

written $\mathrm{M} \models \pi$ *iff for every β*: $\mathrm{X} \to \mathrm{M}$ *we have*

$$\bigwedge_i \boldsymbol{\beta}[t_i] \cap \boldsymbol{\beta}[s_i] \neq \emptyset \Rightarrow \bigvee_j \boldsymbol{\beta}[p_j] \equiv \boldsymbol{\beta}[r_j] \vee \bigvee_k \boldsymbol{\beta}[m_k] \subseteq \boldsymbol{\beta}[n_k],$$

where $\mathrm{A} \equiv \mathrm{B}$ *iff* A *and* B *are the same one-element set.*

An SP-multimodel *is a Σ-multistructure which satisfies all the axioms of SP. We denote the class of multimodels of SP by MMod(SP).*

The reason for using nonempty intersection (and not set equality) as the interpretation of $\frown$ in the antecedents is the same as using "elementwise" equality $\equiv$ in the consequents. Since we avoid set equality in the positive sense (in the consequents), the most natural negative form seems to be the one we have chosen. For deterministic terms this is the same as equality, i.e., deterministic antecedents correspond exactly to the usual (deterministic) conditions. For nondeterministic terms this reflects our interest in binding such terms: the sequent "$\ldots s \frown t \ldots \mapsto \ldots$" is equivalent to "$\ldots x \frown s,\ x \frown t \ldots \mapsto \ldots$." A binding "$\ldots x \frown t \ldots \mapsto \ldots$" is also equivalent to the more familiar "$\ldots x \in t \ldots \mapsto \ldots$," so the notation $s \frown t$ may be read as an abbreviation for the more elaborate formula with two $\in$ and a new variable x not occurring in the rest of the sequent.

For a justification of this, as well as other choices we have made here, the reader is referred to [24].

4.2. The calculus for singular semantics. In [24] we introduced the calculus NEQ which is sound and complete with respect to the class MMod(SP). Its rules are as follows:

(R1) $\quad \mapsto x = x, \qquad x \in \mathcal{V},$

(R2) $$\frac{\boldsymbol{\Gamma}^x_{t_2} \mapsto \boldsymbol{\Delta}^x_{t_2} \quad ; \quad \boldsymbol{\Gamma}' \mapsto t_1 = t_2, \boldsymbol{\Delta}'}{\boldsymbol{\Gamma}^x_{t_1}, \boldsymbol{\Gamma}' \mapsto \boldsymbol{\Delta}^x_{t_1}, \boldsymbol{\Delta}'},$$

(R3) $$\frac{\boldsymbol{\Gamma}^x_{t_2} \mapsto \boldsymbol{\Delta}^x_{t_2} \quad ; \quad \boldsymbol{\Gamma}' \mapsto t_1 \prec t_2, \boldsymbol{\Delta}'}{\boldsymbol{\Gamma}^x_{t_1}, \boldsymbol{\Gamma}' \mapsto \boldsymbol{\Delta}^x_{t_1}, \boldsymbol{\Delta}'}, \quad x \text{ not in a RHS of } \prec,$$

(R4) (a) $x \frown y \mapsto x = y$, (b) $x \frown t \mapsto x \prec t$, $x, y \in \mathcal{V}$,

(R5) $$\frac{\boldsymbol{\Gamma} \mapsto \boldsymbol{\Delta}, s \preceq t \quad ; \quad \boldsymbol{\Gamma}', s \frown t \mapsto \boldsymbol{\Delta}'}{\boldsymbol{\Gamma}, \boldsymbol{\Gamma}' \mapsto \boldsymbol{\Delta}, \boldsymbol{\Delta}'}, \quad \text{(CUT)} \quad (\preceq \text{ stands for either } = \text{ or } \prec),$$

(R6) (a) $$\frac{\boldsymbol{\Gamma} \mapsto \boldsymbol{\Delta}}{\boldsymbol{\Gamma} \mapsto \boldsymbol{\Delta}, e}, \qquad \text{(b)} \quad \frac{\boldsymbol{\Gamma} \mapsto \boldsymbol{\Delta}}{\boldsymbol{\Gamma}, e \mapsto \boldsymbol{\Delta}} \qquad \text{(WEAK)},$$

(R7) $$\frac{\boldsymbol{\Gamma}, x \frown t \mapsto \boldsymbol{\Delta}}{\boldsymbol{\Gamma}^x_t \mapsto \boldsymbol{\Delta}^x_t}, \qquad x \in \mathcal{V} - \mathcal{V}[t], \text{ at most one } x \text{ in } \boldsymbol{\Gamma} \mapsto \boldsymbol{\Delta} \qquad \text{(ELIM)}.$$

$\boldsymbol{\Gamma}^{\mathrm{a}}_{\mathrm{b}}$ denotes $\boldsymbol{\Gamma}$ with b substituted for a. Short comments on each of the rules may be in order.

The fact that "=" is a partial equivalence relation is expressed in (R1). It applies only to variables and is sound because all assignments assign individual values to the (singular) variables.

(R2) is a paramodulation rule allowing replacement of terms which may be deterministic (in the case where $t_1 = t_2$ holds in the second assumption). In particular, it allows derivation of the standard substitution rule when the substituted terms are deterministic and prevents substitution of nondeterministic terms for variables.

(R3) allows "specialization" of a sequent by substituting for a term t_2 another term t_1 which is included in t_2. The restriction that the occurrences of t_2 which are substituted for don't occur in the RHS of $\prec$ is needed to prevent, for instance, the unsound conclusion $\mapsto t_3 \prec t_1$ from the premises $\mapsto t_3 \prec t_2$ and $\mapsto t_1 \prec t_2$.

(R4) and (R5) express the relation between $\frown$ in the antecedent and the equality and inclusion in the consequent. The axiom of standard sequent calculus, $e \mapsto e$, (i.e., $s \frown t \mapsto s \preceq t$) does not hold in general here because the antecedent corresponds to nonempty intersection of the result sets while the consequent to the inclusion ($\prec$) or identity of one-element ($=$) result sets. Only for deterministic terms (in particular, variables) s, t do we have that $s \frown t \mapsto s = t$ holds.

(R5) allows us to cut both $\mapsto s = t$ and $\mapsto s \prec t$ with $s \frown t \mapsto \boldsymbol{\Delta}$.

(R7) eliminates redundant bindings, namely those that bind an application of a term occurring at most once in the rest of the sequent.

We will write $\Pi \vdash_{\mathrm{CAL}} \boldsymbol{\pi}$ to indicate that $\boldsymbol{\pi}$ is provable from $\boldsymbol{\Pi}$ with the calculus CAL.

The counterpart of soundness/completeness of the equational calculus is as follows [24].

THEOREM 4.5. *NEQ is sound and complete with respect to MMod(SP):*

$$\textit{MMod(SP)} \models \boldsymbol{\pi} \textit{ iff } \boldsymbol{\Pi} \vdash_{NEQ} \boldsymbol{\pi}.$$

Proof idea. Soundness is proved by induction on the length of the proof $\boldsymbol{\Pi} \vdash_{\mathrm{NEQ}} \boldsymbol{\pi}$. The proof of the completeness part is a standard, albeit rather involved, Henkin-style argument. The axiom set $\boldsymbol{\Pi}$ of SP is extended by adding all $\mathcal{L}$(SP) formulas $\boldsymbol{\pi}$

which are consistent with $\mathbf{\Pi}$ (and the previously added formulas). If the addition of $\boldsymbol{\pi}$ leads to inconsistency, one adds the negation of $\boldsymbol{\pi}$. Since empty consequents provide only a restricted form of negation, the general negation operation is defined as a set of formulas over the original signature extended with new constants. One shows then that the construction yields a consistent specification with a deterministic basis from which a model can be constructed.

We also register an easy lemma that the set-equivalent terms $t \asymp s$ satisfy the same formulas.

LEMMA 4.6. *$t \asymp s$ iff, for any sequent $\boldsymbol{\pi}$, $\mathbf{\Pi} \vdash_{NEQ} \boldsymbol{\pi}^z_t$ iff $\mathbf{\Pi} \vdash_{NEQ} \boldsymbol{\pi}^z_s$.* □

5. The plural case: Semantics and calculus. The singular semantics for passing nondeterminate arguments is the most common notion to be found in the literature. Nevertheless, the plural semantics has also received some attention. In the denotational tradition most approaches considered both possibilities [18, 19, 20, 22]. Engelfriet and Schmidt gave a detailed study of both—in their language, IO and OI—semantics based on tree languages [5] and continuous algebras of relations and power sets [6]. The unified algebras of Mosses [17] and the rewriting logic of Meseguer [15] represent other algebraic approaches distinguishing these aspects.

We will define the semantics for specifications where operations may have both singular and plural arguments. The next subsection gives the necessary extension of the calculus NEQ to handle this generalized situation.

5.1. Power structures and power models. Singular arguments (such as the variables in $\mathcal{L}$) have the usual algebraic property that they refer to a unique value. This reflects the fact that they are evaluated at the moment of substitution and the result is passed to the following computation. Plural arguments, on the other hand, are best understood as textual parameters. They are not passed as a single value, but every occurrence of the formal parameter denotes a distinct application of the operation.

We will allow both singular and plural parameter passing in one specification. The corresponding semantic distinction is between power set functions which are merely $\subseteq$-monotonic and those which also are $\cup$-additive.

In the language, we merely introduce a notational device for distinguishing the singular and plural arguments. We allow annotating the sorts in the profiles of the operation by a superscript, like S^*, to indicate that an argument is plural.

Furthermore, we partition the set of variables into two disjoint subsets of singular X and plural X^* variables. x and x^* are to be understood as distinct symbols. We will say that an operation f is *singular in the ith argument* iff the ith argument (in its signature) is singular. The specification language extended with such annotations of the signatures will be referred to as $\mathcal{L}^*$.

These are the only extensions of the language we need. We may, optionally, use superscripts t^* at any (sub)term to indicate that it is passed as a plural argument. The outermost applications, e.g., f in $f(\ldots)$, are always to be understood plurally, and no superscripting will be used at such places.

DEFINITION 5.1. *Let Σ be a $\mathcal{L}^*$-signature. A is a* Σ-power structure $A \in PStr(\Sigma)$ *iff A is a (deterministic) structure such that*

1. *for every sort* S*, the carrier* S^{A} *is a (subset of the) power set* $\mathcal{P}^+(\mathrm{S}^-)$ *of some basis set* S^-*,*
2. *for every* $f\colon \mathrm{S}_1 \times \cdots \times \mathrm{S}_\mathrm{n} \to \mathrm{S}$ *in* Σ*, f^A is a $\subseteq$-monotonic function* $\mathrm{S}_1^{\mathrm{A}} \times \cdots \times \mathrm{S}_\mathrm{n}^{\mathrm{A}} \to \mathrm{S}^{\mathrm{A}}$ *such that if the ith argument is* S_i *(singular), then f^A is singular in the ith argument.*

The singularity in the ith argument in this definition refers not to the syntactic notion but to its semantic counterpart.

DEFINITION 5.2. *A function* $f^A\colon \mathrm{S}^{\mathrm{A}}_1 \times \cdots \times \mathrm{S}^{\mathrm{A}}_n \to \mathrm{S}^{\mathrm{A}}$ *in a power structure* A *is* singular in the ith argument *iff it is* $\cup$*-additive in the* i*th argument, i.e., iff for all* $x_i \in \mathrm{S}^{\mathrm{A}}_i$ *and all* $x_k \in \mathrm{S}^{\mathrm{A}}_k$ *(for* $k \neq i$*),* $f^A(x_1 \cdots x_i \cdots x_n) = \cup\{f^A(x_1 \cdots \{x\} \cdots x_n) \mid x \in x_i\}$.

Thus, the definition of power structures requires that syntactic singularity be modeled by the semantic one.

Note the unorthodox point in the definition: we do not require the carrier to be the whole power set but allow it to be a *subset* of some power set. Usually one assumes a primitive nondeterministic operation with the predefined semantics as set union. Then all finite subsets are needed for the interpretation of this primitive operator. Also, the join operation (under the set inclusion as partial order) corresponds exactly to set union only if all sets are present (see Example 6.8). None of these assumptions seem necessary. Consequently, we do not assume any predefined (choice) operation but, instead, give the user means of specifying any nondeterministic operation (including choice) directly.

Let Σ be a signature, A a Σ-power structure, X a set of singular variables and X^* a set of plural variables, and $\boldsymbol{\beta}$ an assignment $X \cup X^* \to |A|$ such that for all $x \in X : |\boldsymbol{\beta}(x)| = 1$. (Saying assignment we will from now on mean only assignments satisfying this last condition.) Then, every term $t(x, x^*) \in W_{\Sigma,\mathrm{X},\mathrm{X}^*}$ has a unique set interpretation $\beta[t(x, x^*)]$ in A defined as $t^A(\boldsymbol{\beta}(x), \boldsymbol{\beta}(x^*))$.

DEFINITION 5.3 (Satisfiability). *Let* A *be a* Σ*-power structure and* $\boldsymbol{\pi}$*:* $t_i \frown s_i \mapsto p_j = r_j,\ m_k \prec n_k$ *be a sequent over* $\mathcal{L}^*(\Sigma, X, X^*)$*.* A satisfies $\boldsymbol{\pi}$*,* $A \models \pi$*, iff for every assignment* $\boldsymbol{\beta}$*:* $X \cup X^* \to |A|$*, we have that*

$$\bigwedge_i \boldsymbol{\beta}[t_i] \cap \boldsymbol{\beta}[s_i] \neq \emptyset \Rightarrow \bigvee_j \boldsymbol{\beta}[p_j] \equiv \boldsymbol{\beta}[r_j] \vee \bigvee_k \boldsymbol{\beta}[m_k] \subseteq \boldsymbol{\beta}[n_k].$$

A *is a* power model *of the specification* $\mathrm{SP} = (\Sigma, \boldsymbol{\Pi})$*,* $A \in PMod(SP)$*, iff* $A \in \mathrm{PStr}(\Sigma)$ *and* A *satisfies all axioms from* $\boldsymbol{\Pi}$*.*

Except for the change in the notion of an assignment, this is identical to Definition 4.4, which is the reason for retaining the same notation for the satisfiability relation.

5.2. The calculus for plural parameters. The calculus NEQ is extended with one additional rule:

$$\text{(R8)} \qquad \frac{\boldsymbol{\Gamma} \mapsto \boldsymbol{\Delta}}{\boldsymbol{\Gamma}^{x^*}_{t} \mapsto \boldsymbol{\Delta}^{x^*}_{t}}.$$

Rules (R1)–(R7) remain unchanged, but now all terms t_i belong to $W_{\Sigma,\mathrm{X},\mathrm{X}^*}$. In particular, any t_i may be a plural variable. We let NEQ* denote the calculus NEQ + R8.

The new rule (R8) expresses the semantics of plural variables. It allows us to substitute an arbitrary term t for a plural variable x^*. Taking t to be a singular variable x, we can thus exchange plural variables in a provable sequent $\boldsymbol{\pi}$ with singular ones. The opposite is, in general, not possible because rule (R1) applies only to singular variables. For instance, a plural variable x^* will satisfy $\mapsto x^* \prec x^*$, but this is not sufficient for performing a general substitution for a singular variable. The main result concerning PMod and NEQ* is as follows.

THEOREM 5.4. *For any* $\mathcal{L}^*$*-specification SP and* $\mathcal{L}^*$*(SP) sequent* $\boldsymbol{\pi}$*:*

$$PMod(SP) \models \boldsymbol{\pi} \text{ iff } \boldsymbol{\Pi} \vdash_{NEQ^*} \boldsymbol{\pi}.$$

Proof idea. The proof is a straightforward extension of the proof of Theorem 4.5. □

6. Comparison. Since plural and singular semantics are certainly not one and the same thing, it may seem surprising that essentially the same calculus can be used for reasoning about both. One would perhaps expect that PMod, being a richer class than MMod, will satisfy fewer formulas than the latter and that some additional restrictions of the calculus would be needed to reflect the increased generality of the model class. In this section we describe precisely the relation between the $\mathcal{L}$ and $\mathcal{L}^*$ specifications (section 6.1) and emphasize some points of difference (section 6.2).

6.1. The "equivalence" of both semantics. The following example illustrates a strong sense of equivalence of $\mathcal{L}$ and $\mathcal{L}^*$.

Example 6.1. Consider the following plural definition:

$$\mapsto f(x^*) \prec \text{ if } x^* = x^* \text{ then } 0 \text{ else } 1.$$

It is "equivalent" to the collection of definitions

$$\mapsto f(t) \prec \text{ if } t = t \text{ then } 0 \text{ else } 1$$

for all terms t.

In the rest of this section we will clarify the meaning of this "equivalence."

Since the partial order of functions from a set A to the power set of a set B is isomorphic to the partial order of additive (and strict, if we take $\mathcal{P}$ (all subsets) instead of $\mathcal{P}^+$) functions from the power set of A to the power set of B, $[A \to \mathcal{P}(B)] \simeq [\mathcal{P}(A) \to_\cup \mathcal{P}(B)]$, we may consider every multistructure A to be a power structure A^* by taking $|A^*| = \mathcal{P}^+(A)$ and extending all operations in A pointwise. We then have the obvious lemma.

LEMMA 6.2. *Let SP be a singular specification (i.e., all operations are singular in all arguments), let* $A \in \mathrm{MStr(SP)}$*, and let* $\boldsymbol{\pi}$ *be a sequent in* $\mathcal{L}(\mathrm{SP})$*. Then* $A \models \boldsymbol{\pi}$ *iff* $A^* \models \boldsymbol{\pi}$*, and so* $A \in MMod(SP)$ *iff* $A^* \in PMod(SP)$*.*

Call an $\mathcal{L}^*$ sequent $\boldsymbol{\pi}$ *p-ground* (for plurally ground) if it does not contain any plural variables.

THEOREM 6.3. *Let* $\mathrm{SP}^* = (\Sigma^*, \boldsymbol{\Pi}^*)$ *be an* $\mathcal{L}^*$ *specification. There exists a (usually infinite)* $\mathcal{L}$ *specification* $\mathrm{SP} = (\Sigma, \boldsymbol{\Pi})$ *such that*

(1) $W_{\Sigma,\mathrm{X}} = W_{\Sigma^*,\mathrm{X}}$

(2) *for any* p*-ground* $\boldsymbol{\pi} \in \mathcal{L}^*(\mathrm{SP}^*) : \mathrm{PMod}(\mathrm{SP}^*) \models \boldsymbol{\pi}$ iff $\mathrm{MMod}(\mathrm{SP}) \models \boldsymbol{\pi}$.

Proof. Let Σ be Σ^* with all "*" symbols removed. This makes (1) true. Any p-ground $\boldsymbol{\pi}$ as in (2) is then a $\boldsymbol{\pi}$ over the language $\mathcal{L}(\Sigma, X)$.

The axioms $\boldsymbol{\Pi}$ are obtained from $\boldsymbol{\Pi}^*$ as in Example 6.1. For every $\boldsymbol{\pi}^* \in \boldsymbol{\Pi}^*$ with plural variables $x_1^* \cdots x_n^*$, let $\boldsymbol{\pi} = \{\boldsymbol{\pi}^* \begin{smallmatrix} x_1^* \cdots x_n^* \\ t_1 \cdots t_n \end{smallmatrix} \mid t_1 \cdots t_n \in W_{\Sigma,\mathrm{X}}\}$. Obviously, for any $\boldsymbol{\pi} \in \mathcal{L}(\mathrm{SP})$ if $\boldsymbol{\Pi} \vdash_{\mathrm{NEQ}} \boldsymbol{\pi}$ then $\boldsymbol{\Pi}^* \vdash_{\mathrm{NEQ}^*} \boldsymbol{\pi}$. If $\boldsymbol{\Pi}^* \vdash_{\mathrm{NEQ}^*} \boldsymbol{\pi}$ then the proof can be simulated in NEQ. Let $\boldsymbol{\pi}'(x^*)$ be the last sequent used in the NEQ*-proof which contains plural variables x^* and the sequent $\boldsymbol{\pi}'$ be the next one obtained by (R8). Build the analogous NEQ-proof tree with all plural variables replaced by the terms which occupy their place in $\boldsymbol{\pi}'$. The leaves of this tree will be instances of the $\boldsymbol{\Pi}^*$ axioms with plural variables replaced by the appropriate terms, and all such axioms are in $\boldsymbol{\Pi}$. Then soundness and completness of NEQ and NEQ* imply the conclusion of the theorem. □

We now ask whether, or under which conditions, the classes PMod and MMod are interchangeable as the models of a specification. Let SP*, SP be as in the theorem. The one-way transition is trivial. Axioms of SP are p-ground, so PMod(SP*) will satisfy all these axioms by the theorem. The subclass $\downarrow$ PMod(SP*) $\subseteq$ PMod(SP*), where for every P $\in\downarrow$ PMod(SP*) all operations are singular, will yield a subclass of MMod(SP).

For the other direction, we have to observe that the restriction to p-ground sequents in the theorem is crucial because plural variables range over arbitrary, also undenotable, sets. Let MMod*(SP) denote the class of power structures obtained as in Lemma 6.2. It is not necessarily the case that MMod*(SP) $\models \mathbf{\Pi}^*$, as the following argument illustrates.

Example 6.4. Let $M^* \in$ MMod*(SP) have infinite carrier, $\boldsymbol{\pi}^* \in \mathbf{\Pi}^*$ be $t_i \frown s_i \mapsto p_j = r_j$, $m_k \prec n_k$ with $x^* \in \mathcal{V}[\boldsymbol{\pi}^*]$, and $\boldsymbol{\beta}$: $X \cup X^* \to |M^*|$ be an assignment such that $\boldsymbol{\beta}(x^*) = \{\underline{m}_1 \cdots \underline{m}_l \cdots\}$ is a set which is not denoted by any term in $W_{\Sigma,\mathrm{X}}$. Let $\boldsymbol{\beta}_l$ be an assignment equal to $\boldsymbol{\beta}$ except that $\boldsymbol{\beta}_l(x^*) = \{\underline{m}_l\}$, i.e., $\boldsymbol{\beta} = \cup_l \boldsymbol{\beta}_l$. Then $M^* \models \boldsymbol{\beta}[\boldsymbol{\pi}^*]$ iff

$$M^* \models \boldsymbol{\beta}[t_i] \cap \boldsymbol{\beta}[s_i] \neq \emptyset \Rightarrow \boldsymbol{\beta}[p_j] \equiv \boldsymbol{\beta}[r_j] \vee \,..\, \vee \boldsymbol{\beta}[m_k] \subseteq \boldsymbol{\beta}[n_k] \quad \text{iff}$$

$$\text{(a)}\ \ M^* \models \bigcup_l \boldsymbol{\beta}_l[t_i] \cap \bigcup_l \boldsymbol{\beta}_l[s_l] \neq \emptyset \Rightarrow \bigcup_l \boldsymbol{\beta}_l[p_j] \equiv \bigcup_l \boldsymbol{\beta}_l[r_j] \vee \,..\, \vee \bigcup_l \boldsymbol{\beta}_l[m_k] \subseteq \bigcup_l \boldsymbol{\beta}_l[n_k]$$

since operations in M^* are defined by pointwise extension. $M^* \in$ MMod*(SP) implies that, for all l

$$\text{(b)}\ \ M^* \models \boldsymbol{\beta}_l[t_i] \cap \boldsymbol{\beta}_l[s_l] \neq \emptyset \Rightarrow \boldsymbol{\beta}_l[p_j] \equiv \boldsymbol{\beta}_l[r_j] \vee \,..\, \vee \boldsymbol{\beta}_l[m_k] \subseteq \boldsymbol{\beta}_l[n_k].$$

But (b) does not necessarily imply (a). In particular, even if for all l, all intersections in the antecedent of (b) are empty, those in (a) may be nonempty. So we are not guaranteed that $M^* \in$ PMod(SP*).

Thus, the intuition that the multimodels are contained in the power models is not quite correct. To ensure that no undenotable sets from M^* can be assigned to the plural variables we redefine the lifting operator *: MMod(SP) $\to$ PMod(SP) from Lemma 6.2.

Definition 6.5. *Given a singular specification SP and $M \in MMod(SP)$, we denote by* $\uparrow$M *the following power structure:*

(1) $|\uparrow \mathrm{M}| \subseteq \mathcal{P}^+(|\mathrm{M}|)$ *is such that*

(a) *for every* $\underline{n} \in |M|$: $\{\underline{n}\} \in |\uparrow \mathrm{M}|$,

(b) *for every* $\underline{m} \in |\uparrow \mathrm{M}|$ *there exists a* $t \in W_{\Sigma,\mathrm{X}}$, $\underline{n} \in |\mathrm{M}|$ *such that:* $t^M(\underline{n}) = \underline{m}$.

(2) *The operations in* $\uparrow$M *can be then defined by:* $f(\underline{m})^{\uparrow M} = f(t(\underline{n}))^M$.

Then, for any assignment $\boldsymbol{\beta}$: $X^* \to |\uparrow \mathrm{M}|$ there exists an assignment $\boldsymbol{\theta}$: $X^* \to W_{\Sigma,\mathrm{X}}$ (1b) and an assignment $\boldsymbol{\alpha}$: $X \to |M|$ (1a) such that $\boldsymbol{\beta}(x^*) = \boldsymbol{\alpha}\boldsymbol{\theta}(x^*)]$ (2), i.e., such that the diagram in Figure 1 commutes.

Since $M \in$ MMod(SP), it satisfies all the axioms $\mathbf{\Pi}$ obtained from $\mathbf{\Pi}^*$ and the commutativity of the figure gives us the second part of the following.

Corollary 6.6. *Let SP* and SP be as in Theorem* 6.3. *Then*

$$\downarrow PMod(SP^*) \models \mathbf{\Pi},\ \textit{i.e.,} \downarrow PMod(SP^*) \subseteq MMod(SP),$$
$$\uparrow MMod(SP) \models \mathbf{\Pi}^*,\ \textit{i.e.,} \uparrow MMod(SP) \subseteq PMod(SP^*).$$

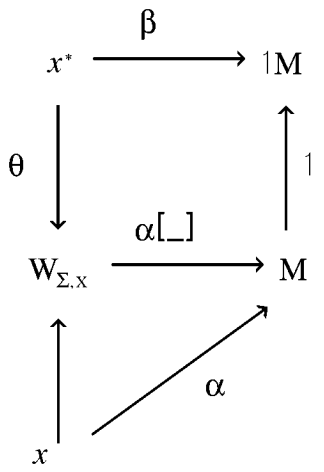


FIG. 1.

The corollary makes precise the claim that the class of power models of a plural specification SP* may be seen as a class of multimodels of some singular specification SP and vice versa. The reasoning about both semantics is essentially the same because the only difference concerns the (arbitrary) undenotable sets which can be referred to by plural variables.

6.2. Plural specification of choice. Plural variables provide access to arbitrary sets. In the following example we attempt to utilize this fact to give a more concise form to the specification of choice.

Example 6.7. The specification

S: { S },
F: { ⊔._ : S* → S },
Π: { ↦ ⊔. $x^* \asymp x^*$ }

defines ⊔. as the choice operator: for any argument t, ⊔. t is capable of returning any element belonging to the set interpreting t.

The specification may seem plausible, but there are several difficulties. Obviously, such a choice operation would be redundant in any specification since the axiom makes ⊔. t observationally equivalent to t, and Lemma 4.6 allows us to remove any occurrences of ⊔. from the (derivable) formulas. Furthermore, observe how such a specification confuses the issue of nondeterministic choice. Choice is supposed to take a *set* as an argument and return *one element* from the set or, perhaps, to convert an argument of type "set" to a result of type "individual." This is the intention behind writing the specification above. But power algebras model all operations as functions on power sets and such a "conversion" simply does not make sense. The only points where conversion of a set to an individual takes place is when a term is passed as a singular argument to another operation. If we have an operation with a singular argument f: S → S, then $f(t)$ will make (implicitly) the choice from t.

This might be particularly confusing because one tends to think of plural arguments as sets and mix up the semantic sets (i.e., the elements of the carrier of a power algebra) and the syntactic ones (as expressed by the profiles of the operations in the

$$\dfrac{\dfrac{\dfrac{x \frown z^*, y \frown z^* \mapsto x \sqcup y \prec z^*}{x \frown p, y \frown p \mapsto x \sqcup y \prec p}\,\text{R8}}{y \frown p \mapsto p \sqcup y \prec p}\,\text{R7}}{\mapsto p \sqcup p \prec p}\,\text{R7} \qquad \dfrac{\mapsto s \prec p \qquad \dfrac{\mapsto t \prec p \quad \mapsto p \sqcup p \prec p \sqcup p}{\mapsto t \sqcup p \prec p \sqcup p}\,\text{R3}}{\mapsto t \sqcup s \prec p \sqcup p}\,\text{R3}$$

$$\dfrac{\mapsto p \sqcup p \prec p \qquad \mapsto t \sqcup s \prec p \sqcup p}{\mapsto t \sqcup s \prec p}\,\text{R3}$$

FIG. 2.

signature). As a matter of fact, the above specification does not at all express the intention of choosing an element from the set. In order to do that it would have to give choice the signature $Set(S) \to S$. Semantically, this would then be a function from $\mathcal{P}^+(Set(S))$ to $\mathcal{P}^+(S)$. Assuming that semantics of $Set(S)$ will somehow correspond to the power set construction, this makes things rather complicated, forcing us to work with a power set of a power set. Furthermore, since $Set(S)$ and S are different sorts, we cannot let the same variable range over both as was done in the example above.

Example 6.7 and remarks illustrate some of the problems with the intuitive understanding of plural parameters. Power algebras, needed for modeling such parameters, significantly complicate the model of nondeterminism as compared with multialgebras.

On the other hand, plural variables allow us to specify the "upper bound" of nondeterministic choice without using disjunction. The choice operation can be specified as the join which under the partial ordering $\prec$ interpreted as set inclusion will correspond to set union (cf. [17]).

Example 6.8. The following specification makes binary choice the join operation wrt. $\prec$:

S: $\{\ \mathrm{S}\ \}$,

F: $\{\ \_\sqcup\_ : \mathrm{S} \times \mathrm{S} \to \mathrm{S}\ \}$,

Π: $\{$ (1) $\mapsto x^* \prec x^* \sqcup y^* \qquad \mapsto y^* \prec x^* \sqcup y^*$

(2) $x \frown z^*, y \frown z^* \mapsto x \sqcup y \prec z^*$ $\}$.

Axiom (2) although using singular variables x, y, does specify the minimality of $\sqcup$ with respect to all terms. (Notice that the axiom $x^* \frown z^*$, $y^* \frown z^* \mapsto x^* \sqcup y^* \prec z^*$ would have a different, and in this context unintended, meaning.) We can show that whenever $\mapsto t \prec p$ and $\mapsto s \prec p$ hold (for arbitrary terms) then so does $\mapsto t \sqcup s \prec p$ (see Figure 2). Violating our formalism a bit, we may say that the above proof shows the validity of the formula stating the expected minimality of join: $t \prec p$, $s \prec p \mapsto t \sqcup s \prec p$.

Thus, in any model of the specification from Example 6.8 $\sqcup$ will be a join. It is then natural to consider $\sqcup$ as the basic (primitive) operation used for defining other nondeterministic operations. Observe also that in order to ensure that join is the same as set union, we have to require the presence of *all* (finite) subsets in the carrier of the model. For instance, the power structure A with the carrier

$$\mathrm{S}^{\mathrm{A}} = \{\ \{1\}, \{2\}, \{3\}, \{1,2,3\}\ \} \text{ and}$$
$$\sqcup^{\mathrm{A}} \text{ defined as } x^{\mathrm{A}} \sqcup^{\mathrm{A}} y^{\mathrm{A}} = \{1,2,3\} \text{ whenever } x^{\mathrm{A}} \neq y^{\mathrm{A}}$$

will be a model of the specification although $\sqcup^{\mathrm{A}}$ is not the same as set union.

7. Conclusion. We have defined the algebraic semantics for singular (call-time-choice) and plural (run-time-choice) passing of nondeterministic parameters. One of the central results reported in the paper is soundness and completeness of two new

reasoning systems NEQ and NEQ*, respectively, for singular and plural semantics. The plural calculus NEQ* is a minimal extension of NEQ which merely allows unrestricted substitution for plural variables. This indicated a close relationship between the two semantics. We have shown that plural specifications have equivalent (modulo undenotable sets) singular formulations if one considers the plural axioms as singular axiom schemata.

Acknowledgments. We are grateful to Manfred Broy for pointing out the inadequacy of our original notation and to Peter D. Mosses for the observation that in the presence of plural variables choice may be specified as join with Horn formulas.

REFERENCES

[1] J. A. Bergstra and J. W. Klop, *Algebra of communicating processes*, in Mathematics and Computer Science, CWI Monographs, 1, North-Holland, Amsterdam, l986, pp. 89–138.
[2] W. Clinger, *Nondeterministic call by need is neither lazy nor by name*, *Proc. ACM Symposium on LISP and Functional Programming*, 1982, pp. 226–234.
[3] E. W. Dijkstra, *A discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[4] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification*, Vol. 1, Springer-Verlag, Berlin, 1985.
[5] J. Engelfriet and E. M. Schmidt, *IO and OI*. 1, J. Comput. System Sci., 15 (1977),pp. 328–353.
[6] J. Engelfriet and E. M. Schmidt, *IO and OI*. 2, J. Comput. System Sci., 16 (1978), pp. 67–99.
[7] J. A. Goguen and J. Meseguer, Completeness of Many-Sorted Equational Logic, *SIGPLAN Notices*, 17 (1982), pp. 9–17.
[8] M. C. B. Hennessy, *The semantics of call-by-value and call-by-name in a nondeterministic environment*, SIAM J. Comput., 9 (1980), pp. 67–84.
[9] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Ltd., Englewood Cliffs, NJ, 1985.
[10] H. Hussmann, *Nondeterministic Algebraic Specifications*, Ph.D. thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1990.
[11] H. Hussmann, *Nondeterminism in Algebraic Specifications and Algebraic Programs*, Birkhäuser, Basel, Switzerland, 1993.
[12] S. Kaplan, *Rewriting with a nondeterministic choice operator*, Theoret. Comput. Sci., 56 (1988), pp. 37–57.
[13] D. Kapur, *Towards a Theory of Abstract Data Types*, Ph.D. thesis, Laboratory for Computer Science, MIT, Cambridge, MA, 1980.
[14] S. Meldal, *An abstract axiomatization of pointer types*, in Proc. 22nd Annual Hawaii International Conference on System Sciences, IEEE Computer Society Press, Piscataway, NJ, 1989.
[15] J. Meseguer, *Conditional rewriting logic as a unified model of concurrency*, Theoret. Comput. Sci., 96 (1992), pp. 73–155.
[16] R. Milner, *Calculi for Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Basel, Switzerland, 1980.
[17] P. D. Mosses, *Unified algebras and institutions*, in *Proc. LICS '89, Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, CA, 1989.
[18] C. E. S. Ore, *Introducing Girard's Quantitative Domains; the Quantitative Domains as a Model for Nondeterminism*, Ph.D. thesis, Dept. of Informatics, University of Oslo, Norway, 1988.
[19] G. Plotkin, Domains, 1983, Lecture notes, Dept. of Computer Science, University of Edinburgh, Scotland.
[20] H. Søndergaard and P. Sestoft, *Non-Determinacy and Its Semantics*, Technical report 86/12, Datalogisk Institut, Københavns Universitet, 1987.
[21] R. L. Schwartz, *An axiomatic treatment of ALGOL* 68 *routines*, in Proc. Sixth Colloquium on Automata, Languages and Programming, Vol. 71, Springer-Verlag, Basel, Switzerland, 1979.
[22] M. B. Smyth, *Power domains*, J. Comput. System Sci., 16 (1978), pp. 23–36.
[23] P. A. Subrahmanyam, *Nondeterminism in abstract data types*, in Automata, Languages and Programming, Lecture Notes in Computer Science, Vol. 115, Springer-Verlag, Basel,

Switzerland, 1981.

[24] M. Walicki, *Algebraic Specifications of Nondeterminism*, Ph.D. thesis, Department of Informatics, University of Bergen, 1993.

[25] M. Walicki and S. Meldal, *Multialgebras, power algebras and complete calculi of identities and inclusions*, in Recent Trends in Data Type Specification, Lecture Notes in Computer Science, Vol. 906, Springer-Verlag, Basel, Switzerland, 1994.

[26] G. Winskel, *An Introduction to Event Structures*, Lecture Notes in Computer Science, Vol. 354, Springer-Verlag, Basel, Switzerland, 1988.

[27] M. Wirsing, *Algebraic specification*, in Handbook of Theoretical Computer Science, Vol. B, The MIT Press, Cambridge, MA, 1990.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1006–1042, August 1997

006

DATA STRUCTURES' MAXIMA*

G. LOUCHARD†, CLAIRE KENYON‡, AND R. SCHOTT§

Abstract. The purpose of this paper is to analyze the maxima properties (value and position) of some data structures. Our theorems concern the *distribution* of these random variables. Previously known results usually dealt with the *mean* and sometimes the variance of the random variables. Many of our results rely on diffusion techniques. This is a very powerful tool that has already been used with some success in algorithm complexity analysis.

Key words. probabilistic analysis of algorithms, data structures, queuing theory, diffusion techniques, Brownian bridges

AMS subject classifications. 60J65, 60J70, 60J60, 60K25, 68P05, 68Q25

PII. S0097539791196603

1. Introduction.

This paper concerns the maximum size reached by a dynamic data structure over a long period of time. The notion of "maximum" is basic to resource preallocation. The expected value of the maximum size has already been studied in numerous probabilistic papers. Our goal here is twofold:

1. Derive more precise asymptotic expressions for the expected maximum (with lower-order terms).

2. When possible, find the whole distribution of the maximum size.

Our proofs involve advanced analytic and probabilistic techniques; in particular, they use Laplace transforms, complex analysis around singularities, diffusion processes, and Brownian motions. They rely on some results that are well known in the world of probabilists. Our hope is to show that diffusion techniques are a powerful tool for the average-case analysis of algorithms.

Given a dynamic data structure whose size evolves through time with each new insertion or deletion, the main requirement for formalizing the maximum-size problem in a mathematical framework is to define the distribution of the sequence of arrivals (insertions) and departures (deletions). The models that we consider come from three main sources.

1. *Probabilistic model* (that is, the world of queuing theory): We study the $M/M/1$, $M/G/1$, $G/M/1$, $M/M/\infty$, and $G/G/\infty$ queuing systems. For example, in the $M/M/\infty$ model, the number of servers is infinite so that a client newly arrived is served right away; there is no waiting time. The interarrivals and service times are all independent and follow an exponential distribution.

2. *Combinatorial models:* In the 1970's, Françon introduced the concept of file histories, which are beautiful combinatorial objects [17]. A file history is a labeled path where each elementary step of the path is of the type $(x, y) \to (x + 1, y \pm 1)$ or $(x + 1, y)$. The x-axis represents time, and the y-axis represents the size of the file. The relative weights of the paths depend on the type of file history considered (stack,

*Received by the editors March 11, 1991; accepted for publication (in revised form) August 8, 1995. This research was partially supported by the PRC "Mathématiques et Informatique."
http://www.siam.org/journals/sicomp/26-4/19660.html

†Laboratoire d'Informatique Théorique, Université Libre de Bruxelles, CP 212, Boulevard du Triomphe, 1050 Bruxelles (Brussels), Belgium (louchard@ulb.ac.be).

‡LIP, Ecole Normale Supérieure de Lyon, 46, Allée d'Italie, 69364 Lyon cedex 07, France (kenyon@lip.ens-lyon.fr).

§CRIN, INRIA-Lorraine, Université de Nancy 1, 54506 Vandoeuvre-lès-Nancy, France (rene.schott@loria.fr).

linear list, dictionary, symbol table, or priority file). However, we must remark that not many real-life situations correspond to the distribution of file histories.

3. *Hashing with lazy deletion:* In the non-Markovian data structure introduced by Van Wyk and Vitter [53], the data structure is a separate-chaining hash table, and the arrivals and lifetimes follow a process of type 1 or 2 above, but the items are not removed from the table as soon as they "die." Instead, we wait until there is a new insertion in a chain of the table before removing the "dead" items from that chain. This enables to save on the access cost to the table.

The results are as follows.

In the first part of this paper, we study the maximum size reached by a queue over $[0 \dots t]$ when t goes to infinity for the most classical types of queues.

When there is only one server processing the requests of the arriving customers, we find expressions describing the distribution for an $M/G/1$, $M/M/1$, or $G/M/1$ process. The technique uses Laplace transforms and developments in the neighborhood of a singularity.

When there are an infinite number of servers, there is no waiting time, and a client stays in the system only as long as it takes for him to be served. This was studied in [46], and in [45], there is an equivalent to the expected value of the maximum size in the $M/M/\infty$ case. In this paper, through a different approach, we get more precise expressions on the expectation. We also solve the $G/G/\infty$ case. Our proof involves reducing the problem to an Ornstein–Uhlenbeck diffusion process, whose properties are well known.

In the second part of this paper, we study some specific kinds of dynamic data structures.

First, we look at combinatorial objects defined by [17] and by Knuth for modeling the evolution of dictionaries, stacks, linear lists, priority files, and symbol tables: file histories. The average value of the maximum was evaluated in [45]. Using sophisticated results on Brownian bridges [14], we show how to get a much more precise estimate of the maximum with several lower-order terms. As an example, we analyze priority queues in the standard model.

Second, we study the maximum size of hashing with lazy deletion. In [53], various models of distribution were suggested. By repeated use of Daniels's theorem [14], we get precise estimates of the average maximum in all of the models with lower-order terms (which are not obtainable by the methods of [45]).

Finally, we look at the limiting profiles of a file history, i.e. what fraction of time is spent by the data structure at a given level l. Again we show that we can obtain the asymptotic distributions.

2. Maximum size of a single server queue when $\rho < 1$. In this section, we study several classical queuing theory models, $M/G/1$ and $G/M/1$, to find the evolution of the maximum size reached over a long period of time.

Notation.

- u_a are interarrival times, assumed to be independent, identically distributed (i.i.d.) random variables (r.v.'s).
- $A(t)$ is the distribution function (d.f.) of u_a.
- $1/\lambda$ is the mean of u_a.
- σ^2 is the variance of u_a.
- u_b are the service times, assumed to be i.i.d., r.v.'s.
- $B(t)$ is the d.f. of u_b.
- $1/\mu$ is the mean of u_b.

- $\rho = \lambda/\mu$.
- $\alpha(s) = \int_o^\infty e^{-st} dA(t)$ $(s \geq 0)$.
- $\beta(s) = \int_o^\infty e^{-st} dB(t)$ $(s \geq 0)$.
- $Q(s)$ is the queue length.
- $M(t) = \max_{s\in[0,t]} Q(s)$.
- $\{x\}$ is the fractional part of x.

To solve our problem, three approaches are possible.

First, we can compute the extreme-value distribution of the maximum on a busy period; we then use renewal theory. Since the queue length is a *discrete* random value, classical techniques give upper and lower bounds (see Cohen [10, Chapter VII, Section 4.5] and Aldous [2, Section C.2.6]). However, it is possible to obtain precise limiting distributions with such an approach; see Heyde [26] for $G/M/1$.

Second, we can use the clumping heuristic (see Aldous [2]).

Third, we can deal directly with the hitting-time distribution; this gives more information on the hitting-time behavior and is the approach chosen here. We easily rederive known theorems on $M/M/1$ and $G/M/1$, write an $M/G/1$ solution, and analyze the Fourier expansion of the maximum moments.

We obtain a family of distributions for the maximum $M(t)$ depending on the fractional part of $C_1 \log(t)+C_2$, where C_1 and C_2 are constants to be determined later. This situation is rather similar to other data structures used in computer science; see, for instance, the number of registers in arithmetic expression evaluation (see Flajolet and Prodinger [18] and Louchard [38]), binary tries (see Louchard [38], Flajolet and Steyaert [19], and Jacquet and Szpankowski [32]), digital search trees (see Louchard [39]), suffix trees (see Jacquet, Rais, and Szpankowski [33]), and approximate counting (see Flajolet [20]). The extreme-value distribution $e^{-e^{-x}}$ frequently appears in this context. We first analyze the $M/G/1$ queue and then the $G/M/1$ queue. (We could not obtain a $G/G/1$ limit theorem). The limiting distribution does not exist for all real values because of the discrete character of $M(t)$. See Anderson [4] and our remarks after Theorem 2.3.

2.1. The $M/G/1$ case.

Notation.

- $\tilde{y}$ is the root[1] greater than 1 and nearest to 1 of $\beta[(1-\tilde{y})\lambda] = \tilde{y}$.
- $\eta = j - \log t/\log\tilde{y} - \log A_1/\log\tilde{y}$.
- $B_1 = 1 - \int_0^\infty e^{-(1-\tilde{y})\lambda t}\lambda t dB(t)$.[2]
- $A_1 = -\lambda(1-\rho)^2(\tilde{y}-1)/B_1$.
- $G_1(\eta) = \exp[-\exp[-\eta\log\tilde{y}]]$.
- $\psi_1(t) = \log(A_1 t)/\log\tilde{y}$.
- $f_1(\eta) = G_1(\eta) - G_1(\eta-1)$.

We note that $\eta = j - \lfloor\psi_1(t)\rfloor - \{\psi_1(t)\}$.

We obtain the following theorem.

THEOREM 2.1. *If j is an integer, then*

$$\Pr[M(t) \leq j] \sim G_1(\eta) \quad \text{as } t = \Theta(\tilde{y}^j) \to \infty. \tag{2.1}$$

The asymptotic distribution is a periodic function of $\psi_1(t)$ (with period 1) given by

$$\log\Pr\{M(t) \leq \lfloor\psi_1(t)\rfloor + k\}e^{\{\psi_1(t)\}\log(\tilde{y})} \to e^{-k\log(\tilde{y})}, \quad t \to \infty.$$

[1] By [10, p. 623], this root does exist if $s_0 < 0$, s_0 being the abscissa of convergence of $\beta(s)$, and if $\beta(s) \to_{s\to s_0} \infty$.

[2] By [10, p. 623], $B_1 < 0$.

We also have

$$\Pr[M(t) = j] \sim f_1(\eta).$$

Proof.

(i) The departure times lead to an imbedded Markov chain whose states correspond to the number of customers just *after* a departure.

Notation.

- τ_j is the hitting time into state j.
- $\varphi_j(s) := E[e^{-s\tau_j}]$, $s \geq 0$.
- $[z^j]f(z)$ is the coefficient of z^j in the power expansion of $f(z)$.
- $y(s)$ is the root with smallest absolute value of equation $z = \beta(s + (1-z)\lambda)$.

By [10, Chapter II, Section 4.35], we have

$$\begin{aligned}\varphi_j(s) = [y(s)]^{-j} &\left\{1 - \left(1 + \frac{s}{\lambda} - y(s)\right)\left[z^j\right]\left(\frac{\beta(s+(1-z)\lambda)}{\beta(s+(1-z)\lambda) - z}\right)\right.\\ &\left.\cdot\ \left(\left[z^j\right]\left(\left(1 + \frac{s}{\lambda} - z\right)\frac{\beta(s+(1-z)\lambda)}{\beta(s+(1-z)\lambda) - z}\right)\right)^{-1}\right\}. \qquad (2.2)\end{aligned}$$

By Takáč's lemma, $y(s) < 1$, and by [10, Chapter II, Section 4.40], $y(s) = 1 - s/\mu(1-\rho) + o(s)$. Expanding (2.2) in the neighborhood of $s = 0$, we derive

$$\begin{aligned}\varphi_j(s) \sim &\left(1 + \frac{js}{\mu(1-\rho)}\right)\left[1 + s\frac{C_{1,j}}{C_{2,j}} - s\left(\frac{1}{\lambda(1-\rho)}\right)\left[\frac{C_{3,j} + sC_{4,j}}{C_{2,j}}\right]\right]\\ &\cdot \left[1 + s\frac{C_{1,j}}{C_{2,j}}\right]^{-1}, \quad s \to 0, \quad j \to \infty, \qquad (2.3)\end{aligned}$$

where we use the following notation.

Notation.

- $\beta_1(z) := \beta[\lambda(1-z)]$.
- $\beta_2(z) := -\int_0^\infty te^{-\lambda(1-z)t}dB(t)$.
- $C_{1,j} := \left[z^j\right]\left((1/\lambda)\beta_1(z)/(\beta_1(z) - z) - z(1-z)\beta_2(z)/(\beta_1(z) - z)^2\right)$.
- $C_{2,j} := \left[z^j\right]\left((1-z)\beta_1(z)/(\beta_1(z) - z)\right)$.
- $C_{3,j} := \left[z^j\right]\left(\beta_1(z)/(\beta_1(z) - z)\right)$.
- $C_{4,j} := \left[z^j\right]\left(-z\beta_2(z)/(\beta_1(z) - z)^2\right)$.

Equivalents of $C_{1,j}$, $C_{3,j}$, and $C_{4,j}$, $j \to \infty$, are given by an asymptotic analysis in the neighborhood of $z = 1$:

$$\begin{aligned}C_{1,j} &\sim \frac{1}{\lambda(1-\rho)^2},\\ C_{3,j} &\sim \frac{1}{1-\rho},\\ C_{4,j} &\sim \frac{j+1}{\mu(1-\rho)^2}.\end{aligned}$$

To obtain $C_{2,j}$ and $C_{5,j} := [C_{1,j} - (1/\lambda(1-\rho))C_{3,j}]$ (which appears as coefficient of s in the numerator of (2.3)), we observe that the corresponding z-functions are regular at $z = 1$. However, the equation that defines $\tilde{y}$ allows an asymptotic analysis in the neighborhood of $z = \tilde{y}$. By Darboux's theorem (see, for instance, Flajolet et al. [23]),

the behavior of $[h(z)]_j$, $j \to \infty$, is related to the behavior of $h(z)$ in the neighborhood of the dominant singularities of $h(z)$.

Notation.

- $s_1^* := \lambda(1-\rho)^2(\tilde{y}-1)/[\tilde{y}^j B_1]$.
- $T_\ell \equiv \tau_{\ell-1}$.
- $\epsilon_{1,\ell} := -\tilde{y}s_1^*$.

Omitting the details, we get $C_{2,j} \sim (1-\tilde{y})/[B_1\tilde{y}^j]$ and $C_{5,j} \sim O(\tilde{y}^{-j})$. Finally, from (2.3), we derive

$$\varphi_j(s) \sim \frac{1}{[1-\frac{s}{s_1^*}]}, \quad s \to s_1^*, \quad j \to \infty.$$

Then $\epsilon_{1,\ell}T_\ell$ is asymptotically distributed as an exponential r.v. as $\ell \to \infty$.

(ii) From $\Pr[T_\ell \geq t] \sim e^{-\epsilon_{1,\ell}t}$, we deduce $\Pr[M(t) < \ell] \sim \exp[-\exp[\log t + \log \epsilon_{1,\ell}]]$ as $t \to \infty$. Taking $\ell := j+1$ and $A_1 := -\lambda(1-\rho)^2(\tilde{y}-1)/B_1$, we readily obtain the theorem.

Notation.

- $\tilde{s_1^*} = s_1^*[1+O(\tilde{y}^{-\ell})]$.
- δ is the absolute error on (2.1).

A detailed analysis shows that

$$\varphi_\ell(s) \sim \frac{1+O(s\ell)}{1-\frac{s}{s_1^*}+O(s^2\ell^2)}, \quad s \to \tilde{s_1^*}, \quad \ell \to \infty. \tag{2.4}$$

We also check that $\phi_\ell(s) \sim \tilde{y}^{-\ell}/|s|^\ell, |s| \to \infty$. The absolute error is given by $\delta = e^{s_1^* t}O(s_1^*\ell)$.

(iii) As $t \to \infty$, we see that the error δ is uniformly small. Thus (2.1) can be put in the form

$$\sup_j |\Pr[M(t) \leq j] - G_1(\eta)| \underset{t\to\infty}{\longrightarrow} 0,$$

which is another appropriate form for a limit theorem [2, Equations (A1.a) and (A10.b)]. □

The asymptotic moments of $M(t)$ are also periodic functions of $\psi_1(t)$. They can be written as harmonic sums and analyzed with Mellin transforms; see Flajolet et al. [21].

The asymptotic nonperiodic term in the moments of $M(t)$ is given by the following result.

THEOREM 2.2. *The constant term $\bar{E}$ in the Fourier expansion (in $\psi_1(t)$) of the moments of $M(t)$ is asymptotically given by*

$$\bar{E}[M(t)-\psi_1(t)]^i \sim \int_{-\infty}^{+\infty} \eta^i [G_1(\eta) - G_1(\eta-1)]d\eta.$$

It is known (see Johnson and Kotz [34, p. 272]) that the extreme-value distribution function $e^{-e^{-x}}$ has mean γ and variance $\pi^2/6$. From this, we can derive, for instance,

$$\bar{E}(M(t)) \sim \psi_1(t) + \frac{1}{2} + \frac{\gamma}{\log(\tilde{y})}. \tag{2.5}$$

The other periodic terms have very small amplitude [21].

2.2. The G/M/1 model. By the same hitting-time technique, this case leads to the following result.

Notation.

- $\tilde{z}$ is the (unique) root of $z - \alpha[(1-z)\mu]$ inside the unit circle.
- $B_3 := \int_o^\infty e^{-(1-\tilde{z})t\mu}\mu t dA(t) - 1$.
- $A_3 := -B_3(1-\tilde{z})\lambda$.
- $\eta := j - \log t/\log(\tilde{z}^{-1}) - \log A_3/\log(\tilde{z}^{-1})$.
- $G_3(\eta) := \exp[-\exp[-\eta\log(\tilde{z}^{-1})]]$.
- $\psi_3(t) := \log(A_3 t)/\log(\tilde{z}^{-1})$.

THEOREM 2.3. *If j is an integer, then*

$$\Pr(M(t) \le j) \sim G_3(\eta), \quad t = \Theta(1/\tilde{z}^j) \to \infty.$$

The last part of Theorem 2.1 *is still valid, with $\psi_1(t)$ replaced by $\psi_3(t)$. Theorem* 2.2 *is still applicable, as well as* (2.5)*, with* $\log(\tilde{y})$ *replaced by* $\log(\tilde{z}^{-1})$.

2.3. Related results.

(i) In the $M/M/1$ case, Theorem 2.3 can be applied with $\tilde{z} \equiv \rho$, $B_3 = -(1-\rho)$, and $A_3 = \lambda(1-\rho)^2$. In this case, the theorem corresponds to the lim inf given in Anderson [4, example on p. 112].

(ii) [10, Section III, Theorem 7.5] gives upper and lower bounds for the distribution of $M(n)$ in $G/M/1$, the maximum number of customers in n busy cycles, and the lower bound matches our theorem. Here the notations ψ and c_1 used in [10] correspond to $\psi \equiv (1-\tilde{z})\mu$ and $c_1 \equiv -B_3$. From [10, Section II, Equation 3.49], we know that the busy-cycle mean length is $\bar{\ell} := 1/[\lambda(1-\tilde{z})]$. From renewal theory, we know that $t/n(t) \to \bar{\ell}$, $t \to \infty$ (where $n(t)$ is the number of busy cycles in $[0,t]$). By the second approach described in the beginning of this section, Heyde [26] derived our theorem.

(iii) In Iglehart [30, Theorem 2], for the $G/G/1$ queue, a (positive) quantity γ is defined by $E[e^{\gamma(u_a - u_b)}] = 1$. It is easy to check that here $\alpha(\gamma)/(1-\gamma/\mu) = 1$, which shows that $\gamma \equiv \psi \equiv (1-\tilde{z})\mu$. [30] then implies [10, Theorems 7.2 and 7.5].

(iv) In a recent report, Sadowsky and Szpankowski [48] analyzed the $G/G/s$ case. They proved under some regularity condition that the maximum number of customers on a busy cycle has a geometric tail with parameter $w = \alpha(\gamma)$ (γ as defined above). However, the coefficient K of $\Pr\{\text{tail} \ge n\} \sim Kw^n$ has a very complicated form; see [48, Equation 4.2.6] (with $L_1^{(n)}$ replaced by $1_{\{\bar{Q} \ge n\}}$). The best way to estimate K would be by simulation. Moreover, the busy-cycle mean length ℓ has no explicit form. In Sadowsky et al. [49], an analysis is derived for $Q_n^{\max}$. We could of course also use Heyde's derivation, but K and ℓ have no computable forms.

(v) Limit distributions are available when $\rho \to 1$ in some particular sense [51, 52].

3. The infinite-server queue maximum problem. In the models where there are an infinite number of servers, a customer arriving always finds a server ready to process its requests. For example, this can model a dynamic data structure with items being inserted in the structure (i.e., customers arriving), staying in for a certain length of time (i.e., service time), and finally being deleted and removed (i.e., exiting the system after service). We will assume that the interarrivals (as well as the service times) are independent and that their distribution is known.

To simplify notation, we will assume in this section that $\mu = 1$. A time-scale change will give the general case.

3.1. The $M/M/\infty$ case.

3.1.1. Weak convergence. Consider an $M/M/\infty$ queue $Q_n(t)$ such that $\lambda_n = n$, $\mu = 1$. It is well known [27, Theorem 4.1] that

$$X_n(t) := \frac{Q_n(t) - n}{\sqrt{n}} \underset{n\to\infty}{\Longrightarrow}{}^{3} \mathrm{OU}(t), \quad t \text{ large},$$

where $\mathrm{OU}(t)$ is a Ornstein–Uhlenbeck process, with infinitesimal mean -1, infinitesimal variance 2, and covariance

$$e^{-(t-s)}, \quad t \geq s. \tag{3.1}$$

From Appendix B, we see that the convergence is correct as far as

$$|X_n(t)|^3 = o(\sqrt{n}) \tag{3.2}$$

or, equivalently

$$|X_n(t)| = o(n^{1/6}), \tag{3.3}$$

i.e., the joint distributions have Gaussian tails $(1 - N(x))$ if $x^3 = o(\sqrt{n})$.[4]

3.1.2. Behavior of λ_n with respect to λ_n/μ. In [45], $\max_{t'\in[0,1]} \bar{Q}(t')$ is analyzed when the parameters satisfy the following three subcases: $\log\lambda' = o(\lambda'/\mu')$, $\log\lambda' = O(\lambda'/\mu')$, $\log\lambda' = \Omega(\lambda'/\mu')$. This corresponds here to the behavior of λ_n with respect to λ_n/μ. We change the time-scale. Let t', λ', and μ' be the new parameters (corresponding to [45]). Letting $t' = h(n)t$, $h(n) \downarrow 0$, and $n \to \infty$, we obtain

$$\left.\begin{aligned} \lambda' &= \frac{\lambda_n}{h(n)} = \frac{n}{h(n)} \to \infty, \\ \mu' &= \frac{1}{h(n)} \to \infty, \end{aligned}\right\} \quad \frac{\lambda'}{\mu'} = n.$$

We derive the convergence

$$X_n(t) := \frac{Q_n(t) - n}{\sqrt{n}} \Rightarrow \mathrm{OU}\left[\frac{t'}{h(n)}\right] \quad \text{if } |X_n(t)| = o(n^{1/6}),$$

and for the queue $\bar{Q}$ of [45], we have in this case

$$\left[\max_{t'\in[0,1]} \bar{Q}(t')\right] \sim n + \sqrt{n}\left[\max_{t\in[0,1/h(n)]} \mathrm{OU}(t)\right] \tag{3.4}$$

The three subcases can be described as follows with $n \to \infty$:

(i) $\log\lambda' \sim (\lambda'/\mu')g(n)$ with $g(n) \downarrow 0$. This leads to $\log n - \log[h(n)] \sim ng(n)$, $1/h(n) \sim e^{ng(n)}/n$, and $\log n/n = o(g(n))$.

(ii) $\log\lambda'/n \to \sigma$, $(\log n - \log[h(n)])/n \to \sigma$, $1/h(n) \sim e^{\sigma n}/n$.

(iii) $\log\lambda'/n \sim f(n)$ with $f(n) \uparrow \infty, 1/h(n) \sim e^{nf(n)}/n$.

[3] $\Rightarrow$ denotes the weak convergence of random functions in the space of all right-continuous functions that have left limits and are endowed with the Skorohod metric (see Billingsley [9]).

[4] $N(a)$ is the standard normal distribution function.

3.1.3. The Ornstein–Uhlenbeck process. From (3.4), we analyze

$$\max_{[0,1/h(n)]} \mathrm{OU}(t).$$

Notation.

- $m_t = [\max_{s\in[0,t]} \mathrm{OU}(s)]$.
- $T_a := \inf[s : \mathrm{OU}(s) = a]$.
- $\delta(a) \sim \sqrt{2\pi}\, e^{a^2/2}/a$.
- $C(t) := 2\log t$.

We shall use two different approaches, each starting from a different probabilistic property.

Approach 1. Let $a \gg 1$. From Keilson [35, Equation 2.31], we know that

$$\Pr[T_a \geq t] \sim e^{-t/\delta(a)}. \tag{3.5}$$

The clumping heuristic also confirms this expression [2, D.9].

$$\Pr[m_t \leq a] = \Pr[T_a \geq t] \sim \exp\left[-\exp\left[\log t - \frac{a^2}{2} + \log a - \frac{1}{2}\log 2\pi\right]\right]. \tag{3.6}$$

To obtain a limiting distribution, set the following in (3.6):

$$a(t) := \sqrt{C(t)} + \frac{\log\log t}{2\sqrt{C(t)}} + \frac{v - \frac{1}{2\log\pi}}{\sqrt{C(t)}} + o\left(\frac{1}{\sqrt{C(t)}}\right), \quad a(t) \uparrow \infty. \tag{3.7}$$

After a few computations, we obtain

$$\Pr[m_t \leq a(t)] \sim e^{-e^{-v}} (\text{extreme-value distribution}), \quad t \to \infty. \tag{3.8}$$

This is a classical known result. It is known (see section 1) that this distribution has mean γ.

Approach 2. Another way of deriving (3.8) is to use Berman's approach [6] (see Appendix C). Using his notation, $w(a), v(a)$, $\Gamma'(0)$, α, $u(t)$, and with $a \to \infty$, [6, Equations 4.3 and 7.1] give $w(a) = f(a)/(1 - F(a))$, where $F(a)$ is the stationary distribution of the process. This gives $w(a) \sim a$. Then (3.1) and [6, Section 7] give $u^2[1 - r(1/v] = 1]$, where r is the process-correlation function. Here $a^2(1 - e^{-1/v(a)}) = 1$, and hence $v(a) \sim a^2$ (index $\alpha = 1$ in Berman's notation).

Notation.

- $\Gamma(x) = P[\int_0^\infty I_{[Z(s)>0]} ds > x]$.
- $Z(s)$ is the process $\eta - t + \sqrt{2}B(t)$.
- η is exponentially distributed.
- $B(\cdot)$ is the classical Brownian motion.

[6, Equation 14.9] gives

$$\frac{\Pr[m_t \geq a]}{vt(1 - F(a))} \to -\Gamma'(0).$$

In Berman [7], it is proved that $\Gamma'(0) = -1$. From Berman's proof, it is easily checked that if we change $\sqrt{2}B(t)$ into $\sigma B(t)$, $\Gamma'(0)$ *does not depend on* σ. Here

$$\frac{\Pr[m_t \geq a]}{a^2 t(1 - \mathbf{N}(a))} \sim 1 \quad \text{for fixed } t, \quad a \to \infty. \tag{3.9}$$

As a check, return to (3.5) for fixed t:

$$\Pr[m_t \geq a] \sim 1 - e^{-t/\delta(a)} \sim \frac{t}{\delta(a)} \sim \frac{at}{\sqrt{2\pi}} e^{-a^2/2}, \quad a \to \infty. \tag{3.10}$$

It is well known that

$$1 - \mathbf{N}(a) \sim \frac{e^{-a^2/2}}{\sqrt{2\pi}a}. \tag{3.11}$$

The identification of (3.10) with (3.9) is immediate.

In [6, Equation 16.9], $u(t)$ is computed from the equation $vt[1-F] = 1$, $t \to \infty$; here

$$u^2 \frac{te^{-u^2/2}}{\sqrt{2\pi}\, u} \sim 1,$$

and after some algebra, we obtain

$$u(t) \sim \sqrt{C(t)} + \frac{\log\log t - \log\pi}{2\sqrt{C(t)}} + o\left(\frac{1}{\sqrt{C(t)}}\right), \quad t \to \infty. \tag{3.12}$$

Finally, [6, Equation 19.2] gives

$$\Pr[w(u(t))[m_t - u(t)] \leq x] \sim e^{\Gamma'(0)e^{-x}}, \quad t \to \infty.$$

we immediately recover (3.8).

3.1.4. Maxima. From (3.7) and (3.8), we obtain

$$\Pr\left[m_t \leq \sqrt{C(t)} + \frac{\log\log t}{2\sqrt{C(t)}} + \frac{v - \frac{1}{2\log\pi}}{\sqrt{C(t)}} + o\left(\frac{1}{\sqrt{C(t)}}\right)\right] \sim e^{-e^{-v}}, \quad t \to \infty,$$

and from (3.4),

$$M := \max_{t' \in [0,1]} Q_n(t') \sim n + \sqrt{n}\, m_{1/h(n)}. \tag{3.13}$$

It is now routine to deduce the following theorem.[5]

Notation.

- $\Pr[Z(n) \leq q(n) + x/\sqrt{2ng(n)}] \sim e^{-e^{-x}}$.
- $q(n) := \sqrt{2ng(n)} + \dfrac{\log(ng(n))}{2\sqrt{2ng(n)}} - \dfrac{\log n}{\sqrt{2ng(n)}} + \dfrac{1}{2}\dfrac{(\log n)^2}{(2ng(n))^{3/2}} - \dfrac{\log\pi/2}{\sqrt{2ng(n)}}$.

THEOREM 3.1. *In the $M/M/\infty$ case, if $g(n) = o(n^{-2/3})$, then subcase* (i) *leads to the following result:*

$$M \sim n + \sqrt{n}\, Z(n) + o\left(\frac{1}{\sqrt{g(n)}}\right), \quad n \to \infty.$$

[5]The condition on $g(n)$ with (3.3) assures weak convergence to an Ornstein–Uhlenbeck process.

3.1.5. Subcase (i) with $g(n) = \Omega(n^{-2/3})$. Now assume that

$$g(n) = \xi n^{-\tau}, \quad 0 < \tau \leq \frac{2}{3}. \tag{3.14}$$

Notation.
- $P(t) := Q_n(t) - Q_n(0)$.
- $X(t) := (Q_n(t) - n)/\sqrt{n}$.[6]

As in [6, Equation 3.4], let us first consider

$$w(u)\left[X\left(\frac{t}{v}\right) - u\right] = w(u)[X(0) - u] + w(u)\frac{P(\frac{t}{v})}{\sqrt{n}}$$

conditioned by $X(0) > u$. $P(t/v)$ for large v is made up of two parts: a Poisson process with rate n and a death process based on a population of size Q_0 and a death rate of t/v. Since v is large, this simply leads to a binomial r.v. We will see later (from (3.24)) that (3.14) leads to $u = o(\sqrt{n})$ and $X(0) - u = o(1)$. Thus

$$E\left[P\left(\frac{t}{v}\right)\right] \sim \frac{nt}{v} - [n + \sqrt{n}u]\frac{t}{v} = -\sqrt{n}u\frac{t}{v} \tag{3.15}$$

and

$$E\left[w(u)\frac{P(\frac{t}{v})}{\sqrt{n}}\right] \sim \frac{-uw(u)t}{v} = -t$$

if we set $v = w(u)u$. This gives $m(t)$ from [6, Equation 5.9].

Moreover, for large v, we formally have for ζ_1 and $\zeta_2 : \mathcal{N}(0,1)$[7] that

$$\begin{aligned}\Delta P := P\left(\frac{\Delta t}{v}\right) - E\left[P\left(\frac{\Delta t}{v}\right)\right] &\sim \sqrt{n}\zeta_1\sqrt{\frac{\Delta t}{v}} - \sqrt{(n + \sqrt{n}u)\frac{\Delta t}{v}}\zeta_2 \\ &\sim \frac{\sqrt{n}}{\sqrt{v}}\sqrt{2}B_1(\Delta t)\end{aligned} \tag{3.16}$$

for a classical Brownian motion $B_1(t)$.[8] Therefore,

$$\frac{w(u)\Delta P}{\sqrt{n}} \sim \sqrt{\frac{w^2(u)}{v}}\sqrt{2}B_1(\Delta t) \sim \sqrt{2}B_1(\Delta t)$$

if we set $v = w^2(u)$. We obtain $V(t)$ from [6, Theorem 5.2].

Finally, the exact stationary distribution of $M/M/\infty$ is well known. It is given by a Poisson distribution with rate ρ. Here $\rho = \lambda_n/\mu = n$.

$$\Pr[Q(t) = k] = \frac{e^{-n}n^k}{k!}. \tag{3.17}$$

The variance and third centered moment of this distribution are given by n.

It is easily checked that by [6, Equations 4.1 and 4.3], this leads to

(i) $w(u) = u$ if $u = o(\sqrt{n})$ so that $v = u^2$;

[6] We should write P_n and X_n, but we simplify notation.

[7] (standard Gaussian random variables)

[8] This can be rigorously proved by weak convergence.

(ii) an exponential distribution for $w(u)[X(0)-u], X(0)>u$.

Thus assumption (i) of [6, Theorem 3.1] is satisfied. Let us remark that the same arguments lead to

$$dX(t) = -X(t)dt + \sqrt{2}B_2(dt) \tag{3.18}$$

for $X(t) = o(\sqrt{n})$, which is exactly the stochastic differential equation for OU(t).

It remains to derive u from [6, Equation 16.9]. We could start from the stationary Poisson distribution (3.17) and develop pure analytical asymptotics, but we prefer to use a more probabilistic large-derivation approach in the spirit of [16], which clearly leads to successive corrections.

Let

$$u = \alpha n^{\beta}, \quad 0<\beta<\frac{1}{2}, \tag{3.19}$$

and $Y(t) := Q_n(t) - n$. It is well known that a Poisson input process with rate n is asymptotically equivalent to a uniform repartition of a population of $\tilde{n} := nt_0$ customers on a time interval $[0, t_0]$ (with $t(n) = o(t_0(n))$). Therefore, for large t, the size of the queue at time t is a binomial with parameters $\tilde{n}$ and $\tilde{p} = (1/t_0)\int_o^t e^{-(t-v)}dv \sim 1/t_0$ for large t. [16, Equation 7.3] gives

$$\Psi(s) \sim \log\left\{\left[\frac{1}{t_0}e^s + \left(1-\frac{1}{t_0}\right)\right]e^{-s/t_0}\right\}. \tag{3.20}$$

The variance of the binomial is $\tilde{\sigma^2} = \tilde{p}(1-\tilde{p}) \sim 1/t_0$.

Following the notation of [16], we have here (x in the notation of [16] is given by u in the notation of [6]) that

$$x = \frac{Y}{\tilde{\sigma}\sqrt{\tilde{n}}} \sim \frac{Y}{\sqrt{n}} = u = \alpha n^{\beta}.$$

(3.20) gives

$$\Psi'(s) \sim \frac{e^s-1}{t_0}, \quad \Psi''(s), \Psi'''(s), \Psi''''(s) \sim \frac{e^s}{t_0}. \tag{3.21}$$

[16, Equation 7.8] leads to $(e^s-1)/t_0 = (1/\sqrt{t_0})(\alpha n^{\beta}/\sqrt{\tilde{n}})$ or $e^s - 1 = \alpha n^{\beta-1/2}$, i.e., $s \sim \alpha n^{\beta-\frac{1}{2}}(1-\alpha n^{\beta-1/2}/2)$.

[16, Equation 7.14] gives $\bar{x} - x = s\sqrt{\tilde{n}\frac{e^s}{t_0}} - \alpha n^{\beta} = O(n^{3\beta-1})$ so that $x^2/2 - \bar{x}^2/2 = O(n^{4\beta-1}) = O(u^4/n)$.

Finally, from [16, Equation 7.28], we derive—instead of (3.11)—the following tail for the distribution of $X(t) := Y(t)/\sqrt{n}$:

$$1 - F(u) \sim [1-\mathbf{N}(u)]\exp\left(\frac{\lambda_1 u^3}{\sqrt{n}} + O\left(\frac{u^4}{n}\right)\right), \tag{3.22}$$

where here $\lambda_1 = 1/6$ by (3.21). If $u^4 = o(n)$, [6, Equation 16.9] leads to

$$\frac{u^2 t e^{-\frac{u^2}{2}} e^{\frac{u^3}{6\sqrt{n}}}}{\sqrt{2\pi}u} = 1. \tag{3.23}$$

If we choose $g(n) = \xi_1 n^{-2/3}$, then $C(t) \sim 2\xi_1 n^{1/3}$. The first approximation in (3.12) is $u(t) \sim \sqrt{2\xi_1 n^{1/3}}$, and after a little algebra, (3.23) shows that we must add $\kappa_1 := (2\xi_1^{3/2}/6)$ to $-\log \pi/2$ in $g(n)$ in Theorem 3.1 if the mixing conditions are satisfied (so that we can use [6, Equation 16.9]), i.e., if there is no long-range dependency. However, as mentioned in [8, p. 21], the mixing property is satisfied in the cases of diffusion, random walk, and birth and death processes, which cover the present case. Thus we obtain the following result.

THEOREM 3.2. *If $g(n) = \xi_1 n^{-2/3}$, then Theorem* 3.1 *is valid with $\kappa_1 = (2\xi_1)^{3/2}/6$ added to $-\log \pi/2$ in $g(n)$.*

If $g(n) = \xi_2 n^{-1/2}$, we now use [16, Equation 7.29]; an extra term $C_1 u^4/n$ appears in (3.23) and a similar analysis would lead to another correction in Theorem 3.1 (we omit the details). An increasing number of corrections are needed as τ decreases in (3.14), but we always have:

$$u(t) \sim \sqrt{C(t)} = O(n^{(1-\tau)/2}) = o(\sqrt{n}), \tag{3.24}$$

and (3.19) is satisfied with $\beta = (1-\tau)/2$.

Let us check our result with the clumping heuristic of Aldous. By [2, Equation D.2b], the clump rate is $\lambda_u = f(u)\mu(u)$, where by (3.22), the density $f(u) \sim n(u)\exp(\lambda_1 u^3/\sqrt{n} + O(u^4/n))$, and by (3.18), $\mu(u) = u$. After standard algebra, [2, Equation D.2c] leads to the theorem.

3.1.6. Subcase (ii) with $g(n) = \sigma$. We will see in (3.33) that this subcase corresponds to $u = O(\sqrt{n})$. Let $u = \alpha\sqrt{n}$ (i.e., x in the notation of [16]). [16, Equation 7.8] leads to $e^s - 1 \sim \alpha$ or $s \sim \log(1+\alpha)$. (3.21) and [16, Equation 7.14] give

$$\bar{x} \sim \log(1+\alpha)\sqrt{n}e^{s/2} \sim \sqrt{n}\log(1+\alpha)\sqrt{1+\alpha}.$$

From [16, Equations 7.11 and 7.23], we derive

$$\begin{aligned} 1 - F(u) &\sim \frac{e^{-\frac{\bar{x}^2}{2} + \tilde{n}[\Psi(s) - s\Psi'(s) + \frac{1}{2}s^2\Psi''(s)]}}{\sqrt{2\pi}\bar{x}} \\ &\sim \frac{e^{n[\alpha - (\alpha+1)\log(1+\alpha)]}}{\sqrt{2\pi}\sqrt{n}\log(1+\alpha)\sqrt{1+\alpha}} = \frac{e^{-n\varphi_1(\alpha)}}{\sqrt{n}\varphi_2(\alpha)} \text{ (say)} \end{aligned} \tag{3.25}$$

with $\varphi_1(\alpha) \geq 0$. (This can also be checked directly from (3.17) or by using a large-deviation technique based on the binomial-generating function $[(1 - 1/t_o) + z/t_o]$.)

The corresponding asymptotic density is given by

$$\frac{e^{-n\varphi_1(\alpha)}}{\sqrt{2\pi}\sqrt{1+\alpha}}du. \tag{3.26}$$

(3.25) and [6, Equation 4.3] now lead to

$$w(u) \sim \sqrt{n}\log(1+\alpha). \tag{3.27}$$

With (3.25), it is easily checked that [6, Equation 4.1] gives an exponential distribution for $w(u)[X(0) - u]$.

To obtain $m(t)$ as given in [6, Equation 5.9], we compute, with $P(t) := Q_n(t) - Q_n(0)$,

$$E\left[w(u)\frac{P(t/v)}{\sqrt{n}}\right] \sim \frac{-uw(u)t}{v}. \tag{3.28}$$

This gives $-t$ if we choose

$$v = w(u)u \tag{3.29}$$

or, equivalently, $v = n\alpha\log(1+\alpha)$. Now, however, $P(t/v) = O(1)$, so we instead use $v' = v/c(n)$ with $c(n)$ increasing and $c(n) = o(\sqrt{n})$. (3.28) becomes $-c(n)t$.

Again, the Poisson input and the binomial death process lead to

$$\begin{aligned}\Delta P &\sim \left[\sqrt{n}\zeta_1\sqrt{\frac{\Delta t}{v'}} - \sqrt{n(1+\alpha)}\zeta_2\sqrt{\frac{\Delta t}{v'}}\right]\\ &\sim \sqrt{\frac{(2+\alpha)n}{v'}}B_3(\Delta t).\end{aligned} \tag{3.30}$$

With (3.27) and (3.29), this gives

$$w(u)\frac{\Delta P}{\sqrt{n}} \sim \sqrt{\frac{(2+\alpha)\log(1+\alpha)}{\alpha}}B_3(\Delta t)\sqrt{c(n)},$$

which *depends on* α. This gives $V(t)$ from [6, Theorem 5.2].

Here, in the neighborhood of $\alpha\sqrt{n}$, (3.18) becomes

$$\begin{aligned}dX(t) &= -X(t)dt + \sqrt{2+\alpha}B_6(dt)\\ &= -X(t)dt + \sqrt{2 + \frac{X(t)}{\sqrt{n}B_6(dt)}}.\end{aligned} \tag{3.31}$$

The corresponding diffusion has space-dependent infinitesimal variance. From Berman [5], the limiting distribution for the Maximum exists. With the results of Berman [6], we check that $\Gamma'(0) = -c(n)$; α remains asymptotically constant during the clump ($w(u)[X(0)-u]$ is exponential). Therefore, modulo the mixing conditions, [6, Equation 19.2] leads to

$$\Pr\left[m_t < u(t) + \frac{x'}{w}\right] \sim e^{\Gamma'(0)e^{-x'}}, \quad t\to\infty,$$

where $u(t)$ is given by [6, Equation 16.9], i.e.,

$$\frac{n\alpha\log(1+\alpha)}{c(n)}\frac{e^{n\sigma-\log n-n\varphi_1(\alpha)}}{\sqrt{2\pi}\,\sqrt{n\log(1+\alpha)\sqrt{1+\alpha}}} = 1. \tag{3.32}$$

Set α_0 as the solution of $\sigma - \varphi_1(\alpha_0) = 0$. Then the solution of (3.32) is $\alpha_0 + \Delta\alpha$ with, on the first order,

$$\Delta\alpha \sim \frac{[\log(\frac{\alpha_0}{c(n)\sqrt{2\pi}\sqrt{1+\alpha_0}}) - \frac{1}{2}\log n]}{(n\log(1+\alpha_0))}.$$

Setting $x' = x + \log(c(n))$, this leads to the following result.[9]

THEOREM 3.3. *If* $g(n) = \sigma$ *(i.e.,* $t = e^{n\sigma}/n$*), then* (3.13) *is satisfied, with*

$$\Pr\left[m_{1/h(n)} \le \sqrt{n}[\alpha_0+\Delta\alpha] + \frac{x}{\sqrt{n}\log(1+\alpha_0)}\right] \sim e^{-e^{-x}}, \quad n\to\infty. \tag{3.33}$$

Note that the dominant term $\sqrt{n}\alpha_0$ corresponds to Mathieu and Vitter [45].

Again, [2, Equation D.E.b] confirms our theorem. The density is given by (3.26), the clump rate $= \alpha\sqrt{n}$ is given by (3.31), and routine computation leads to (3.33).

[9] (3.31) entails the mixing property.

3.1.7. Subcase (iii) with $1/h(n) \sim e^{nf(n)}/n$. Let us use the approach of Aldous. (Berman's technique leads to the same result.) We have $t = e^{nf}/n$, which gives $\log t = nf - \log n$.[10] Set $u = \alpha\sqrt{n}g(n)$, where g will be determined later on. Proceeding as in case (ii), we obtain

$$\begin{aligned} 1 - F(u) &\sim \frac{\exp[n(\alpha g - \log(1+\alpha g) - \alpha g \log(1+\alpha g))]}{\sqrt{2\pi}\ \sqrt{n}\log(1+\alpha g)\sqrt{1+\alpha g}} \\ &= \frac{e^{-n\varphi_3(\alpha,g)}}{\sqrt{n}\varphi_4(\alpha,g)} \end{aligned}$$

with φ_3 and φ_4 defined appropriately, and $\varphi_3' = g\log(1+\alpha g)$, $\varphi_3'' \sim g$.

The corresponding density is given by

$$\psi(u) = \frac{e^{-n\varphi_3(\alpha,g)}}{\sqrt{2\pi}\ \sqrt{1+\alpha g}}.$$

By (3.31) we now have, in the neighborhood of $u = \alpha\sqrt{n}g(n)$,

$$dX(t) \sim -X(t)dt + \sqrt{\alpha g}B_6(dt). \tag{3.34}$$

We must give the same order of magnitude to the two parts of (3.34). Thus we make the time-scale change $t = gt'$. This gives

$$dX(t') = -X(t')g dt' + \sqrt{\alpha}g B_6(dt').$$

We now proceed to the space-scale change $z' = z/g$. The new process $Z := X/g$ satisfies

$$\begin{aligned} dZ(t') &= -X(t')dt' + \sqrt{\alpha}B_6(dt') \\ &\sim -\alpha\sqrt{n}g dt' + \sqrt{\alpha}B_6(dt'), \end{aligned}$$

which is again a diffusion stochastic differential equation.

Aldous's clump rate is given by $\lambda_b = g\psi(u)\cdot\alpha\sqrt{n}g$, and with $t' = e^{nf}/(ng)$, [2, Equation D.2.c] leads to

$$\Pr\left[\sup_{[0,t']} Z(s) \le \alpha\sqrt{n}\right] \sim e^{-\lambda_b t'}$$

or

$$\Pr[m_t \le \alpha\sqrt{n}g] \sim e^{-\lambda_b t'}. \tag{3.35}$$

To get a limiting distribution, we choose α and g such that $\log(\lambda_b t') = -x$. We can set $\alpha = 1 + \Delta_\alpha$. The equation for g is given by

$$\begin{aligned} &n[\alpha g - \log(1+\alpha g) - \alpha g\log(1+\alpha g)] + nf - \log n - \log g - \log(\sqrt{2\pi}) \\ &\quad + \frac{3}{2}\log g + \frac{1}{2}\log\alpha + \frac{1}{2}\log n = -x. \end{aligned} \tag{3.36}$$

[10] We drop the n dependency to ease notation.

Set g such that the dominant term (in n) of (3.36) is equal to 0 for $\alpha = 1$. This leads to

$$g - \log(1+g) - g\log(1+g) + f = 0.$$

After some algebra, we obtain

$$g = \frac{f}{\log f}\left[1 + \frac{\log\log f}{\log f} + O\left(\frac{1}{\log f}\right)\right].$$

Note that the full expansion must be used in (3.36). For this value of g, we now compute Δ_α. Then (3.36) gives, on the first order (the other terms can be neglected),

$$\Delta_\alpha \sim \frac{\left[-x + \frac{1}{2}\log n - \frac{1}{2}[\log f - \log\log f] + \log(\sqrt{2\pi})\right]}{(-nf)},$$

and finally, equation (3.35) leads to the following result.

THEOREM 3.4. *If $1/h(n) \sim e^{nf(n)}/n$, then* (3.13) *is satisfied with*

$$\Pr[m_{1/h(n)} \le \sqrt{n} g(1+\Delta_\alpha)] \sim e^{-e^{-x}}, \quad n \to \infty. \tag{3.37}$$

Again, the dominant term $\sqrt{n} f/\log f$ corresponds to Mathieu and Vitter [45].

Remark. A similar analysis can be done for $M/G/\infty$, which is not Markovian but has Poisson arrivals. Note that the analysis of the case $\rho =$ constant is done in Aldous et al. [3].

3.1.8. The maximum mean. We now turn to the mean

$$\bar{M} := E\left[\max_{t' \in [0,1]} Q_n(t')\right] \sim n + \sqrt{n}\, E[m_{1/h(n)}].$$

We deduce the following theorem.

THEOREM 3.5. *In the $M/M/\infty$ case, if $g(n) = o(n^{-2/3})$, the first subcase leads to the following result:*

$$\bar{M} \sim n + \sqrt{n} q(n) + \frac{\gamma}{\sqrt{2ng(n)}} + o\left(\frac{1}{\sqrt{g(n)}}\right), \quad n \to \infty.$$

Proof.

(i) Let $g(n) = \xi_1 n^{-\tau}, 2/3 < \tau < 1$. Theorem 3.1 is valid (i.e., we are in the Gaussian range) if $|x|/\sqrt{ng(n)} = o(n^{1/6})$ (see (3.3)), i.e., if $|x| = o(n^{2/3-\tau/2})$. For instance, we set

$$|x| = O(n^{2/3-\tau/2-\epsilon}), \quad 0 < \epsilon < \frac{1}{6}. \tag{3.38}$$

Let $x_1 = -\xi_2 n^{2/3-\tau/2-\epsilon}$, $0 < \epsilon < 1/6$. Then

$$P_1 := \Pr\left[M \le n + \sqrt{n} q(n) + \frac{x_1}{\sqrt{2\xi_1 n^{-\tau}}}\right] \sim \exp\left[-e^{\xi_2 n^{2/3-\tau/2-\epsilon}}\right].$$

However, nP_1 can be made arbitrary small by choosing n sufficiently large. The contribution to the mean of the lower tail of the distribution of M is asymptotically negligible.

(ii) Let us now turn to the upper tail. Begin the process with the stationary distribution (3.17). Divide the observation interval $e^{ng(n)}/n$ into intervals of size $1/n^{1-\tau}$. This gives $e^{\xi_1 n^{1-\tau}}/n^\tau$ intervals. Then

$$\Pr[M \geq n + \sqrt{n}u] \leq \frac{e^{\xi_1 n^{1-\tau}}}{n^\tau} \Pr\left[\max_{s\in[0,n^{-(1-\tau)}]} Q_n(s) \geq n + \sqrt{n}u\right]. \tag{3.39}$$

Let $x_2 = \xi_2 n^{1-\tau}$, which satisfies (3.38).

We know from Theorem 3.1 that $q(n) \sim \sqrt{2\xi_1 n^{1-\tau}}$. Choose ξ_2 such that $\sqrt{2} + \xi_2/\sqrt{2}\xi_1 = 4$.

We must then analyze the tail $\Pr[M \geq n + \sqrt{n}4u_0]$ with $u_0 = \sqrt{\xi_1 n^{1-\tau}}$.

(ii)a Let us first consider the range $[n + \sqrt{n}4u_0, n(1+\alpha_0)]$, $\alpha_0 > 0$. Note that u_0 is in the Gaussian range. From (3.39), we must consider

$$\begin{aligned}\Pr\left[\max_{[0,n^{-(1-\tau)}]} Q_n(s) \geq n + \sqrt{n}4u_0\right] &\leq \Pr[Q_n(0) \geq n + \sqrt{n}2u_0] \\ &+ \Pr\left[\max_{[0,n^{-(1-\tau)}]} Q_n(s) \geq n + \sqrt{n}4u_0 | Q_n(0) = n + \sqrt{n}2u_0\right].\end{aligned}$$

By (3.11), the first term on the right-hand side is bounded by

$$\frac{e^{-2u_0^2}}{\sqrt{2\pi}2u_0}.$$

For the second term, by (3.15) and (3.16) (setting $v = n^{(1-\tau)}$), we asymptotically obtain

$$\begin{aligned}&\Pr\left[\max_{t\in[0,1]} \left(\frac{-\sqrt{n}2u_0 t}{n^{1-\tau}} + \frac{\sqrt{n}\sqrt{2}B_1(t)}{n^{(1-\tau)/2}}\right) \geq \sqrt{n}4u_0\right] \\ &\quad = \Pr\left[\max_{[0,1]}(-\sqrt{2\xi_1}t + B_1(t)) \geq \sqrt{\frac{\xi_1}{2}}4n^{1-\tau}\right] \\ &\quad \leq \frac{2e^{-4\xi_1 n^{2(1-\tau)}}}{\sqrt{2\pi}\sqrt{\frac{\xi_1}{2}}4n^{1-\tau}}.\end{aligned}$$

By (3.39), we finally obtain

$$\begin{aligned}P_2 &:= \Pr[M \geq n + \sqrt{n}4u_0] \\ &\leq \frac{e^{\xi_1 n^{1-\tau}}}{\sqrt{2\pi}n^\tau}\left[\frac{e^{-2\xi_1 n^{1-\tau}}}{2\sqrt{\xi_1 n^{1-\tau}}} + \frac{2e^{-4\xi_1 n^{2(1-\tau)}}}{\sqrt{\frac{\xi_1}{2}}4n^{1-\tau}}\right].\end{aligned}$$

Obviously, $n(1+\alpha_0)P_2$ can be made arbitrary small by taking n sufficiently large.

(ii)b We now analyze the range $[n(1+\alpha_0), \infty]$. Again, from (3.39), we consider

$$\begin{aligned}\Pr\left[\max_{[0,n^{-(1-\tau)}]} Q_n(s) \geq n + n\alpha\right] &\leq \Pr\left[Q_n(0) \geq n + \frac{n\alpha}{2}\right] \\ &+ \Pr\left[\max_{[0,n^{-(1-\tau)}]} Q_n(s) \geq n + n\alpha | Q_n(0) = n + \frac{n\alpha}{2}\right].\end{aligned}$$

By (3.25), the first term is bounded by

$$\frac{e^{-n\varphi_1(\alpha/2)}}{\sqrt{n}\varphi_2(\frac{\alpha}{2}rgy)}.$$

With (3.30), the second term is asymptotically given by

$$\begin{aligned}
&\Pr\left[\max_{t\in[0,1]}\left(\frac{-n\alpha t}{2n^{(1-\tau)}}+\frac{\sqrt{n(2+\frac{\alpha}{2})}B_3(t)}{n^{(1-\tau)/2}}\right)\geq n\alpha\right]\\
&=\Pr\left[\max_{[0,1]}\left(\frac{-\alpha t n^{\tau/2}}{2\sqrt{2+\frac{\alpha}{2}}}+B_3(t)\right)\geq\frac{\alpha n^{1-\tau/2}}{\sqrt{2+\frac{\alpha}{2}}}\right]\\
&\leq\frac{2e^{-\alpha^2n^{2-\tau}/(2(2+\alpha/2))}\sqrt{2+\frac{\alpha}{2}}}{\sqrt{2\pi}\alpha n^{1-\tau/2}}=\frac{e^{-\varphi_4(\alpha)n^{2-\tau}}\varphi_5(\alpha)}{n^{1-\tau/2}}\ \text{(say)}.
\end{aligned}$$

From (3.39), we finally obtain

$$\begin{aligned}
P_3(\alpha)&=\Pr(M\geq n+n\alpha)\\
&\leq\frac{e^{\xi_1 n^{1-\tau}}}{n^\tau}\left[\frac{e^{-n\varphi_1(\alpha/2)}}{\sqrt{n}\varphi_2(\frac{\alpha}{2})}+\frac{e^{-\varphi_4(\alpha)n^{2-\tau}}\varphi_5(\alpha)}{n^{1-\tau/2}}\right].
\end{aligned}$$

However, $\int_{\alpha_0}^{\infty}P_3(\alpha)nd\alpha$ can be made arbitrary small by choosing some suitable α_0 and n large enough. □

3.2. The $G/M/\infty$ case. Let us again set $\lambda_n=n$, $\mu=1$. Assume that $A(t)$ has a finite variance σ^2. By Iglehart [28, Equation 5.3], we know that

$$\frac{Q_n(t)-n}{\sqrt{n}\sqrt{C_1}}\Rightarrow \mathrm{OU}(t),\quad t\text{ large},$$

where $C_1:=(1+\sigma^2)/2$. We readily obtain the following result.

THEOREM 3.6. *In the $G/M/\infty$ case, if $g(n)=o(n^{-2/3})$, subcase* (i) *leads to*

$$M\sim n+\sqrt{nC_1}\,Z(n)+o\left(\frac{1}{\sqrt{g(n)}}\right),\quad n\to\infty,$$

where $Z(n)$ has the same distribution as in Theorem 3.1.

3.3. The $G/G/\infty$ case. Let $\lambda_n=n,\mu=1$. Assume that $A(t)$ has finite variance σ^2. By Iglehart [29, Example 5.2], we know that if $B(t)$ has a continuous density,[11] then

$$\frac{Q_n(t)-nh(t)}{\sqrt{n}}\Rightarrow X^1(t)+X^2(t),$$

where $h(t):=\int_o^t[1-B(\tau)]d\tau$. Here $X^1(t)$ is a Gaussian process with covariance $(s\leq t)$

$$\int_o^s B(s-\tau)[1-B(t-\tau)]d\tau$$

[11] Actually, less severe constraints are sufficient; see [29] for details.

and

$$X^2(t) := \int_o^t [1 - B(t-\tau)]\sqrt{\sigma^2} B_7(d\tau),$$

where B_7 is a classical Brownian motion. As $s, t \to \infty$, it is easily seen that $h(t) \to 1$ and the covariance of $X^1(t) + X^2(t)$ is given by $(s \le t)$

(3.40)
$$\Sigma(t-s) \sim \sigma^2 \int_o^\infty [1 - B(t-s+\tau)][1 - B(\tau)]d\tau + \int_o^\infty B(\tau)[1 - B(t-s+\tau)]d\tau.$$

This is clearly a non-Markovian stationary Gaussian process.

Note that the particular case $\sigma^2 = 1$ leads to the simple form

$$\Sigma(t-s) \sim \int_o^\infty [1 - B(t-s+\tau)]d\tau.$$

This is true, for instance, for the $M/G/\infty$ case.

Returning to (3.41), we see that for $\epsilon \to 0$, the correlation coefficient of the limiting process is given by

(3.41)
$$r(\epsilon) := \frac{\Sigma(\epsilon)}{\Sigma_2} \sim 1 - \frac{\epsilon}{2}\frac{[\sigma^2 + 1]}{C_2} = 1 - r_1\epsilon \text{ (say)},$$

with $\Sigma_2 := \Sigma(0) = 1 + (\sigma^2 - 1)\int_o^\infty [1 - B(\tau)]^2 d\tau$.

(3.41) and [6, Equation 7.2] give $a^2[1 - r(\frac{1}{v})] = 1$, and hence $v(a) \sim r_1 a^2$.

From Berman [7], we can again deduce $\Gamma'(0) = -1$. Now [6, Equation 16.9] computes $u(t)$ from

$$r_1 u^2 \frac{te^{-u^2/2}}{\sqrt{2\pi}u} \sim 1,$$

which leads to

$$u(t) \sim \sqrt{C(t)} + \frac{\log\log t - \log\pi + 2\log r_1}{2\sqrt{C(t)}} + o\left(\frac{1}{\sqrt{C(t)}}\right), \quad t \to \infty.$$

(3.7) is still valid if we add $\log r_1/\sqrt{C(t)}$. The mixing condition can be deduced from Leadbetter et al. [37, Theorem 12.3.5].

We finally derive the following theorem.

THEOREM 3.7. *In the $G/G/\infty$ case, if $g(n) = o(n^{-2/3})$ and $\Sigma(t)\log t \to 0$, $t \to \infty$ ($\Sigma(t)$ is given by* (3.41)*), subcase* (i) *leads to*

$$M \sim n + \sqrt{nC_2}\left[Z(n) + \frac{\log r_1}{\sqrt{2ng(n)}}\right] + o\left(\frac{1}{\sqrt{g(n)}}\right), \quad n \to \infty,$$

where $Z(n)$ has the same distribution as in Theorem 3.1.

Theorem 3.2 now uses $\kappa_1 = \mu_{3,Y}(2\xi_1)^{3/2}/6$.

4. List structures' maxima. In this section, we leave the world of queuing theory models to enter that of combinatorial models for dynamic data structures. More specifically, we are interested in file histories as invented by Françon and developed by Flajolet, Françon, and Vuillemin. The size of a dynamic data structure increases by 1 with each insertion and decreases by 1 with each deletion. The successive sizes as a function of time form a path on the plane, which starts at level 0 (the data structure is initially empty) and normally returns to 0 after a sequence of n operations (insertions, deletions, or queries). A probability distribution is defined by weighing the paths according to the data structure being studied. File histories have been studied by Françon, Flajolet, Puech, Vuillemin, and Viennot in particular, and they have discovered beautiful links with orthogonal polynomials, continued fractions, and various combinatorial objects. In [40] and [41], Louchard, Schott, and Randrianarimanana developed a complete probabilistic analysis of these structures. In [45], the average value of the height of the path (maximum size) was given for some kinds of file histories. In this section, once again we look at the distribution of the maximum size. Assume that the operations happen at times $1, 2, \ldots, 2n$. Our technique is based on the observation that the process can be decomposed into two simple components. Let $S(t)$ denote the size of the data structure at time t ($1 \le t \le 2n$). Then $S(t) = n(\tilde{y} + Z(t))$, where $\tilde{y}$, the *average* size of the structure at time t, is a fairly simple curve (for instance a concave parabola in our first subsection), and $Z(t)$ is a (small) Gaussian process.

4.1. Daniels's fundamental result. All of the remaining results are based on a general theorem by Daniels [14]. We want information about $\mathrm{Max}_{t\in[0,1]}Y(t)$, where Y is a certain random process. Assume that $Y(t)$ can be written as

$$Y(t) = \tilde{z}(t) + Z(t),$$

where $\tilde{z}(t)$ is a certain deterministic curve and $Z(t)$ is a random Gaussian process, of covariance $C(s,t)$. Note that $\tilde{z}(t)$ is *not* random. Let M be the maximum of $Y(t)$ for $t \in [0,1]$ and t^* be the first time at which the maximum M is reached. Thus we can look for the hitting time of $Z(t)$ to the absorbing boundary $M - \tilde{z}(t)$. Near that crossing point, $Z(t)$ locally behaves like a Brownian motion (or a variant of one, such as a Brownian bridge) [15]. It is also known that the hitting time and place densities for a Brownian bridge can be deduced from the hitting time density for a Brownian motion (see, for instance, Louchard [41] for a constant boundary and Csaki et al. [12] for a general proof).

Suppose that we are looking at the maximum size of a data structure over a time interval $[0 \ldots 1]$ when some parameter n goes to infinity (for instance, n might be the number of operations). We will assume that $\tilde{z}(t)$ satisfies

$$\tilde{z}(t) = \sqrt{n}z(t),$$

where $z(t)$ is independent of n. (This assumption will be true for all of the applications in this paper.) Moreover, assume that $z(t)$ has a unique maximum, reached at time $\overline{t}$, with $z(\overline{t}) = 0$. (Up to doing a translation, we can always assume that.)

Daniels has matched the local behavior of $C(s,t)$ with the Brownian bridge covariance near $\overline{t}$ [14]. From [14] and [13], we can deduce information about the maximum M and the time t^* when it is reached.

Notation. Let A and B be the constants defined by

$$A = [\partial_s C]_{\overline{t}} + |[\partial_t C]_{\overline{t}}| \quad \text{and} \quad B = (-z''(\overline{t}))^{-1/3}.$$

Let

$$G(x) = \frac{2^{-1/3}}{2\pi i} \int_{-i\infty}^{i\infty} e^{sx} \frac{ds}{A_i(-2^{1/3}s)},$$

where A_i is the classical Airy function, and let λ be the universal constant defined by

$$\lambda = \int_{-\infty}^{\infty} [e^{x^3/6} G(x) - \max(x, 0)] dx \sim 0.99615.$$

Let

$$u = n^{1/3} A^{-1/3} B^{-2} (t^* - \overline{t}).$$

THEOREM 4.1. *With the above assumptions and notations, if* $[\partial_t C(s,t)]_{\overline{t}} \le 0$ *and* $[\partial_s C(s,t)]_{\overline{t}} > 0$*, then we have the following:*

1. *M is asymptotically Gaussian with mean and variance*

$$\begin{cases} E(M) = \lambda B n^{-1/6} A^{2/3} + O(n^{-1/3}), \\ \sigma^2(M) = C(\overline{t}, \overline{t}) + O(n^{-1/3}). \end{cases}$$

2. *The conditional maximum $M|t^*$ is asymptotically Gaussian with mean and variance*

$$\begin{cases} (M|t^*) = n^{-1/6} A^{-1/3} B \left[[\partial_s C]_{\overline{t}} \dfrac{G'(-u)}{G(-u)} + |[\partial_t C]_{\overline{t}}| \dfrac{G'(u)}{G(u)} \right] + O(n^{-1/3}), \\ \sigma^2(M|t^*) = C(\overline{t}, \overline{t}) + O(n^{-1/3}). \end{cases}$$

3. *The joined density of M and t^* is given by*

$$\phi(M,t) = \frac{2}{\sqrt{2\pi C(\overline{t},\overline{t})}} e^{-M^2/2C(\overline{t},\overline{t})} \left\{ G(t)G(-t) + n^{-1/6} B A^{-1/3} \frac{M}{C(\overline{t},\overline{t})} \right.$$

$$\left. [[\partial_s C]_{\overline{t}} G(t) G'(-t) + |[\partial_t C]|_{\overline{t}} G(-t) G'(t)] + O(n^{-1/3}) \right\}.$$

4. *u has density $2G(u)G(-u)[1 + O(n^{-1/3})]$.*

Proof. A direct new proof is given in Appendix A using modern tools such as the Radon–Nikodym derivative for some absolutely continuous probability measure (see Salminen [50]). □

4.2. File histories and list structure in the Markov model. A file history represents the evolution of the size of a dynamic data structure by a path of length $2n$ = number of operations performed (insertions, queries, or deletions), going from level 0 to level 0. In [45], the average value of the height of the path (i.e., the maximum size) was given for some kinds of data structures, but only a rough equivalent was found.

Here we study the example of priority file histories [22]. Other structures, such as linear lists or dictionaries, could be analyzed with similar techniques. A priority list is by definition a data structure on which no queries are performed: only insertions and deletions occur, and, moreover, deletions happen only for the minimum. Thus when a new item is inserted in a priority list of size k, there are $(k+1)$ intervals defined

by the elements already present and to which the new item may belong, but when an item is deleted, there is only one possible choice. This is reflected in the weights of the paths; see [22].

Assume that there are $2n$ operations performed during the history, n insertions and n deletions. Let $Y(t)$ be the size of the data structure after the $\lfloor nt \rfloor$th operations ($0 \le t \le 2$). From Louchard [40], we know that

$$\forall t, \quad \frac{Y(t) - \frac{1}{2}nt(2-t)}{\sqrt{n}} \Rightarrow X(t) \quad \text{when } n \to \infty,$$

where $X(t)$ is a Markovian Gaussian process with mean 0 and covariance

$$C(s,t) = \frac{s^2(2-t)^2}{4} \quad \text{when } s \le t.$$

The error term can be deduced for the various expansions in [40, Section 4]. It appears that the relative error in the density is $O(1/\sqrt{n})$ (nonuniform in X). Thus in this case, using the notations of Daniels's theorem, we have $z(t) = t(2-t)/2 - 1/2$, with a maximum at $\overline{t} = 1$ and $z''(\overline{t}) = -1$. Since the covariance here is a simple function, we can easily apply the theorem, and we find the following.

THEOREM 4.2. *If $Y_n(t)$ denotes the size of a random priority file history of length $2n$ at time $\lfloor nt \rfloor$, we have*

$$E(\mathrm{Max}_t Y_n(t)) \sim \frac{n}{2} + \lambda n^{1/3} + O(n^{1/6}),$$

and, more generally,

$$\mathrm{Max}_t Y_n(t) \sim \frac{n}{2} + \sqrt{n}\, M + O(n^{1/6})$$

is reached at time t^*, with M and t^* given by Daniels's theorem. We can prove that the error term in the weak convergence to $X(t)$ is negligible *with the error in* Max$Y$.

This is not the first example of applications of diffusion processes to computer science. Other structures have been analyzed using Brownian excursions [42, 44] or Brownian meandering; see [38] for height in planar trees and stack structures' maxima.

5. Limiting profiles of list structures. In the last section, we analyzed the distribution of the maximum of file histories. We can also get information on the limiting profile, i.e., on how much time is spent at each level k (for k fixed and n going to infinity). Our investigations are based on the assumption that the size $Y(\lfloor nt \rfloor)$ of the list at time nt (with $0 \le t \le 2$) satisfies a weak convergence property:

$$\frac{Y(\lfloor nt \rfloor) - ny(t)}{\sqrt{n}} \Rightarrow X(t), \quad 0 \le t \le 2,$$

where y is a symmetric function around 1 and X is a Gaussian process with mean 0 and known covariance C. In all applications to either classical or Knuth-type file histories, this assumption is true.

Let k, the level under study, be fixed, and let $\overline{t}$ be the time such that $y(\overline{t}) = k/n$, with $\overline{t} < 1$. If we consider the time u where Y first hits level k, the density of u is given by

$$\begin{aligned} g(u)du &= \Pr\{\min\{u', Y([nu']) = k\} \in du\} \\ &= \Pr\left\{\min\left\{u', X(u') = \sqrt{n}\left(\frac{k}{n} - y(u')\right)\right\} \in du\right\}, \end{aligned}$$

and so

$$u - \overline{t} = O\left(\frac{1}{\sqrt{n}}\right).$$

Thus we examine the behavior of Y near $\overline{t}$. Locally, y can be approximated by a straight line so that if $y'(\overline{t})$ denotes the slope of y at $\overline{t}$, we have

$$g(u)du \sim \Pr\{\min\{u', X(u') = \sqrt{n}(\overline{t} - u')y'(\overline{t})\} \in du\}.$$

However, it is known that X behaves locally like a Brownian motion [15]. Using classical results on the crossing time of a Brownian motion and a straight line (see Cox and Miller [11, p. 221]) and an asymptotic analysis, we find that the density of $\tau := \sqrt{n}(u - \overline{t})$ is

$$\frac{y'(\overline{t})}{\sqrt{2\pi C(\overline{t},\overline{t})}} e^{-(\tau y'(\overline{t}))^2/2C(\overline{t},\overline{t})} d\tau,$$

where C is the covariance of X. Thus u is a Gaussian variable centered at $\overline{t}$.

We now study the *total time spent* at level k. Let $p(u)$, $q(u)$, and $r(u)$ be the probabilities that the next move on the list is an insertion, deletion, and q, respectively, if we start at $Y([nu])$ at time u. The random walk describing the evolution of the data structure is transient, so the sojourn time is $O(1)$. We can thus replace $p(u)$ by $p(\overline{t}) + O(1/n)$, etc. All distributions are affected by errors of order $O(1/n)$. In the neighborhood of $\overline{t}$, we may consider $p(u)$, $q(u)$, and $r(u)$ to be locally constant and equal to the probabilities p, q, and r of insertion, deletion, and query at level k.

If the first step leaving level k is a deletion, we are sure of coming back to level k (since $\overline{t} < 1$). If it is an insertion, then the probability of ever returning to level k is q/p. If we look at the whole file history after hitting level k, the history spends l steps at level k, i steps inserting from k to $k+1$, d steps deleting from k to $k-1$, plus various other operations at other levels.

Thus if $F(z, w, v)$ is the joint multidimensional generating function

$$F(z, w, v) = \sum_{i,d,l} \Pr\{l, i, d\} z^l w^i v^d,$$

we have when $n \to \infty$ that

$$F(z, w, v) = qzvF(z, w, v) + pw\left[\frac{q}{p} zF(z, w, v) + \left(1 - \frac{q}{p}\right)\right] + rzF(z, w, v),$$

which leads to

$$F(z, w, v) = \frac{(p - q)w}{1 - qz(v + w) - rz}.$$

All of the distributions that we need can be extracted from this equation. For instance, the time ℓ spent at level k is characterized by (letting $w = v = 1$)

$$F_\ell(z) = \frac{p - q}{1 - (2q + r)z}, \tag{5.1}$$

which shows that ℓ is a geometric r.v., with mean $E(\ell) = (2q + r)/(p - q)$. The *total* time spent at k has mean $\tilde{E}_\ell := 1 + E(\ell) = 1/(p - q)$.

The number i of insertions steps *from* k is characterized by (letting $z = v = 1$)

$$F_i(w) = \frac{(p-q)w}{p-qw} = \frac{(1-\frac{q}{p})w}{1-w\frac{q}{p}}.$$

This gives

$$P(i=\kappa) = \left(1-\frac{q}{p}\right)\left(\frac{q}{p}\right)^{\kappa-1} \quad (\text{for } i \geq 1) \tag{5.2}$$

with mean

$$E(i) = \frac{1}{1-\frac{q}{p}} \tag{5.3}$$

Finally, the number d of deletion steps *from* k is characterized by (letting $z = w = 1$)

$$F_d(v) = \frac{p-q}{p-qv} = \frac{1-\frac{q}{p}}{1-v\frac{q}{p}}, \tag{5.4}$$

which shows that d is a geometric r.v. with mean

$$E(d) = \frac{\frac{q}{p}}{1-\frac{q}{p}}. \tag{5.5}$$

5.1. Example: Classical priority queues. For priority queues, we have for the average size at t

$$y(t) = \frac{1}{2}t(2-t),$$

and the covariance of X is given by $C(s,t) = s^2(2-t)^2/4$. Thus $\overline{t} = 1-\sqrt{1-2k/n}$, and $y'(\overline{t}) = \sqrt{1-2k/n}$. Now adapting the proof of Lemma 13 in [41], we can prove that

$$p = 1-\frac{\overline{t}}{2}+O\left(\frac{1}{n}\right), \qquad q = \frac{\overline{t}}{2}+O\left(\frac{1}{n}\right), \qquad r = 0.$$

Then we find that

$$E(i) = E(d) = \frac{1}{\sqrt{1-2\frac{k}{n}}},$$

$$\tilde{E}_\ell = 1+E(\ell) = \frac{2}{\sqrt{1-2\frac{k}{n}}}.$$

6. Hashing with lazy deletion. Hashing with lazy deletion was introduced in [53] as a data structure suitable for example for line-sweep algorithms on a set of segments of the plane. When a new item arrives (i.e., the line reaches the extremity of a new segment), it is inserted in a random bucket of a separate hash table, along with the x-coordinate of its right extremity. When an item "dies," i.e., the line goes past the right extremity of the segment, it is not removed from the data structure right away but only at the time of a later insertion within the same bucket. This method presents the advantage of minimizing the number of accesses to the hash table, at the cost of some extra space used. In [53], several models of distribution are suggested to analyze the extra space: two nonstationary models (which are quite similar to one another so that we will only study the first one since we know that the same approach works also for the other model) and one stationary model.

6.1. The first nonstationary model. In the first nonstationary model, there are n segments in $[0\ldots1]$ drawn independently from the following distribution: segment $s = [\min(x,y), \mathrm{Max}(x,y)]$, where x and y are uniform independent r.v.'s in $[0\ldots1]$. Thus the arrival-time density is $2n(1-u)du$ and the lifetime z, conditional on u, has density $ndz/(1-u)$. To remove the conditionality, it is convenient to change the time scale. Let the new time t be given by $1-u = e^{-2t}$. It is easy to see that the arrival times now have density $2e^{-2t}dt$, and the service times have unconditional density $e^{-t}dt$.

Two parameters are of interest here: Need(t), the number of items alive at time t (which have to be present in the data structure); and Use(t), the number of items which are actually present in the table at time t. Thus Waste(t) = Use(t) − Need(t) counts the items that are dead but not yet deleted at time t.

6.1.1. Study of Max_tNeed(t). We note that Need(t) depends on n but not on H. Our calculations are asymptotic when n, the total number of items, goes to infinity. From Louchard [43, Theorem 1], we find that as n grows, Need(t) converges to a Gaussian process:

$$\frac{\mathrm{Need}(t) - nz_{\mathrm{Need}}(t)}{\sqrt{n}} \Rightarrow Q(t),$$

where $z_{\mathrm{Need}(t)}$, the probability that a given item is alive at time t, is defined by

$$z_{\mathrm{Need}}(t) = \int_0^t 2e^{-2u}e^{-(t-u)}du = 2e^{-t}(1-e^{-t}).$$

$Q(t)$ is a Gaussian process with mean 0 and covariance

$$C(s,t) = 2e^{-t}(1-e^{-s}) - 4e^{-t}(1-e^{-t})e^{-s}(1-e^{-s})$$

if $s \le t$. It is easy to check that $z_{\mathrm{Need}}(t)$ is maximized at $\bar{t} = \ln 2$, where its value is $1/2$. Thus we can rewrite the convergence as

$$\mathrm{Need}(t) \sim \sqrt{n}\left[\frac{\sqrt{n}}{2} + Q(t) + \sqrt{n}\left(z_{\mathrm{Need}}(t) - \frac{1}{2}\right)\right],$$

and, applying Daniels's theorem, we obtain the following result after some algebra.

THEOREM 6.1.

$$\mathrm{Max}_t\mathrm{Need}(t) \sim \frac{n}{2} + \sqrt{n}M + O(n^{1/6}),$$

where M is characterized by the Theorem 3.1 *in section* 3.1, *with $A = 1$, $B = 1$, $C(\bar{t},\bar{t}) = 1/4$, $[\partial_s C]_{\bar{t}} = 1/2$ and $[\partial_t C]_{\bar{t}} = -1/2$.*

6.1.2. Study of Max_tUse(t). Use(t) depends heavily on the number of buckets. The larger H is compared to n, the more discrepancy there is between Max_tNeed(t) and Max_tUse(t). We assume that the number H of buckets also goes to infinity, and there are two cases: either $H = rn$, with r fixed, or $H = \alpha(n)n$, with $\alpha(n) \to 0$.

We must now study $M = \mathrm{Max}_t\mathrm{Use}(t)$. Let Waste$(k)$ be the time during which customer k was dead but not yet deleted. The distribution of Waste(k) is known:

$$\Pr\{\mathrm{Waste}_k > t - y\} = \left[1 - \frac{e^{-2t} - e^{-2y}}{H}\right]^{n-1}.$$

If $H = rn$, this gives

$$Pr[\text{Waste}_k \geq t - y] \sim e^{-p(t,y)/r}\left[1 + O\left(\frac{1}{n}\right)\right], \tag{6.1}$$

where $p(t, y) = e^{-2t} - e^{-2y}$.

Modifying the proof of Theorem 1 in [43] slightly (the error term from (6.1) leads to $O(1/n^{3/2})$, similar to the other error terms of our proof), we see that

$$\frac{\text{Use}(t) - nz(t)}{\sqrt{n}} \Rightarrow Q(t),$$

where $Q(t)$ is a Gaussian process with mean 0 and $nz(t)$ is the average number of items in use at time t: $z(t)$ is the probability that a fixed item is in use at time t. The item is alive at time t if it arrived at $u < t$ and lived for more than $t - u$; it is dead but not deleted at time t if it arrived at time u, died at time $y < t$, and was in waste for longer than $t - y$. Thus approximating Waste_k, we get

$$z(t) = \int_0^t 2e^{-2u}e^{-(t-u)}du + \int_0^t 2e^{-2u}du \int_u^t e^{-(y-u)}e^{-(e^{-2y}-e^{-2t})/r}dy.$$

The limit process $Q(t)$ has mean 0, and we must calculate its covariance. Let $\eta_i(s)$ be the random variable, which is 1 if customer i is in use at time s and 0 otherwise. Then the number $\text{Use}(s)$ of customers in use at time s satisfies

$$\text{Use}(s) = \sum_{1 \leq i \leq n} \eta_i(s).$$

Thus the covariance is given by

$$\text{Cov}(\text{Use}(s)\text{Use}(t)) = \sum_{1 \leq i \leq n} E(\eta_i(s)\eta_i(t)) + 2\sum_{i<j} E(\eta_i(s)\eta_j(t)) - n^2 z(s)z(t)$$

for $s \leq t$. The first part of the sum can be expressed easily as a sum of integrals. The second term depends on whether or not i and j are in the same bucket (events which have probabilities $1/H$ and $1 - 1/H$). After careful development in $1/n$, we obtain

$$\begin{aligned}
&\text{Cov}(\text{Use}(s)\text{Use}(t)) \\
&= n\left[2e^{-t}(1 - e^{-s}) + \int_0^s 2e^{-2u}du \int_u^t e^{-(y-u)}e^{-\frac{e^{-2y}-e^{-2t}}{r}}dy\right. \\
&+ \eta(s)z(t) + \eta(t)z(s) - \eta(s)\eta(t) \\
&+ \frac{1}{r}\int_0^s 2e^{-2u}du \int_0^t 2e^{-2v}dv \int_u^s e^{-(x-u)}dx \int_v^t e^{-(y-v)}dy e^{-A([xs]\cup[yt])/r} 1_{[u\notin\{y,t\}]\cap[v\notin\{x,s\}]} \\
&- \frac{1}{r}\int_0^s 2e^{-2u}du \int_u^s e^{-(y-u)}e^{-(e^{-2y}-e^{-2s})/r}dy \int_0^t 2e^{-2u}du \int_u^t e^{-(y-u)}e^{-(e^{-2y}-e^{-2t})/r}dy \\
&\left. - z(s)z(t)\right] + O(1),
\end{aligned}$$

where $z(t)$ is given as above, $\eta(s)$ is defined by

$$\eta(s) = \int_0^s 2e^{-2u}du \int_u^s e^{-(x-u)}e^{-(e^{-2x}-e^{-2s})/r}\frac{e^{-2x} - e^{-2s}}{r}dx,$$

and $A([xs] \cup [yt])$ is the measure of the union of intervals $[xs]$ and $[yt]$ in the distribution $A(x) = 1 - e^{-2x}$.

To apply Daniels's theorem, we must find $\overline{t}$ such that $z'(\overline{t}) = 0$. From [53], it is known that $\overline{t}$ exists only if $r < 0.84$.

If $r < 0.84$, we can apply Daniels's theorem. If $r \geq 0.84$, we know from Iglehart [30] that $\mathrm{Max}_{s \leq t}\mathrm{Use}(s)$ is asymptotically distributed as $\mathrm{Use}(t)$. Letting t go to infinity gives an equivalent of the maximum.

Finally, we have the following result.

THEOREM 6.2. *Assume that $H = rn$.*

1. *If $r \geq 0.84$, then*

$$\mathrm{Max}_t\mathrm{Use}(t) \sim Cn + \sqrt{n}\sqrt{C(1-C) + C'}X,$$

where $X = \mathcal{N}(0,1)$ is the classical Gaussian random variable, C is defined to be $2\int_0^\infty e^{-x}(1-e^{-x})e^{-e^{-2x}/r}dx$, and C' (which can be evaluated numerically) is given by

$$\begin{aligned} C' = {} & 2\eta(\infty)z(\infty) - \eta^2(\infty) \\ & + \frac{1}{r}\int_0^\infty 2e^{-2u}du \int_0^\infty 2e^{-2v}dv \int_u^\infty e^{-(x-u)}dx \\ & \qquad \int_v^\infty e^{-(y-v)}dy e^{-A([xs]\cup[yt])/r} I[[u \notin \{y,t\}] \cap [v \notin \{x,s\}]] \\ & - \frac{1}{r}\left[2\int_0^\infty e^{-x}(1-e^{-x})e^{-e^{-2x}/r}dx\right]^2. \end{aligned}$$

2. *If $r < 0.84$, then*

$$\mathrm{Max}_t\mathrm{Use}(t) \sim n[2e^{-\overline{t}}(1-e^{-\overline{t}}) + r] + \sqrt{n}M + O(n^{-1/6}),$$

where $\overline{t}$ is the solution of

$$2\int_0^{\overline{t}} e^{-x}(1-e^{-x})e^{-(e^{-2x}-e^{-2\overline{t}})/r}dx = r$$

and M is given by Daniels's theorem, with $B = [4e^{-3\overline{t}}(1-2e^{-\overline{t}})/r]^{-1/3}$, and $C(\overline{t},\overline{t})$, $[\partial_s C]_{\overline{t}}$ and $[\partial_t C]_{\overline{t}}$ can be written in terms of complicated multiple integrals.

In the case where $H = \alpha(n)n$ with $\alpha(n) \to 0$, the arrival times in a bucket behave in first approximation like a Poisson process by classical sample distribution analysis. Our goal here is again to apply Daniels's theorem. At each step of the calculation, we write the quantities as power series in α so that we can neglect high powers of α, and the expressions remain simple enough. Apart from some technical complications, the approach is the same as for the case where $H = rn$. Our calculations are accurate if $\alpha(n) = o(1/\sqrt{n})$. If $\alpha(n)$ decreased more slowly than that, it would still be possible to get some results, but that would require writing the power series more accurately.

THEOREM 6.3. *Let $H = \alpha(n)n$, with $\alpha(n) = o(1/\sqrt{n})$. Then we have*

$$\mathrm{Max}_t\mathrm{Use}(t) \sim n\left(\frac{1}{2} + \alpha - 2\alpha^2 + O(\alpha^3)\right) + \sqrt{n}M + O(n^{-1/6}),$$

where M, given by Daniels's theorem, is asymptotically Gaussian, with mean

$$E(M) = 0.99615n^{-1/6}2^{2/3} + O(n^{-1/3}).$$

We have $A = 2 + O(\alpha)$, $B = 1 + O(\alpha)$, and

$$C(\bar{t}, \bar{t}) = \frac{1}{4} + 2\alpha + O(\alpha^2),$$

$$[\partial_s C]_{\bar{t}} = 1 + O(\alpha),$$

$$[\partial_t C]_{\bar{t}} = -1 + O(\alpha).$$

Sketch of proof. The local rate of the arrival times Poisson process is given by $2ne^{-2t}/H = 2e^{-2t}/\alpha(n)$. We must compute $\bar{t}$ such that $\tilde{z}'(\bar{t}) = 0$; this gives (we omit the details) $\bar{t} = \log 2 + 2\alpha - 2\alpha^2 + O(\alpha^3)$ and $\tilde{z}(\bar{t}) = 1/2 + \alpha - 2\alpha^2 + O(\alpha^3)$, $\tilde{z}''(\bar{t}) = -1 + 8\alpha + O(\alpha^2)$. From this point, the proof is the same as for the case where $H = rn$.

6.2. The second nonstationary model. Van Wyk and Vitter [53] suggested another nonstationary model, in which arrivals and deaths are not symmetric. Here again n segments in $[0 \dots 1]$ are drawn independently from a given distribution: segment $[x \dots y]$ is constructed by drawing x uniformly in $[0 \dots 1]$ and then drawing y uniformly in $[x \dots 1]$. Up to a change of scale, we can assume that the densities of the arrival and service times are $e^{-t}dt$. Henceforth, the proof mirrors that of the previous subsection. The only difference in the calculations is that the arrival times have density $e^{-t}dt$ instead of $2e^{-2t}dt$ so that there is no critical value like 0.84; there is always a point at which $z(t)$ is maximized.

THEOREM 6.4.

$$\mathrm{Max}_t\mathrm{Need}(t) \sim \frac{n}{e} + \sqrt{n}M + O(n^{1/6}),$$

where M is characterized by Daniels's theorem, with $A = 2/e$, $B = (1/e)^{-1/3}$, $C(\bar{t}, \bar{t}) = (1 - 1/e)/e$, $[\partial_s C]_{\bar{t}} = 1/e$, and $[\partial_t C]_{\bar{t}} = -1/e$.

THEOREM 6.5. *If $H = \alpha(n)n$, with $\alpha(n) = o(1/\sqrt{n})$, then we have*

$$\mathrm{Max}_t\mathrm{Use}(t) \sim n\left(\frac{1}{e} + \alpha + O(\alpha^2)\right) + \sqrt{n}M + O(n^{1/6}),$$

where M is characterized by Daniels's theorem, with $A = 4/e + O(\alpha)$, $B = (1/e)^{-1/3} + O(\alpha)$,

$$C(\bar{t}, \bar{t}) = \frac{1}{e}\left(1 - \frac{1}{e}\right) + 2\alpha + O(\alpha^2),$$

$[\partial_s C]_{\bar{t}} = 2/e + O(\alpha)$, *and* $[\partial_t C]_{\bar{t}} = -2/e + O(\alpha)$.

7. Conclusion. Diffusion techniques allowed us to derive several new results on data structures' maxima. Many problems remain open and are the object of work in progress. Let us mention the symbol table (the probabilistic properties of which are unknown), the $G/G/\infty$ case, with $\ln(\lambda t) = \Omega(\lambda)$. Let us mention that in a recent report [3], Aldous, Hofri, and Szpankowski analyzed hashing with lazy deletion in the stationary case and proved open conjectures introduced in [45] and [53].

Appendix A. A direct computation of Daniels's formula. Daniels's results [13], [14] are based on the following hitting-time density for a Brownian motion $X(t)$.

Let $w(t) = m + \sqrt{n}\, f(t)$, with $f(t_o) = f'(t_o) = 0$. Daniels obtained the density (see Daniels and Skyrme [13])

$$g(t)dt = Pr[\min(t' : X(t') = w(t')) \in dt] = \frac{e^{-\frac{[w(t)]^2}{2t}}}{\sqrt{2\pi t}}\mu(t)\, dt, \tag{A.1}$$

where

$$\mu(t) = n^{-1/3}(2\beta)^{1/3}F\{n^{1/3}[2\beta]^{2/3}(\bar{t} - t)\}(1 + O(n^{-1/3}))$$

with F given by (4.1), $\beta := f''(\bar{t})/2$, and $\bar{t}$ is such that $h'(\bar{t}) = h(\bar{t})/\bar{t}$.

1. We will first extract all dominant terms from (A.1). As in Daniels and Skyrme [13], set

$$x = n^{1/3}(2\beta)^{2/3}(t_0 - t) = O(1).$$

They computed

$$\bar{t} - t_o = n^{-1/2}\frac{m}{t_0 2\beta} + O(n^{-2/3}),$$

so

$$\bar{t} - t = \frac{xn^{-1/3}}{(2\beta)^{2/3}} + \frac{n^{-1/2}m}{2\beta t_0} + O(n^{-2/3}).$$

Expanding $G(x)$ as given in (4.1), we obtain

$$G(x) = \frac{1}{2^{2/3}}\sum_{k=o}^{\infty}\frac{e^{-\lambda_k x/2^{1/3}}}{A_i'(\lambda_k)},$$

where λ_k are the zeroes of $A_i(x)$. A detailed expansion of (A.1) now leads to

$$g(t) = \frac{(2\beta)^{1/3}}{\sqrt{2\pi t_o}\, 2^{2/3}}\exp\left[\frac{-m^2}{2t_0} + \frac{x^3}{6}\right]\sum_{k=o}^{\infty}\frac{\exp[\frac{-\lambda_k}{2^{1/3}}(x + \frac{m}{t_o(2\beta)^{1/3}n^{1/6}})]}{A_i'(\lambda_k)}(1 + O(n^{-1/3})). \tag{A.2}$$

2. Such a simple formula as (A.2) should be explained in terms of direct hitting-time density for a Brownian motion. This will be done with a technique introduced by Salminen [50]. Let us first remark that to derive (A.2), it is enough to limit ourselves to a second-order boundary (the error is within $O(n^{-1/3})$), so we can simply use

$$w(t) = m + \sqrt{n}\beta(t - t_o)^2$$

with $\bar{x} = w(0) = m + \beta\sqrt{n}\, t_o^2$. Our hitting problem can now be transformed into

$$g(t)dt = Pr_{\bar{x}}[\min(t' : X(t') = h(t')) \in dt]$$

with $h(t) = -\sqrt{n}\,\beta(t - t_o)^2 + \sqrt{n}\,\beta t_o^2$ such that $h(0) = 0$. This also gives $h'(t) = -2\sqrt{n}\,\beta(t - t_o); h'' = -2\beta\sqrt{n}$.

By Salminen [50, Equations (2.6) and (3.9)], we now obtain

(A.3)

$$g(t) = 2\left(\frac{\beta\sqrt{n}}{2}\right)^{2/3} \exp\left[h'(0)\bar{x} - \frac{1}{2}\int_o^t [h'(s)]^2 ds\right] \sum_{k=o}^{\infty} e^{\lambda_k (2\beta^2 n)^{1/3} t}\, \frac{A_i[\lambda_k + 2(\frac{\beta\sqrt{n}}{2})^{1/3}\bar{x}]}{A'(\lambda_k)}.$$

However, $h'(0) = 2\sqrt{n}\,\beta t_o$ and

$$-\frac{1}{2}\int_o^t [h'(s)]^2 ds = -\int_o^t 2n\beta^2 (s - t_o)^2 ds$$
$$= -2n\beta^2\left[\frac{(t - t_o)^3}{3} + \frac{t_0^3}{3}\right] = \frac{x^3}{6} - \frac{2n\beta^2 t_0^3}{3}.$$

By Abramowitz and Stegun [1, Equation (10.4.59)] we know that for large z,

$$A_i(z) \sim \frac{1}{2}\pi^{-1/2} z^{-1/4} e^{-\xi}\left(1 + O\left(\frac{1}{\xi}\right)\right)$$

with $\xi := \frac{2}{3} z^{3/2}$.

From (A.3), we must use

$$z = \lambda_k + 2^{2/3} n^{2/3} \beta^{4/3} t_o^2 + 2^{2/3}\beta^{1/3} m n^{1/6}$$

so that

$$\xi = \frac{4}{3} n\beta^2 t_o^3\left[1 + \frac{3}{2}\,\frac{m}{\beta t_o^2\sqrt{n}} + \frac{3}{8}\,\frac{m^2}{\beta^2 t_o^4 n} + \frac{3}{2}\,\frac{\lambda_k}{2^{2/3} n^{2/3} \beta^{4/3} t_o^2}\right.$$
$$\left. + \frac{3}{4}\,\frac{m\lambda_k}{\beta^{7/3} t_o^4 2^{2/3} n^{7/6}}\right]\left(1 + O\left(\frac{1}{n^{1/3}}\right)\right).$$

Also, we deduce

$$\lambda_k (2\beta^2 n)^{1/3} t = \lambda_k (2\beta^2 n)^{1/3} t_o - \frac{\lambda_k x}{2^{1/3}},$$

$$h'(0)\bar{x} = 2\beta\sqrt{n}\, t_o \bar{x} = 2\beta\sqrt{n}\, m t_o + 2n\beta^2 t_o^3.$$

The identification of (A.3) with (A.2) is now routine.

Appendix B. Rate of convergence for $G/G/\infty$. We prove that for $G/G/\infty$ the asymptotic stationary distribution of $Y_n(t) := [Q_n(t) - n]/\sqrt{n}$ has a Gaussian tail $(1 - N(x))$ if $x^3 = o(\sqrt{n})$. We also analyze the correction term.

1. Let the renewal process associated with arrivals be $A_n(t)$ (related parameters are indexed by $A, \lambda_A = 1$); we count the origin as a renewal. Then it is well known that the process

$$T_n^*(t) := \frac{A_n(t) - nt}{\sqrt{n}}$$

is asymptotically equivalent to a Brownian motion (with variance $\sigma_A^2 t$). However, we need some rate of convergence. We will detail only the $1/\sqrt{n}$ correction to the distribution, but all terms could be obtained by the same method. Thus we write $T_n^*(t)$ as

$$\frac{\left[\sum_{\ell=1}^{A_n(t)} (1-u_{A,\ell}) + Z_n(t)\right]}{\sqrt{n}},$$

where we set $Z_n(t) := \sum_{j=1}^{A_n(t)} u_{A,j} - nt$ (i.e., the residual waiting time).

By Feller [16, Chapter XI, Equation 4.10], we know that for large n, $Z_n(t)$ has a stationary distribution with mean $(1+\sigma_A^2)/2$.

We must compute $E[e^{-\theta T_n^*(t)}]$. Now

$$\Pi(x) := \Pr[A_n(t) \le r] = \Pr[T^r \ge nt]$$

when T^r is the total time for r arrivals and $r = nt + x\sqrt{n}$. Actually, the discrete character of the process entails some correction. On first order, it is enough to set $r = nt + x\sqrt{n} - 1/2$. (We omit the details; this is similar to Euler summation formula first correction.) We immediately obtain

$$\int_0^\infty e^{-\theta x} \int_0^\infty e^{-wnt} \Pi(x) dx d(nt) = \int_0^\infty e^{-\theta x} \frac{[1 - e^{r[\varphi_1(-w)-w]}]}{w} dx, \tag{B.1}$$

where $\varphi_1(s) := (1/2)s^2\sigma_A^2 + (s^3/6)\mu_{3,A} + O(s^4)$.

(B.1) leads to

$$\frac{1}{\theta w} - \frac{e^{[nt-1/2][\varphi_1(-w)-w]}}{w[\theta - \sqrt{n}(\varphi_1(-w)+w)]}.$$

Thus the asymptotic Laplace transform $\varphi_2(\theta) = \int_0^\infty e^{-\theta x}\Pi(dx)$ is given by

$$\varphi_2(\theta) = \mathcal{R}\frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty} \left[\frac{1}{w} + \frac{\theta e^{(nt-1/2)[\varphi_1(-w)-w]}}{w[\sqrt{n}(\varphi_1(-w)+w) - \theta]}\right] \tag{B.2}$$

with c to the right of integrand singularities.

The first term leads to 1. By classical residue analysis, the coefficient of $(-\theta)$ is given by $m_1 \sim (\sigma_A^2+1)/2\sqrt{n}$, we recover $Z_n(t)$ mean (see also Feller [16, Chapter XI, Equation 3.1]). The coefficient of $\theta^2/2$ is $\sim \sigma_A^2 t$. The third centered moment is $\sim \tilde{\mu}_3 t/\sqrt{n}$ with $\tilde{\mu}_3 = m_3 - 3\sigma_A^2 m_1 + 2m_1^3$ and $m_3 t/\sqrt{n}$ is the coefficient of $-\theta^3/6$ in (B.2). After computation, we obtain $\tilde{\mu}_3 \sim [3\sigma_A^4 - \mu_{3,A}]$. Thus we derive

$$\varphi_2(\theta) \sim \exp\left[-\theta m_1 + \frac{\theta^2\sigma_A^2 t}{2} - \frac{\theta^3 \tilde{\mu}_3 t}{(6\sqrt{n})} + O\left(\frac{\theta^4}{n}\right)\right]. \tag{B.3}$$

2. Let us turn to

$$Y_n(t) := \frac{Q_n(t) - \bar{Q}_n(t)}{\sqrt{n}}.$$

We will analyze the stationary distribution, but k-dimensional distributions can be treated similarly by tedious but routine algebra.

Each customer arriving at time s induces at time t a survival probability $1-B(t-s)$. Set $\gamma := 1-B$ ($\gamma(0)=1$). Thus the survival process has mean $\mu_{1,S} := \gamma$, variance $\mu_{2,S} := \gamma(1-\gamma)$, and third (centered) moment $\mu_{3,S} := \gamma(1-\gamma)(1-2\gamma)$ (all functions with argument $t-s$). Then we can write

(B.4)

$$E[e^{iwY_n(t)/\sqrt{n}}] = E\left\{\exp\left[\int_0^t \left[\frac{iw\mu_{1,S}(t-s)}{\sqrt{n}} - \frac{w^2}{2n}\mu_{2,S}(t-s) + \frac{(iw)^3}{6n^{3/2}}\mu_{3,S}(t-s) + O\left(\frac{w^4}{n}\right)\right] dA(s) - iw\sqrt{n}\int_0^t \mu_1(t-s)ds\right]\right\},$$

and we replace $dA(s)$ by $nds + \sqrt{n}dT_n^*(s)$.

Now using (B.5) in (B.3), we finally obtain for large t

$$\begin{aligned} E[e^{iwY_n(t)/\sqrt{n}}] \sim \exp\Bigg\{ & \frac{-w^2}{2}\int_0^\infty \mu_{2,S}(t-s)ds + \frac{(iw)^3}{6\sqrt{n}}\int_0^\infty \mu_{3,S}(t-s)ds \\ & - \frac{\sigma_A^2}{2}\int_0^\infty \left[w\mu_{1,S}(t-s) - \frac{w^2}{2i\sqrt{n}}\mu_{2,S}(t-s)\right]^2 ds \\ & + \frac{i^3\tilde{\mu}_3}{6\sqrt{n}}\int_0^\infty [w\mu_{1,S}(t-s)]^3 ds + O\left(\frac{w^4}{n}\right)\Bigg\} \cdot \\ & \cdot E\left[\exp\left[\frac{i}{\sqrt{n}}\int_0^\infty w\mu_1(t-s)dZ(t-s)\right]\right]. \end{aligned}$$

Of course, we rediscover the classical asymptotic variance of $Y_n(t)$, $t\to\infty$,

$$\sigma_Y^2 = \int_0^\infty \mu_{2,S}(t-s)ds + \int_0^\infty \sigma_A^2\mu_{1,S}^2(t-s)ds.$$

The third centered moment of $Y_n(t)$ is given by $\mu_{3,Y}/\sqrt{n}$, with

$$\mu_{3,Y} := \left[\int_0^\infty [\mu_{3,S}(t-s) + 3\sigma_A^2\mu_{1,S}(t-s)\mu_{2,S}(t-s) + \tilde{\mu}_3\mu_{1,S}^3(t-s)]ds\right].$$

The mean $\sqrt{n}m_Q (m_Q = \int_0^\infty \mu_{1,S}(t-s)ds = 1/\mu_S)$ of $Q_n(t)/\sqrt{n}$ is now affected by an $O(1/\sqrt{n})$ term that we can obtain as follows:

$$E\left[\int_0^\infty \mu_{1,S}(t-s)dZ(t-s)\right] = E\left[Z(t) - \int_0^\infty f(t-s)Z(t-s)\right]. \tag{B.5}$$

Returning to the original scale, we set $s = v/n$; the last term of (B.5) gives

$$E\left[\int_0^\infty f\left(t-\frac{v}{n}\right) Z\left(t-\frac{v}{n}\right)\frac{dv}{n}\right],$$

and by the ergodic theorem, this gives $(1/2)E(u_A) = 1/2$. Also, from [16, Chapter XI, Equation 4.10], $E[Z(t)] \sim (1+\sigma_A^2)/2$. The final $O(1/\sqrt{n})$ correction to $\sqrt{n}m_Q$ is given by

$$\frac{(1+\sigma_A^2/2)}{\sqrt{n}}.$$

3. We are now ready for a large-deviation analysis of the tail of the $Y_n(t)$ distribution. We will only consider the situation where $Y_n(t)$ is large but $o(\sqrt{n})$. Since we have the first moments of $Y_n(t)$, we can write a Thiele expansion of $E[-\xi Q_n(t)/\sqrt{n}](\xi > 0)$ (the r.v. is positive); this gives $e^{-\xi\sqrt{n}m_Y}e^{\varphi_3(-\xi)}$, with

$$\varphi_3(s) := \frac{1}{2}s^2\sigma_Y^2 + \frac{s^3}{6}\,\frac{\mu_{3,Y}}{\sqrt{n}} + O\left(\frac{s^4}{n}\right) = n\Psi\left(\frac{s}{\sqrt{n}}\right) \quad \text{say,}$$

with

$$\Psi(s) := \frac{1}{2}s^2\sigma_Y^2 + \frac{s^3}{6}\mu_{3,Y} + O(s^4).$$

We will use Feller's [16, Chapter XVI] notation, but we cannot proceed as in [16, Equation 7.6] because we do not deal with a sum of n r.v.'s. Nevertheless, a saddle-point method will provide an equivalent. Let $\tilde{F}$ be the d.f. of $Q_n(t)/\sqrt{n}$. We are interested in

$$1 - \tilde{F}(x\sigma_Y + \sqrt{n}m_Q) = \mathcal{R}\left(\frac{1}{2\pi i}\int_{-i\infty}^{+i\infty}[1 - e^{-\sqrt{n}m_Q\xi + \varphi_3(-\xi)}]e^{\xi[\sqrt{n}m_Q + \sigma_Y x]}\,\frac{d\xi}{\xi}\right).$$

Set $\xi = -s\sqrt{n}$. We obtain

$$\mathcal{R}\left[\frac{1}{2\pi i}\int_{+i\infty}^{-i\infty}[e^{-nm_Q s} - e^{n\Psi(s)}]e^{-s\sigma_Y x\sqrt{n}}\,\frac{ds}{s}\right]. \tag{B.6}$$

We will soon see that our saddle point s^* is positive and of order $x/\sqrt{n}$.

We write $\log[-e^{-nm_Q s} + e^{n\Psi(s)}]$ as

$$n\Psi(s) + \log[1 - e^{-n[sm_Q + \Psi(s)]}].$$

The saddle point s^* of (B.6) is solution of

$$n\Psi'(s^*) - \sqrt{n}x\sigma_Y + \frac{n[m_Q + \Psi'(s^*)]e^{-n[s^*m_Q + \Psi(s^*)]}}{1 - e^{-n[s^*m_Q + \Psi(s^*)]}}. \tag{B.7}$$

After a detailed analysis, the solution of (B.7) is checked to be the solution of $n\Psi'(s^*) - \sqrt{n}x\sigma_Y$ up to exponential small terms $O(e^{-\sqrt{n}x})$, which is exactly equation (7.8) of [16, Chapter XVI].

By a standard saddle-point analysis, (B.6) finally leads to

$$\frac{e^{[n\Psi(s^*) - \sqrt{n}s^*\sigma_Y x]}}{\sqrt{2\pi}s^*\sqrt{n\Psi''(s^*)}},$$

which is exactly [16, Equation (7.11)]. (Use the classical Gaussian tail.) Thus [16, Theorem 2] is still applicable. If $x^3 = O(\sqrt{n})$, we can compute a first correction to the Gaussian tail depending on $\mu_{3,Y}$; we obtain

$$1 - \tilde{F}(x\sigma_Y + \sqrt{n}m_Q) \sim [1 - N(x)]\exp\left(\frac{\mu_{3,Y}x^3}{6\sqrt{n}\sigma_Y^3}\right).$$

Appendix C. Sojourns and extremes of stationary processes. The content of this section is from Berman [6].

THEOREM C.1 (Sojourn limit theorem; [6, Theorem 3.1]). *Let $X(t)$, $-\infty < t < +\infty$, be separable, measurable, and stationary. Suppose that the following two conditions hold:*

(i) *There exists a measurable process $Z(t)$, $-\infty < t < +\infty$, with continuous finite-dimensional distributions and functions $v = v(u)$ and $w = w(u)$, $u > 0$, with $\lim_{u\to\infty} v(u) = \infty$ such that the finite-dimensional distributions of the process*

$$w(u)\left(X\left(\frac{t}{u}\right) - u\right), \quad -\infty < t < +\infty, \tag{C.1}$$

conditioned by $X(0) > u$ converge to those of $Z(t)$.

(ii) *The function v satisfies*

$$\lim_{d\to\infty} \limsup_{u\to\infty} v \int_{d/v}^{t} P(X(s) > u \mid X(0) > u)ds = 0$$

for $0 < t \le T$ for some $T > 0$. Define

$$\Gamma(x) = P\left(\int_0^\infty 1_{[Z(s)>0]}ds > x\right), \quad x > 0.$$

Then

$$\lim_{u\to\infty} \int_x^\infty \frac{P(vL_t(u) > y)}{E(vL_t(u))}dy = \Gamma(x)$$

at all continuity points $x > 0$ of Γ for $0 < t \le T$.

If

$$\lim_{u\to\infty} \frac{1 - F(u + \frac{x}{w})}{1 - F(u)} = e^{-x}, \quad -\infty < x < \infty,$$

the function w is then necessarily of the form

$$w(u) = \frac{1 - F(u)}{\int_u^\infty (1 - F(x))dx}.$$

If $f(x) = F'(x)$ exists and is nonincreasing for all sufficiently large x, then w may be taken as

$$w(u) = \frac{f(u)}{1 - F(u)}.$$

THEOREM C.2 ([6, Theorem 5.2]). *Let $X(t)$ be a stationary process with the marginal distribution function F satisfying conditions $\lim(1-F(u+\frac{x}{w}))/(1-F(u)) = e^{-x}$; and $w(u) = f(u)/(1 - F(u))$. Consider the process*

$$w \cdot \left(X\left(\frac{t}{v}\right) - X(0)\right), \quad -\infty < t < +\infty, \tag{C.2}$$

conditioned by

$$X(0) = u + \frac{y}{w},$$

Suppose that there is a measurable function $\mu(t,x)$ of (t,x) such that the following hold:

(i) *There exists a process $V(t)$, $-\infty < t < +\infty$, with continuous finite-dimensional distributions such that the finite-dimensional distributions of the process*

$$w \cdot \left[X\left(\frac{t}{v}\right) - \mu\left(\frac{t}{v}; u + \frac{y}{w}\right)\right], \quad -\infty < t < +\infty,$$

suitably conditioned, converge to those of $V(t)$ for any fixed real y.

(ii) *There is a function $m(t)$ not depending on y such that*

$$\lim_{u\to\infty} w \cdot \left[\mu\left(\frac{t}{v}, u + \frac{y}{w}\right) - \left(u + \frac{y}{w}\right)\right] = m(t) \quad \textit{for } -\infty < t, y < \infty.$$

Then the finite-dimensional distributions of process (C.2)*, conditioned by $X(0) = u + \frac{y}{w}$, converge to those of the process $V(t) + m(t)$, and so $Z(t)$ in the sojourn limit theorem is of the form*

$$Z(t) = V(t) + m(t) + \eta,$$

where η is a random variable with an exponential distribution, and is independent of the process V.

Let $X(t)$, $-\infty < t < \infty$, be a stationary Gaussian process with mean 0, variance 1, and continuous covariance function $r(t)$. Here F is the standard normal distribution function Φ and $\phi(x)$ is the density function. Φ is in the domain of attraction of the extreme-value distribution function Λ, and according to (4.3) and the well-known asymptotic formula for the tail of the normal distribution, we have

$$w(u) = \frac{\phi(u)}{1 - \Phi(u)} \sim u \quad \text{for } u \to \infty.$$

It follows that the sojourn time theorem is valid for stationary Gaussian processes of a very general type on an interval $[0,t]$, where t is sufficiently small.

We will assume that $1 - r(t)$ is regularly varying of index α for some $0 < \alpha \leq 2$ for $t \to 0$. Define $v = v(u)$ as the largest solution of the equation $u^2(1 - r(1/v)) = 1$; then it follows from the regularity property that

$$u^2\left(1 - r\left(\frac{t}{v}\right)\right) \to t^\alpha \quad \text{for } u \to \infty.$$

Under some regularity conditions (which are satisfied by our processes $X(t)$), the maximum limit theorem gives

$$\lim_{u\to\infty} \frac{P(\max_{[0,t]} X(s) > u)}{vt(1 - F(u))} = -\Gamma'(0)$$

For general stationary processes (not necessarily Gaussian), we take $u(t)$ as the solution of

$$v(u)t(1 - F(u)) = 1.$$

THEOREM C.3 ([6, Theorem 19.1]). *For all sufficiently large t, let $u = u(t)$ and $v = v(t)$ be defined as above, and let $w = w(u(t))$ be defined as before and satisfy*

$$\lim_{u\to\infty} uw(u) = \infty$$

and

$$\lim_{u\to\infty, u'\to\infty, u/u'\to 1} \left(\frac{w(u)}{w(u')}\right) = 1.$$

Let v satisfy $v(u')/v(u) \to 1$, $u \to \infty$. Then under some regularity conditions (which are satisfied by our processes $X(t)$)

$$\lim_{t\to\infty} P\left\{w\left(\max_{[0,t]} X(s) - u\right) \le x\right\} = \exp(\Gamma'(0)e^{-x}),$$

for every x.

Acknowledgments. We thank Ph. Flajolet for helpful discussions on this topic and the anonymous referees for pertinent comments. Maple was of great help for computing complicated expressions.

REFERENCES

[1] M. ABRAMOWITZ AND I. A. STEGUN, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, Dover, New York, 1965.
[2] D. ALDOUS, *Probability Approximations via the Poisson Clumping Heuristic*, Springer-Verlag, Berlin, 1989.
[3] D. ALDOUS, M. HOFRI, AND W. SZPANKOWSKI, *Maximum size of a dynamic data structure: Hashing with lazy deletion revisited*, SIAM J. Comput., 21 (1992), pp. 713–732.
[4] C. W. ANDERSON, *Extreme value theory for a class of discrete distributions with applications to some stochastic processes*, J. Appl. Probab., 7 (1970), pp. 99–113.
[5] S. M. BERMAN, *Limiting distribution of the maximum of a diffusion process*, Ann. Math. Statist., 35 (1964), pp. 319–329.
[6] S. M. BERMAN, *Sojourns and extremes of stationary processes*, Ann. Probab., 10 (1982), pp. 1–46.
[7] S. M. BERMAN, *Sojourns and extremes of a diffusion process on a fixed interval*, Adv. Appl. Probab., 14 (1982), pp. 811–832.
[8] S. M. BERMAN, *Sojourns and Extremes of Stochastic Processes*, Wadsworth and Brooks, Belmont, CA, 1992.
[9] P. BILLINSLEY, *Convergence of Probability Measures*, John Wiley, New York, 1968.
[10] J. W. COHEN, *The Single Server Queue*, revised ed., North–Holland, Amsterdam, 1982.
[11] D. R. COX AND H. D. MILLER, *The Theory of Stochastic Processes*, Chapman and Hall, London, 1965.
[12] E. CSAKI, A. FÖLDES, AND P. SALMINEN, *On the joint distribution of the maximum and its location for a linear diffusion*, Ann. Inst. H. Poincaré Probab. Statist., 23 (1987), pp. 179–194.
[13] H. E. DANIELS AND T. H. R. SKYRME, *The maximum of a random walk whose mean path has a maximum*, Adv. Appl. Probab., 17 (1985), pp. 85–99.
[14] H. E. DANIELS, *The maximum of a Gaussian process whose mean path has a maximum, with an application to the strength of bundles of fibres*, Adv. Appl. Probab., 21 (1989), pp. 315–333.
[15] J. DURBIN, *The first-passage density of a continuous Gaussian process to a general boundary*, J. Appl. Probab., 22 (1985), pp. 99–122.
[16] W. FELLER, *Introduction to Probability Theory and Its Applications*, Vol. II, John Wiley, New York, 1971.
[17] P. FLAJOLET, J. FRANÇON, AND J. VUILLEMIN, *Sequence of operations analysis for dynamic data structures*, J. Algorithms, 1 (1980), pp. 111–141.

[18] P. Flajolet and H. Prodinger, *Register allocation for unary-binary trees*, SIAM J. Comput., 15 (1986), pp. 629–640.
[19] P. Flajolet and J. M. Steyaert, *A branching process arising in dynamic hashing, trie searching and polynomial factorization*, in Proc. 9th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 140, Springer-Verlag, Berlin, 1982, pp. 239–251.
[20] P. Flajolet, *Approximate counting: A detailed analysis*, BIT, 25 (1985), pp. 113–134.
[21] P. Flajolet, M. Regnier, and R. Sedgewick, *Some uses of the Mellin integral transform in the analysis of algorithms*, Rapport de Recherche 398, INRIA, Le Chesnay, France, 1985.
[22] P. Flajolet, C. Puech, and J. Vuillemin, *The analysis of simple list structures*, Inform. Sci., 38 (1986), pp. 121–146.
[23] P. Flajolet, and A. Odlyzko, *Singularity analysis of generating functions*, SIAM J. Discrete Math., 3 (1990), pp. 216–240.
[24] J. Françon, *Combinatoire des structures de données*, thèse de doctorat d'etat, Université de Strasbourg, Strasbourg, France, 1979.
[25] J. Françon, B. Randrianarimanana, and R. Schott, *Analysis of dynamic algorithms in Knuth's Model*, Theoret. Comput. Sci., 72 (1990), pp. 147–167.
[26] C. C. Heyde, *On the growth of the maximum queue length in a stable queue*, Oper. Res., 19 (1970), pp. 447–452.
[27] D. L. Iglehart, *Limiting diffusion approximations for the many server queue and the repairman problem*, J. Appl. Probab., 2 (1965), pp. 429–441.
[28] D. L. Iglehart, *Weak convergence in queuing theory*, Adv. Appl. Probab., 5 (1973), pp. 570–594.
[29] D. L. Iglehart, *Weak convergence of compound stochastic process* I, Stochastic Process. Appl., 1 (1973), pp. 11–31.
[30] D. L. Iglehart, *Extreme values in the $GI/G/1$ queue*, Ann. Math. Statist., 43 (1972), pp. 627–635.
[31] D. Jaeschke, *The asymptotic distribution of the supremum of the standardized empirical distribution function on subintervals*, Ann. Math. Statist., 7 (1979), pp. 108–115.
[32] P. Jacquet, B. Rais, and W. Szpankowski, *Compact suffix trees resemble particia tries: Limiting distribution of depth*, SIAM J. Discrete Math., 6 (1993), pp. 197–213.
[33] P. Jacquet and W. Szpankowski, *Analysis of digital tries with Markovian dependency*, IEEE Trans. Inform. Theory, 37 (1991), pp. 1470–1475.
[34] N. L. Johnson and S. Kotz, *Distribution in Statistics: Continuous Univariate Distributions*, John Wiley, New York, 1970.
[35] J. Keilson and H. F. Ross, *Passage Time Distributions for Gaussian Markov* (*Ornstein–Uhlenbeck*) *Statistical Processes*, Selected Tables in Mathematical Statistics III, AMS, Providence, RI, 1975.
[36] D. E. Knuth, *Deletions that preserve randomness*, Trans. Software Engrg., SE-3 (1977), pp. 351–359.
[37] M. R. Leadbetter, G. Lindgreen, and H. Rootzen, *Extremes and Related Properties of Random Sequences and Processes*, Springer-Verlag, Berlin, 1983.
[38] G. Louchard, *Brownian motion and algorithm complexity*, BIT, 26 (1986), pp. 17–34.
[39] G. Louchard, *Exact and asymptotic distributions in digital and binary search trees*, Theoret. Inform. Appl., 21 (1987), pp. 479–496.
[40] G. Louchard, *Random walks, Gaussian processes and list structures*, Theoret. Comput. Sci., 53 (1987), pp. 99–124.
[41] G. Louchard, R. Schott, and B. Randrianarimanana, *Dynamic algorithms in D. E. Knuth's model: A probabilistic analysis*, Theoret. Comput. Sci., 93 (1992), pp. 201–225.
[42] G. Louchard, *Kac's formula, Levy's local time and Brownian excursion*, J. Appl. Probab., 21 (1984), pp. 479–499.
[43] G. Louchard, *Large finite population queuing systems, part* I: *The infinite server model*, Comm. Statist. Stochastic Models, 4 (1988), pp. 473–505.
[44] G. Louchard and R. Schott, *Probabilistic Analysis of Some Distributed Algorithms*; Random Structures Algorithms, 2 (1991), pp. 151–186.
[45] C. Mathieu-Kenyon and J. S. Vitter, *General methods for the analysis of the maximum size of dynamic data structures*, SIAM J. Comput., 20 (1991), pp. 807–823.
[46] J. Morrison, L. A. Shepp, and C. J. Van Wyk, *A queuing analysis of hashing with lazy deletion*, SIAM J. Comput., 16 (1987), pp. 1155–1164.
[47] B. Randrianarimanana, *Complexité des structures de données dynamiques*, thèse de doctorat, Université de Nancy 1, Vandoeuvre-lès-Nancy, France, 1989 (in French).
[48] J. S. Sadowsky and W. Szpankowski, *The probability of large queue lengths and waiting*

times in a heterogeneous multiserver queue, part I: *Tight limits*, Adv. Appl. Probab., 27 (1995), pp. 532–566.

[49] J. S. Sadowsky and W. Szpankowski, *Maximum queue length and waiting time revisited: GI/GI/c queue*, Problems Engrg. Inform. Sci., 6 (1992), pp. 157–170.

[50] P. Salminen, *On the first hitting time and the last exit time for a Brownian motion to/from a moving boundary*, Adv. Appl. Probab., 20 (1988), pp. 411–426.

[51] R. F. Serfozo, *Extreme values of birth and death processes and queues*, Stochastic Process. Appl., 27 (1988), pp. 291–306.

[52] R. F. Serfozo, *Extreme values of queue length in $M/G/1$ and $G/M/1$ systems*, Math. Oper. Res., 13 (1988), pp. 349–357.

[53] C. J. Van Wyk and J. S. Vitter, *The complexity of hashing with lazy deletion*, Algorithmica, 1 (1986), pp. 17–29.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1043–1065, August 1997

007

ORACLES THAT COMPUTE VALUES*

STEPHEN FENNER†, STEVEN HOMER‡, MITSUNORI OGIHARA§, AND ALAN SELMAN¶

Abstract. This paper focuses on complexity classes of partial functions that are computed in polynomial time with oracles in NPMV, the class of all multivalued partial functions that are computable nondeterministically in polynomial time. Concerning deterministic polynomial-time reducibilities, it is shown that

1. a multivalued partial function is polynomial-time computable with k adaptive queries to NPMV if and only if it is polynomial-time computable via $2^k - 1$ nonadaptive queries to NPMV;
2. a characteristic function is polynomial-time computable with k adaptive queries to NPMV if and only if it is polynomial-time computable with k adaptive queries to NP;
3. unless the Boolean hierarchy collapses, for every k, k adaptive (nonadaptive) queries to NPMV are different than $k + 1$ adaptive (nonadaptive) queries to NPMV.

Nondeterministic reducibilities, lowness, and the difference hierarchy over NPMV are also studied. The difference hierarchy for partial functions does not collapse unless the Boolean hierarchy collapses, but, surprisingly, the levels of the difference and bounded query hierarchies do not interleave (as is the case for sets) unless the polynomial hierarchy collapses.

Key words. computational complexity, complexity classes, relativized computation, bounded query classes, Boolean hierarchy, multivalued functions, NPMV

AMS subject classifications. 68Q05, 68Q10, 68Q15, 03D10, 03D15

PII. S0097539793247439

1. Introduction.

In this paper, we study classes of partial functions that can be computed in polynomial time with the help of oracles that are themselves partial functions. We want to know whether there is a difference between computing with function oracles and computing with set oracles. Specifically, we investigate classes of partial functions that can be computed in polynomial time with oracles in NPMV and NPSV, that is, the classes $\mathrm{PF}^{\mathrm{NPMV}}$ and $\mathrm{PF}^{\mathrm{NPSV}}$.

NPMV is the set of all partial multivalued functions that are computed nondeterministically in polynomial time, and NPSV is the set of all partial functions in this class that are single-valued. NPMV captures the complexity of computing witnesses to problems in NP. For example, let *sat* denote the partial multivalued function defined by $sat(x)$ maps to a value y if and only if x encodes a formula of propositional logic and y encodes a satisfying assignment of x. Then *sat* belongs to NPMV, and the domain of *sat* (i.e., the set of all words x for which the output of $sat(x)$ is nonempty) is the NP-complete satisfiability problem, SAT. Also, NPMV captures

* Received by the editors April 14, 1993; accepted for publication (in revised form) August 17, 1995.

http://www.siam.org/journals/sicomp/26-4/24743.html

† Department of Computer Science, University of Southern Maine, Portland, ME 04103 (fenner@cs.usm.maine.edu). The research of this author was partially supported by National Science Foundation grants CCR-9209833 and CCR-9501794.

‡ Department of Computer Science, Boston University, Boston, MA 01003 (homer@cs.bu.edu). The research of this author was partially supported by National Science Foundation grants CCR-9103055, CCR-9400229, and INT-9123551.

§ Department of Computer Science, University of Rochester, Rochester, NY 14627 (ogihara@cs.rochester.edu). The research of this author was done while visiting at the Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260 and while affiliated with Department of Computer Science, University of Electro-Communications, Tokyo, Japan and was partially supported by National Science Foundation grants CCR-9002292 and NSF-INT-9116781/JSPS-ENGR-207.

¶ Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260 (selman@cs.buffalo.edu). The research of this author was partially supported by National Science Foundation grants CCR-9002292, INT-9123551, and CCR-9400229.

the complexity of inverting polynomial-time honest functions. To wit, the inverse of every polynomial-time honest function belongs to NPMV, and the inverse of every one–one polynomial-time honest function belongs to NPSV.

The class of partial functions with oracles in NP, namely, $\mathrm{PF}^{\mathrm{NP}}$ has been well studied [13], as have been the corresponding class of partial functions that can be computed nonadaptively with oracles in NP, viz $\mathrm{PF}_{tt}^{\mathrm{NP}}$ [15], and the classes of partial functions that are obtained by limiting the number of queries to some value $k \geq 1$, namely, $\mathrm{PF}^{\mathrm{NP}[k]}$ and $\mathrm{PF}_{tt}^{\mathrm{NP}[k]}$ [2, 1]. A rich body of results is known about these classes.

Here we raise the question "What is the difference between computing with an oracle in NPMV versus an oracle in NP?" The answer is not obvious. If the partial function *sat* is provided as an oracle to some polynomial-time computation M, then on a query x, where x encodes a satisfiable formula of propositional logic, the oracle will return some satisfying assignment y. However, if the oracle to M is the NP-compete set SAT, then to this query x, the oracle will only return a Boolean value "yes." On the other hand, by the well-known self-reducibility of SAT, M could compute y for itself by judicious application of a series of adaptive queries to SAT. Indeed, Theorem 2.4 states that unbounded access to an oracle in NPMV is no more powerful than such an access to an oracle in NP. However, in section 3 we will see that the situation for bounded query classes is much more subtle. In general, *function oracles cannot be replaced by set oracles*—but set oracles are still useful. We will show that every partial multivalued function in $\mathrm{PF}^{\mathrm{NPMV}[k]}$ can be computed by a partial multivalued function of the form $f \circ g$, where f is in NPMV and g is a single-valued function belonging to $\mathrm{PF}^{\mathrm{NP}[k]}$. Moreover, most surprisingly, the relationship between access to an oracle in NPMV and access to an oracle in NP is tight regarding set recognition; that is, $\mathrm{P}^{\mathrm{NPMV}[k]} = \mathrm{P}^{\mathrm{NP}[k]}$. This means that when we are computing characteristic functions, k bounded queries to an oracle in NPMV give no more information than the same number of queries to an oracle in NP.

We will show that the levels of the nonadaptive and adaptive bounded query hierarchies interleave (for example, k adaptive queries to a partial function in NPMV are equivalent to $2^k - 1$ nonadaptive queries to a partial function in NPMV), and we will show that these bounded query hierarchies collapse only if the Boolean hierarchy collapses.

In section 4, we study nondeterministic polynomial-time reductions to partial functions in NPMV. Unlike the case for deterministic functions, we will see that just one query to an NP oracle can substitute for an unbounded number of queries to any partial function in NPMV. The hierarchy that is formed by iteratively applying NP reductions is an analogue of the polynomial hierarchy, and we will show that this hierarchy collapses if and only if the polynomial hierarchy collapses.

In section 5, we will study the difference hierarchy over NPMV. We define $f - g$ to be a partial multivalued function that maps x to y if and only if f maps x to y and g does *not* map x to y, and we define $\mathrm{NPMV}(k) = \{f_1 - (f_2 - (\cdots - f_k)) \mid f_1, \ldots, f_k \in \mathrm{NPMV}\}$. Since the properties of the bounded query hierarchies over NPMV are largely similar to those over NP, one might hope that the same thing happens here—that the difference hierarchy over NPMV and the difference hierarchy over NP are similar. However, the contour of this hierarchy is, to our astonishment, totally different than its analogue for NP. Although $\mathrm{BH} = \bigcup_k \mathrm{NP}(k) \subseteq \mathrm{P}^{\mathrm{NP}}$, with no assumption, we will show that NPMV(2) is included in $\mathrm{PF}^{\mathrm{NPMV}}$ if and only if $\mathrm{PH} = \Delta_2^{\mathrm{P}}$. Also, in this section, we will introduce the notion of NPMV-lowness, and we will give a complete

characterization of NPMV-lowness.

Consideration of reduction classes with oracles in NPSV, to be studied in section 6, is motivated in part by a desire to understand how difficult it is to compute satisfying assignments for satisfiable formulas. We take the point of view that a partial multivalued function is easy to compute if for each input string in the domain of the function, some value of the function is easy to compute. For this reason, we define the following technical notions. Given partial multivalued functions f and g, define g to be a *refinement* of f if $dom(g) = dom(f)$ and for all $x \in dom(g)$ and all y, if y is a value of $g(x)$, then y is a value of $f(x)$. Let $\mathcal{F}$ and $\mathcal{G}$ be classes of partial multivalued functions. Purely as a convention, if f is a partial multivalued function, we define $f \in_c \mathcal{G}$ if $\mathcal{G}$ contains a refinement g of f, and we define $\mathcal{F} \subseteq_c \mathcal{G}$ if for every $f \in \mathcal{F}$, $f \in_c \mathcal{G}$. This notation is consistent with our intuition that $\mathcal{F} \subseteq_c \mathcal{G}$ should entail that the complexity $\mathcal{F}$ is not greater than the complexity of $\mathcal{G}$. Let PF denote the class of partial functions that are computable deterministically in polynomial time. The assertion "NPMV $\subseteq_c$ PF" means that every partial multivalued function in NPMV has a refinement that can be computed efficiently by some deterministic polynomial-time transducer. It is well known that $sat \in_c$ PF if and only if NPMV $\subseteq_c$ PF if and only if P = NP [15]. Thus one does not expect that $sat \in_c$ PF. Is *sat* computable in some larger single-valued class of partial functions? Selman [15] showed that PF $\subseteq$ NPSV $\subseteq \mathrm{PF}_{tt}^{\mathrm{NP}}$. If $sat \in_c$ NPSV, then the polynomial hierarchy collapses [11], and it is an open question whether $sat \in_c$ NPSV or whether $sat \in_c \mathrm{PF}_{tt}^{\mathrm{NP}}$. (Watanabe and Toda [18] have shown that $sat \in_c \mathrm{PF}_{tt}^{\mathrm{NP}}$ relative to a random oracle.) We will consider classes of the form $\mathrm{PF}^{\mathrm{NPSV}[k]}$ and $\mathrm{PF}_{tt}^{\mathrm{NPSV}[k]}$, where $k \geq 1$, and we will show that the adaptive and the nonadaptive classes form proper hierarchies unless the Boolean hierarchy collapses. Thus these classes form a finer classification in which to study the central question of whether *sat* has a refinement in some interesting class of single-valued partial functions.

Finally, we note in passing that the complexity theory of decision problems, i.e., of sets, is extremely well developed. Although the computational problems in which we are most interested are naturally thought of as partial multivalued functions, the structural theory to support classification of these problems has been slight. By introducing several natural hierarchies of complexity classes of partial multivalued functions, with strong evidence supporting these claims, we intend this paper to make significant steps in correcting this situation.

2. Preliminaries. We fix Σ to be the finite alphabet $\{0, 1\}$. $<$ denotes the standard canonical lexicographic order on Σ^*. Let $f : \Sigma^* \mapsto \Sigma^*$ be a partial multivalued function. We write $f(x) \mapsto y$ (or $f(x)$ maps to y), if y is a value of f on input string x. Define $graph(f) = \{\langle x, y\rangle \mid f(x) \mapsto y\}$, $dom(f) = \{x \mid \exists y(f(x) \mapsto y)\}$, and $range(f) = \{y \mid \exists x(f(x) \mapsto y)\}$. We will say that f is undefined at x if $x \notin dom(f)$.

A transducer T is a nondeterministic Turing machine with a read-only input tape and a write-only output tape and accepting states in the usual manner. A transducer T computes a value y on an input string x if there is an accepting computation of T on x for which y is the final content of T's output tape. (In this case, we will write $T(x) \mapsto y$.) Such transducers compute partial, multivalued functions. (Since transducers do not typically accept all input strings, when we write "function," "partial function" is always intended. If a function f is total, it will always be explicitly noted.)

- NPMV is the set of all partial, multivalued functions computed by nondeterministic polynomial-time bounded transducers;
- NPSV is the set of all $f \in$ NPMV that are single-valued;

- PF is the set of all partial functions computed by deterministic polynomial-time-bounded transducers.

A function f belongs to NPMV if and only if it is polynomially length-bounded and $graph(f)$ belongs to NP. The domain of every function in NPMV belongs to NP. These definitions originate in Book, Long, and Selman's study of restricted-access relativizations [5].

Now we describe oracle Turing machines with oracles that compute partial functions. For the moment, we assume that the oracle is a single-valued partial function. Let $\perp$ be a symbol not belonging to the finite alphabet Σ. In order for a machine M to access a partial function oracle, M contains a write-only input oracle tape, a separate read-only output oracle tape, and a special oracle call state q. When M enters state q, if the string currently on the oracle input tape belongs to the domain of the oracle partial function, then the result of applying the oracle appears on the oracle output tape, and if the string currently on the oracle input tape does not belong to the domain of the oracle partial function, then the symbol $\perp$ appears on the oracle output tape. Thus, if the oracle is some partial function g, given an input x to the oracle, the oracle, if called, returns a value $g(x)$ if one exists and returns $\perp$ otherwise. (It is possible that M may read only a portion of the oracle's output if the oracle's output is too long to read with the resources of M.) We shall assume, without loss of generality, that M never makes the same oracle query more than once, i.e., all of M's queries (on any possible computation path) are distinct. $\mathrm{PF}^{\mathrm{NP}}$ is the class of partial functions computed in polynomial time with oracles in NP. $\mathrm{PF}^{\mathrm{NP}}_{tt}$ is the class of partial functions that can be computed nonadaptively with oracles in NP; that is, a partial function f is in $\mathrm{PF}^{\mathrm{NP}}_{tt}$ if there is a deterministic oracle Turing machine transducer T such that $f \in \mathrm{PF}^{\mathrm{NP}}$ via T with an oracle L in NP and a total polynomial-time computable function $g : \{0,1\}^* \mapsto (c\{0,1\}^*)^*$ such that, for each input x to T, T only makes queries to L from the list $g(x)$.

If g is a single-valued partial function and M is a deterministic oracle transducer as just described, then we let $M[g]$ denote the single-valued partial function computed by M with oracle g.

DEFINITION 2.1. *Let f and g be multivalued partial functions. f* is Turing reducible to g in polynomial time, *$f \leq^{\mathrm{P}}_{\mathrm{T}} g$, if for some deterministic oracle transducer M, for every single-valued refinement g' of g, $M[g']$ is a single-valued refinement of f.*[1]

PROPOSITION 2.1. *Polynomial-time Turing reducibility, $\leq^{\mathrm{P}}_{\mathrm{T}}$, is a reflexive and transitive relation over the class of all partial multivalued functions.*

Let $\mathcal{F}$ be a class of partial multivalued functions. $\mathrm{PF}^{\mathcal{F}}$ denotes the class of partial multivalued functions f that are $\leq^{\mathrm{P}}_{\mathrm{T}}$-reducible to some $g \in \mathcal{F}$. $\mathrm{PF}^{\mathcal{F}[k]}$ (respectively, $\mathrm{PF}^{\mathcal{F}[\log]}$) denotes the class of partial multivalued functions f that are $\leq^{\mathrm{P}}_{\mathrm{T}}$-reducible to some $g \in \mathcal{F}$ via a machine that, on input x, makes k adaptive queries (respectively,

[1] A notion of polynomial-time Turing reducibility between partial functions is defined by Selman [15]. It is important to note that the definition given here is *different* than the one given there. Here the oracle "knows" when a query is not in its domain. In the earlier definition, this is not the case. The authors recommend that the reducibility defined in the earlier paper should in the future be denoted as $\leq^{\mathrm{PP}}_{\mathrm{T}}$, which is the common notation for reductions between promise problems. We make this recommendation because conceptually and technically this reducibility between functions is equivalent to a promise problem reduction. Also, we note that the reducibility defined by Selman [15] is not useful for our purposes here. In particular, it is easy to see that iterating reductions between functions in NPMV does not gain anything new unless the oracle is endowed with the ability to know its domain.

$\mathcal{O}(\log |x|)$ adaptive queries) to its oracle.

$\mathrm{PF}_{tt}^{\mathcal{F}}$ denotes the class of partial multivalued functions f that are $\leq_{\mathrm{T}}^{\mathrm{P}}$-reducible to some $g \in \mathcal{F}$ via an oracle Turing-machine transducer that queries its oracle nonadaptively. That is, a partial multivalued function f is in $\mathrm{PF}_{tt}^{\mathcal{F}}$ if there is an oracle Turing-machine transducer T such that $f \in \mathrm{PF}^{\mathcal{F}}$ via T with an oracle g in $\mathcal{F}$ and a polynomial time computable function $h : \{0,1\}^* \mapsto (c\{0,1\}^*)^*$ such that, for each input x to T, T only calls the oracle g on strings in the list $h(x)$.

$\mathrm{PF}_{tt}^{\mathcal{F}[k]}$ denotes the class of partial multivalued functions f that are $\leq_{\mathrm{T}}^{\mathrm{P}}$-reducible to some $g \in \mathcal{F}$ via a machine that makes k nonadaptive queries to its oracle, i.e., just as in the last paragraph, but with $h : \{0,1\}^* \mapsto (c\{0,1\}^*)^k$.

$\mathrm{P}^{\mathcal{F}}$, $\mathrm{P}^{\mathcal{F}[k]}$, $\mathrm{P}^{\mathcal{F}[\log]}$, $\mathrm{P}_{tt}^{\mathcal{F}}$, and $\mathrm{P}_{tt}^{\mathcal{F}[k]}$, respectively, denote the classes of all characteristic functions contained in $\mathrm{PF}^{\mathcal{F}}$, $\mathrm{PF}^{\mathcal{F}[k]}$, $\mathrm{PF}^{\mathcal{F}[\log]}$, $\mathrm{PF}_{tt}^{\mathcal{F}}$, and $\mathrm{PF}_{tt}^{\mathcal{F}[k]}$.

For a class of sets $\mathcal{C}$, we may say that $\mathrm{PF}^{\mathcal{C}}$ denotes the class of partial multivalued functions that are $\leq_{\mathrm{T}}^{\mathrm{P}}$-reducible to the characteristic function of some set in $\mathcal{C}$. $\mathrm{PF}^{\mathcal{C}[k]}$, $\mathrm{PF}^{\mathcal{C}[\log]}$, $\mathrm{PF}_{tt}^{\mathcal{C}}$, $\mathrm{PF}_{tt}^{\mathcal{C}[k]}$, $\mathrm{P}^{\mathcal{C}}$, $\mathrm{P}^{\mathcal{C}[k]}$, $\mathrm{P}^{\mathcal{C}[\log]}$, $\mathrm{P}_{tt}^{\mathcal{C}}$, and $\mathrm{P}_{tt}^{\mathcal{C}[k]}$ are defined similarly. In particular, $\mathrm{PF}^{\mathrm{NP}}$ is the class of partial multivalued functions computed in polynomial time with oracles in NP, and $\mathrm{PF}_{tt}^{\mathrm{NP}}$ is the class of partial functions that can be computed nonadaptively with oracles in NP. In the current literature, these classes contain single-valued functions only. The reason is that, heretofore, polynomial-time Turing reducibility, $\leq_{\mathrm{T}}^{\mathrm{P}}$, has been defined as a binary relation over single-valued objects. To see that $\mathrm{PF}^{\mathrm{NP}}$ contains partial functions that are not single-valued, consider the partial single-valued function *maxsat* that on an input x, where x encodes a formula of propositional logic, maps to the encoding of the lexicographically largest satisfying assignment of x, if $x \in \mathrm{SAT}$. Clearly, $maxsat \in \mathrm{PF}^{\mathrm{NP}}$, and $sat \leq_{\mathrm{T}}^{\mathrm{P}} maxsat$ by Definition 2.1, so the partial multivalued function *sat* belongs to $\mathrm{PF}^{\mathrm{NP}}$. Readers are free to interpret references to $\mathrm{PF}^{\mathrm{NP}}$ and $\mathrm{PF}_{tt}^{\mathrm{NP}}$ with their familiar meaning because the results that we will state for these classes, and for the corresponding bounded query classes, remain correct if the classes are replaced with the result of including only the single-valued partial functions that they contain.

Given a class of partial multivalued functions $\mathcal{F}$, let $\mathcal{F}/_{sv}$ denote the class of single-valued partial functions that $\mathcal{F}$ contains.

All the classes of partial multivalued functions that we have defined, other than NPMV, are closed "backwards" under refinement. That is, with the exception of NPMV, property (1) below holds for each of these classes $\mathcal{F}$:

$$f \in \mathcal{F} \wedge f \text{ is a refinement of } g \rightarrow g \in \mathcal{F}. \tag{1}$$

Let us say that classes that satisfy property (1) are *c-closed*. The c-closure of a class is c-closed. Let us say that a *basis* for a class $\mathcal{F}$ is a subset $\mathcal{F}'$ of $\mathcal{F}$ such that for all $f \in \mathcal{F}$, there is an $f' \in \mathcal{F}'$ such that f' is a refinement of f. Essentially all interesting c-closed classes are uncountable, but this is not problematic because they all arise as the c-closure of classes that are countable and effectively enumerable (that is, they are indexed by machines of some appropriate type). Property (2) holds for every class of partial functions $\mathcal{F}$ that is Turing reducible in polynomial time to a class of single-valued partial functions.

$$f \in \mathcal{F} \rightarrow \exists f'[f' \text{ is a single-valued refinement of } f \wedge f' \in \mathcal{F}]. \tag{2}$$

For example, property (2) holds for $\mathrm{PF}^{\mathrm{NP}}$. Property (2) states that the set of single-valued functions in $\mathcal{F}$ is a basis for $\mathcal{F}$. (To use an analogy from lattice theory, if

one thinks of the single-valued functions as "atoms," then property (2) is the "atomic basis property.") Also, note that "is a refinement of" is reflexive and transitive over the class of all partial multivalued functions.

PROPOSITION 2.2. *If $\mathcal{F}$ satisfies property* (1)*, then $g \in_c \mathcal{F} \leftrightarrow g \in \mathcal{F}$ and $\mathcal{G} \subseteq_c \mathcal{F} \leftrightarrow \mathcal{G} \subseteq \mathcal{F}$.*

Thus $\in_c$ is identical to class containment and $\subseteq_c$ is identical to class inclusion for the classes we have defined.

PROPOSITION 2.3. *If $\mathcal{F}$ satisfies property* (2) *and $\mathcal{G}$ satisfies property* (1)*, then $\mathcal{F}/_{sv} \subseteq \mathcal{G}/_{sv} \leftrightarrow \mathcal{F} \subseteq \mathcal{G}$.*

Beigel [2] observed that for all $k \geq 1$, $\mathrm{PF}^{\mathrm{NP}[k]}/_{sv} \subseteq \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]}/_{sv}$. Using Proposition 2.3, it follows that $\mathrm{PF}^{\mathrm{NP}[k]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]}$. This example illustrates that known inclusion results for the classes we are considering remain true under the new interpretation that these classes contain multivalued functions. Thus passing to multivalued functions does not disturb our current understanding of previously studied function classes. We are recasting the definitions in no small part because we will be dealing with many classes that (most likely) do not satisfy property (2), and hence our results are strictly more general.

Obviously, $\mathrm{PF}^{\mathrm{NP}} \subseteq \mathrm{PF}^{\mathrm{NPMV}}$. Conversely, for a function $f \in \mathrm{NPMV}$, define f' to be a function such that $f'(x) = \min\{y \mid f(x) \mapsto y\}$. The function f' is a single-valued refinement of f and in $\mathrm{PF}^{\mathrm{NP}}$, so $\mathrm{NPMV} \subseteq \mathrm{PF}^{\mathrm{NP}}$ by Proposition 2.2. This implies that $\mathrm{PF}^{\mathrm{NPMV}} \subseteq \mathrm{PF}^{\mathrm{PF}^{\mathrm{NP}}} = \mathrm{PF}^{\mathrm{NP}}$ since $\leq_{\mathrm{T}}^{\mathrm{P}}$ is transitive. Therefore, the following theorem holds.

THEOREM 2.4. $\mathrm{PF}^{\mathrm{NPMV}} = \mathrm{PF}^{\mathrm{NP}}$.

Theorem 2.4 states that unbounded access to an oracle in NPMV is no more powerful than such an access to an oracle in NP.

The following examples, the first of which was pointed out by Buhrman [3], illustrate the power of $\mathrm{PF}^{\mathrm{NPMV}}$ and $\mathrm{PF}_{tt}^{\mathrm{NPMV}}$. Consider the partial multivalued function *maxTsat* defined as follows:

> $maxTsat(x) \mapsto y$ if y is a satisfying assignment of x with the maximum number of *true*s.

Obviously, *maxTsat* belongs to $\mathrm{PF}^{\mathrm{NPMV}}$. Let f be a function that maps a pair (x, n) to y if and only if y is a satisfying assignment of x with n *true*s. Since the number of variables in a formula is bounded by its length, it holds that $maxTsat(x) = f(x, n_x)$, where n_x is the largest $n, 1 \leq n \leq |x|$, such that $(x, n) \in dom(f)$. This implies that $maxTsat \in \mathrm{PF}_{tt}^{\mathrm{NPMV}}$.

Similarly, the partial multivalued function *maxclique* that on input a graph G outputs a clique of maximum size belongs to $\mathrm{PF}_{tt}^{\mathrm{NPMV}}$. The function *MaxEdgeWeightClique* that is defined over edge-weighted graphs and that outputs a clique of maximum weight, if G has a clique, belongs to $\mathrm{PF}^{\mathrm{NPMV}}$ but may not belong to $\mathrm{PF}_{tt}^{\mathrm{NPMV}}$ because weights may grow exponentially.

We should note that several of the classes we investigate here seem to capture the complexity of finding witnesses to NP-optimization problems. This observation is explored by Chen and Toda [9] and by Wareham [17].

3. Bounded query classes. In this section, we prove a number of basic results clarifying the structure of the bounded adaptive and nonadaptive query hierarchies over NPMV, both for computing functions and for set recognition. The new hierarchies are mostly analogous to those over NP, but there are some interesting and subtle differences. General techniques developed in this section are reminiscent of the

"mind-change" technique [2, 19]. We will use them first to compare $\mathrm{PF}^{\mathrm{NPMV}[k]}$ and $\mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$ with $\mathrm{PF}^{\mathrm{NP}[k]}$ and $\mathrm{PF}_{tt}^{\mathrm{NP}[k]}$, respectively.

The following two propositions are central to the rest of the paper and will be used in several places later on: Theorems 3.3, 3.5, 3.7, and 3.8, Lemma 5.10, and Proposition 6.6. Each proposition abstracts the general idea, common to all these results, that we can replace the oracle queries in any $\mathrm{PF}^{\mathrm{NPMV}}$ computation by nondeterminism in a way that preserves information about outputs of the computation. Proposition 3.1 deals with computations making adaptive queries; Proposition 3.2 deals with nonadaptive queries. After we prove them we will discuss briefly how they are used.

PROPOSITION 3.1. *Let $t \in \mathrm{PF}$. Let f be in $\mathrm{PF}^{\mathrm{NPMV}}$ be computed by a deterministic oracle Turing-machine transducer M with $g \in \mathrm{NPMV}$ as the oracle. Suppose that M on x makes $t(x)$ queries to its oracle. Then there is an* NPMV *function $s[M,g] : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$ that satisfies the following conditions.*

1. *$\lambda x.[s[M,g](x, 0^{t(x)})]$ is total, single-valued, and polynomial-time computable.*
2. *For every x, there uniquely exists $a_x \in \Sigma^{t(x)}$ such that*
   (a) *for every $a \in \Sigma^{t(x)}$, $(x,a) \in dom(s[M,g])$ if and only if $a \leq a_x$,*
   (b) *$f(x)$ is undefined if and only if $s[M,g](x,a_x)$ maps to 0, and*
   (c) *for every y, if $s[M,g](x,a_x)$ maps to $1y$, then $f(x)$ maps to y.*

Proof. Let t, f, M, and g be as in the hypothesis. The idea of the proof is as follows: given an input x, say that a string $a \in \Sigma^{t(x)}$ is *okay* if there is a legitimate computation path of $M(x)$ where the ith query is in $dom(g)$ if and only if the ith bit of a is 1. The "magic" string a_x that we seek will be the lexicographically maximum okay string. Computing the function $s[M,g]$ involves, among other things, guessing if there is an okay string no smaller than the given input string $a \in \Sigma^{t(x)}$. This must be done indirectly since one cannot necessarily verify—even nondeterministically—whether a given string is okay.

Let N be a polynomial-time nondeterministic Turing machine witnessing that $g \in \mathrm{NPMV}$. Define U to be the following machine: On input x and $b \in \Sigma^{t(x)}$, U simulates M on input x in the following manner:

- For each $i, 1 \leq i \leq t(x)$, when M makes its ith query q_i, U behaves as follows:
  —If the ith symbol in b is a 0, then U assumes that the answer is $\perp$.
  —If the ith symbol in b is a 1, then U simulates N on q_i. If N on q_i does not accept, then U halts without accepting, and if N on q_i outputs some z_i, then U assumes that the answer is z_i.
- When M enters a halting state, U behaves as follows:
  —If M rejects, then U outputs 0.
  —If M outputs y, then U outputs $1y$.

Let r be the NPMV function defined by U. For every x, U on $(x, 0^{t(x)})$ makes no nondeterministic guesses. So U on $(x, 0^{t(x)})$ always has a unique output and $\lambda x.[r(x, 0^{t(x)})]$ is polynomial-time computable.

For a given x, let b_x be the largest $b \in \Sigma^{t(x)}$ such that U on (x,b) has an output, and let π be an arbitrary computation path of U on (x, b_x) that leads to an output. Suppose that along path π, U generates query strings $q_1, \ldots, q_{t(x)}$ in this order and computes the answers to them as $z_1, \ldots, z_{t(x)}$, respectively. By definition, for every i such that $z_i \neq \perp$, $g(q_i) \mapsto z_i$. Furthermore, we claim that for every i such that $z_i = \perp$, $q_i \notin dom(g)$. This is seen as follows: Assume that there is some i such that $z_i = \perp$ and $q_i \in dom(g)$. Let j be the smallest such i. By the minimality of j, there exist some c and some computation path π' of U on (x,c) such that along path π',

(i) U has an output,
(ii) the first j queries U computes are $q_1, \ldots, q_j$,
(iii) the first $j-1$ answers U computes are $z_1, \ldots, z_{j-1}$,
(iv) the jth answer U computes is not $\perp$, and
(v) π and π' agree until q_j.

Let $b_x = u0w$ with $|u| = j-1$. By (ii), (iii), and (iv), we have $c = u1v$ for some v. Thus $c > b_x$. By (i), U on (x, c) has an output. So by the maximality of b_x, $b_x \geq c$, which contradicts $c > b_x$. Therefore, for every i such that $z_i = \perp$, $q_i \notin dom(g)$.

Thus we see that all the answers $z_1, \ldots, z_{t(x)}$ are correct. Define g' to be a single-valued refinement of g that is defined by path π. U on (x, b_x) along path π correctly simulates $M[g']$ on x. Thus it holds that

$$\begin{aligned} x \in dom(f) &\leftrightarrow M[g'] \text{ has an output} \\ &\leftrightarrow U \text{ on } (x, b_x) \text{ along path } \pi \text{ outputs a string of the form } 1y, \text{ and} \\ x \notin dom(f) &\leftrightarrow M[g'] \text{ does not have an output} \\ &\leftrightarrow U \text{ on } (x, b_x) \text{ along path } \pi \text{ outputs } 0. \end{aligned}$$

Therefore, U on (x, b_x) along path π outputs 0 if and only if $f(x)$ is undefined, and if U on (x, b_x) along path π outputs $1y$, then $f(x)$ maps to y.

Now define V to be the machine that, on input (x, a) with $|a| = t(x)$, behaves as follows:

- if $a = 0^{t(x)}$, then V simulates U on (x, a), and
- if $a \neq 0^{t(x)}$, then V guesses $b \in \Sigma^{t(x)}$ with $b \geq a$ and simulates U on (x, b).

Let s be the NPMV function defined by V. We claim that s is the desired function. Since V and U are the same on input $(x, 0^{t(x)})$, $\lambda x.[s(x, 0^{t(x)})]$ is total, single-valued, and polynomial-time computable. Let a_x be the largest $a \in \Sigma^{t(x)}$ such that $(x, a) \in dom(s)$. It is not hard to see that $a_x = b_x$. Since b_x is the largest b such that $(x, b) \in dom(r)$, and V on (x, a) simulates U on (x, b) for all $b \geq a$ except when $a = 0^{t(x)}$, it holds that

(i) for every $a > a_x$, $(x, a) \notin dom(s)$,
(ii) for every $a \leq a_x$, $(x, a) \in dom(s)$,
(iii) $x \notin dom(f)$ if and only if $s(x, a_x) \mapsto 0$, and
(iv) if $s(x, a_x) \mapsto 1y$, then $f(x) \mapsto y$.

Hence all the required properties are satisfied. This proves the proposition. □

PROPOSITION 3.2. *Let $t \in$ PF. Let f in $\mathrm{PF}_{tt}^{\mathrm{NPMV}}$ be computed by a deterministic oracle Turing-machine transducer M with $g \in$ NPMV as the oracle. Suppose that M on x makes $t(x)$ queries. Then there is an* NPMV *function $s[M, g] : \Sigma^* \times \{0, \ldots, t(x)\} \mapsto \Sigma^*$ that satisfies the following conditions:*

1. *$\lambda x.[s[M, g](x, 0)]$ is total, single-valued and polynomial-time computable,*
2. *for every x and $0 \leq m \leq n \leq t(x)$, if $(x, n) \in dom(s[M, g])$, then $(x, m) \in dom(s[M, g])$,*
3. *for every x, $f(x)$ is undefined if and only if $s[M, g](x, n_x)$ maps to 0, and*
4. *for every x and y, $f(x)$ maps to y if and only if $s[M, g](x, n_x)$ maps to $1y$, where n_x is the largest $n \in \{0, \ldots, t(x)\}$ such that $(x, n) \in dom(s[M, g])$.*

Proof. Let t, f, M, and g be as in the hypothesis and let N be a nondeterministic Turing machine witnessing that $g \in$ NPMV. The idea of this proof is analogous to, but simpler than, that of the last proposition. The "magic" number n_x that we seek will be the number of queries of $M(x)$ that are in $dom(g)$.

Let h be the function defined by the following machine U: On input x and $n \leq t(x)$, U behaves as follows:

(A) U first computes all the query strings $q_1, \ldots, q_{t(x)}$ of M on x.
(B) If $n = 0$, then for every i, U assumes that the answer to q_i is $\perp$. If $n > 0$, then U does the following:
- For each i, U simulates N on q_i. If N does not accept q_i, then U assumes that the answer to q_i is $\perp$, and if N outputs w on q_i, then U assumes that the answer to q_i is w. After doing this, if the number of answers obtained as $\perp$ is larger than $t(x) - n$, then U halts without accepting.

(C) U simulates M on x using the answers computed in (B). If M rejects, then U outputs 0, and if M outputs z, then U outputs $1z$.

We claim that h is the desired function.

For every x, U on $(x, 0)$ runs deterministically and always has an output. So $\lambda x.[h(x, 0)]$ is total, single-valued, and polynomial-time computable. Suppose $0 < m \leq n \leq t(x)$ and $(x, n) \in dom(h)$. Then U must have an accepting path in step (B), where it obtains at most $t(x) - n$ query answers as $\perp$. The same set of query answers will also allow U to accept on input (x, m).

For each x, let n_x be the maximum n such that $(x, n) \in dom(h)$. For every x, n_x coincides with the number of queries of M on x that are in $dom(g)$. Let π be any computation path of U on (x, n_x) leading to an output. Let $z_1, \ldots, z_{t(x)}$ be the answers that U computes along path π for queries $q_1, \ldots, q_{t(x)}$, respectively. Then by the maximality of n_x, for every i, $z_i = \perp$ if and only if $q_i \notin dom(g)$ and if $z_i \neq \perp$, then $g(q_i)$ maps to z_i. So the output along path π is 0 if and only if $f(x)$ is undefined, and if the output is $1y$, then $f(x)$ maps to y. Therefore, h is the desired function. □

We will use Propositions 3.1 and 3.2 in three ways. First, we can simulate the behavior of a $\mathrm{PF}^{\mathrm{NPMV}}$ (respectively, $\mathrm{PF}_{tt}^{\mathrm{NPMV}}$) computation on input x purely nondeterministically, *provided* we know a_x (respectively, n_x). Such a simulation $s[M, g]$ always accepts, tells us whether $M(x)$ outputs a value, and, if so, provides us with an output. Second, $dom(s[M, g])$ is such that we can find a_x (respectively, n_x) by binary search with an NP oracle. Third, the fact that $s[M, g](x, 0^{t(x)})$ (respectively, $s[M, g](x, 0)$) can be computed deterministically saves us an NP query so that we can get an exact connection between bounded adaptive and nonadaptive NPMV queries in Theorem 3.5.

Let f and g be partial multivalued functions. $f \circ g$ denotes the function h such that for every x,
- $h(x)$ maps to y if and only if there exists some z such that $g(x)$ maps to z and $f(z)$ maps to y.

Let $\mathcal{F}$ and $\mathcal{G}$ be classes of partial multivalued functions. $\mathcal{F} \circ \mathcal{G}$ denotes $\{f \circ g \mid f \in \mathcal{F}$ and $g \in \mathcal{G}\}$.

Although composition is a natural operator and an important tool in our investigations, we should caution that the classes we consider tend not to be closed under composition, and the composition of two easy-to-compute functions may be very difficult. To see this, consider the functions r and s defined as follows: $r(x) \mapsto 1$ for all $x \neq 0$, and r is undefined at $x = 0$; $s(x) \mapsto 0$ for all strings x, and $s(x) \mapsto 1$ if $x \in K$, where K is a complete recursively enumerable set. The partial multivalued functions r and s have refinements in PF, but $dom(r \circ s) = K$, so $r \circ s$ does *not* have a refinement in PF.

The following theorem relates computing with an oracle in NPMV$[k]$ to computing with an oracle in NP$[k]$. In particular, we see that every partial multivalued function in $\mathrm{PF}^{\mathrm{NPMV}[k]}$ can be computed by a partial multivalued function of the form $f \circ g$,

where f is in NPMV and g is a single-valued function in $\mathrm{PF}^{\mathrm{NP}[k]}$.

THEOREM 3.3.

(1) *For every* $k \geq 1$, $\mathrm{PF}^{\mathrm{NPMV}[k]} \subseteq_c \mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[k]}/_{sv})$.

(2) *For every* $k \geq 1$, $\mathrm{PF}^{\mathrm{NPMV}[k]} \subseteq \mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[k]}$.

Proof. Let $f \in \mathrm{PF}^{\mathrm{NPMV}[k]}$ via a deterministic oracle Turing-machine transducer M with $g \in \mathrm{NPMV}$ as the oracle. Let $h = s[M, g]$ be the function defined in Proposition 3.1. Let V be a machine witnessing that $h \in \mathrm{NPMV}$. Define b to be a function that maps x to a_x, where a_x is the largest $a \in \Sigma^k$ such that $(x, a) \in dom(h)$. Recall that $b(x)$ is defined for every x. A binary search algorithm over $\Sigma^k - \{0^k\}$ computes b in polynomial time with oracle $dom(h)$. The number of questions is exactly k, so $b \in \mathrm{PF}^{\mathrm{NP}[k]}/_{sv}$. Now define f' to be a function that maps x to $(x, b(x))$. Define V' to be a machine that on input (x, a) simulates V on (x, a) and does not accept if either V does not accept or V outputs 0, and outputs y if V outputs $1y$. Let h' be the partial multivalued function defined by V'. Then $h' \in \mathrm{NPMV}$.

Now define $r(x) = h'(f'(x))$. It is easy to see that $r(x)$ is undefined if and only if $f(x)$ is undefined, and if $r(x)$ maps to y, then $f(x)$ maps to y. Therefore, r is a refinement of f and is in $\mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[k]}/_{sv}$. This proves (1).

To prove (2) we proceed exactly as above except that instead of f' we use a new function f'' defined so that for all x and y, $f''(x)$ maps to (x, y) if and only if either

1. $y = b(x)$, or
2. $y = 0^k 1z$ for some z such that $f(x)$ maps to z.

Note that f' is a refinement of f'', so $f'' \in \mathrm{PF}^{\mathrm{NP}[k]}$ by Proposition 2.2. Define V'' to be a machine that on input (x, a) first checks if a is of the form $0^k 1z$ for some z. If so, V'' outputs z and halts. Otherwise, V'' behaves exactly as V' above. Let h'' be the function defined by V''. We have $h'' \in \mathrm{NPMV}$ as before. Now defining $r'(x) = h''(f''(x))$, we show that $r' = f$, completing the proof.

Suppose $f(x)$ maps to z. Then $f''(x)$ maps to $(x, 0^k 1z)$ and $h''(x, 0^k 1z)$ maps to z, so $r'(x)$ maps to z. Conversely, suppose $r'(x)$ maps to z. Then it must be the case that either $f''(x)$ maps to $(x, b(x))$ and $h''(x, b(x))$ maps to z, or $f''(x)$ maps to $(x, 0^k 1z)$ and $h''(x, 0^k 1z)$ maps to z. In the latter case, $f(x)$ maps to z by the definition of f''. In the former case, $V(x, b(x))$ must output $1z$, and thus $f(x)$ maps to z. □

Selman [15] showed that $\mathrm{PF}^{\mathrm{NP}[\log]} \neq \mathrm{PF}_{tt}^{\mathrm{NP}}$ unless P = FewP and R = NP.[2] The next two theorems are interesting because they imply that (i) composing on the left with NPMV is enough to absorb the difference between the two reduction classes and (ii) the NPMV analogue of Selman's result is *false*.

THEOREM 3.4. *For each* $k \geq 1$, $\mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[k]} = \mathrm{NPMV} \circ \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]}$.

Proof. $\mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[k]} \subseteq \mathrm{NPMV} \circ \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]}$ follows immediately from $\mathrm{PF}^{\mathrm{NP}[k]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]}$ [2]. (Recall the comment that follows Proposition 2.3.)

Now we show $\mathrm{NPMV} \circ \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]} \subseteq \mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[k]}$. Let $f = g \circ h \in \mathrm{NPMV} \circ \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]}$ with $g \in \mathrm{NPMV}$ and $h \in \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]}$. Let $g \in \mathrm{NPMV}$ via a nondeterministic Turing machine N and let h' be a single-valued refinement of h that is computed by a deterministic oracle Turing-machine transducer M with oracle $A \in \mathrm{NP}$. Define s to be a function that maps x to the number of queries in A that M

[2] Actually, it was shown there that $\mathrm{PF}^{\mathrm{NP}[\log]}/_{sv} \neq \mathrm{PF}_{tt}^{\mathrm{NP}}/_{sv}$ unless P = FewP and R = NP, but the two forms of the statement are equivalent by Proposition 2.3. This result also follows by directly modifying a proof of Beigel [1, Theorem 9].

makes on input x. s is a total, single-valued function. Define $B = \{\langle x, n\rangle \mid s(x) \geq n\}$. Clearly B belongs to NP. Since s is total and $0 \leq s(x) \leq 2^k - 1$, a binary search algorithm over $\{1, \ldots, 2^k - 1\}$ computes s in polynomial time with oracle B. The number of queries is exactly k, so $s \in \mathrm{PF}^{\mathrm{NP}[k]}$. Define s' to be a multivalued function that maps x to $\langle x, n\rangle$ if and only if either

1. $n = s(x)$, or
2. $n = 2^k + w$ for some w such that $h(x)$ maps to w.

Clearly, s' has a refinement in $\mathrm{PF}^{\mathrm{NP}[k]}$, so $s' \in \mathrm{PF}^{\mathrm{NP}[k]}$ by Proposition 2.2.

Let $A \in \mathrm{NP}$ be witnessed by a machine D and define E to be the machine that on input $\langle x, n\rangle$ behaves as follows:

(1) If $n \geq 2^k$, then E sets $w = n - 2^k$ and goes to step (6).
(2) E computes the set Q of all query strings of M on x.
(3) E nondeterministically guesses $R \subseteq Q$ of size n.
(4) For each $y \in R$, E simulates D on y. If D on y does not accept for some $y \in R$, then E halts without accepting.
(5) E simulates M on x answering a query $q \in Q$ affirmatively if and only if $q \in R$. If M halts without accepting, so does E. Otherwise (M has an output), E computes the output w.
(6) E simulates N on w. If N outputs a string z, then so does E, and if it does not accept, then E halts without accepting.

Let $t \in \mathrm{NPMV}$ denote the partial multivalued function that E computes. We fix x and z in what follows. Suppose $f(x)$ maps to z. Then by definition there is a w such that $h(x) \mapsto w$ and $g(w) \mapsto z$, but in this case we know that $s'(x) \mapsto \langle x, 2^k + w\rangle$ and $t(x, 2^k + w)$ simulates N on w so $t(x, 2^k + w)$ maps to z. Thus $t \circ s'$ maps x to z.

Conversely, suppose $s'(x)$ maps to some $\langle x, n\rangle$ and $t(x, n)$ maps to z. If $n \geq 2^k$, then it must be that both $n = 2^k + w$ such that $h(x) \mapsto w$, and $g(w)$ maps to z ($t(x, n)$ just simulates N on $w = n - 2^k$). Thus $f(x)$ maps to z in this case. If $n \leq 2^k - 1$, then $n = s(x)$. We have $t(x, s(x)) \mapsto z$ if and only if there exist w and a set $R \subseteq A$ consisting of $s(x)$ query strings of M on x such that (i) given affirmative answers to all strings in R and negative answers to all strings in $Q - R$, M on x computes w and (ii) N on w outputs z on some computation path. Since $s(x)$ is exactly the number of query strings in A, $t(x, s(x)) = g(h'(x))$. Since $g \circ h'$ is a refinement of f, if $t(x, s(x))$ maps to z, then $f(x)$ maps to z.

Therefore, $f = t \circ s' \in \mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[k]}$. □

Remark. If we restrict s' in the above proof so that $s'(x)$ only maps to $\langle x, s(x)\rangle$, then s' is single-valued and $t \circ s'$ is a refinement of f. This shows that $\mathrm{NPMV} \circ \mathrm{PF}_{tt}^{\mathrm{NP}[2^k-1]} \subseteq_c \mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[k]}/_{sv})$.

The left-to-right inclusion in the next theorem is completely analogous with the NP case, given by Beigel [2].

THEOREM 3.5. *For every $k \geq 1$, $\mathrm{PF}^{\mathrm{NPMV}[k]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]}$.*

Proof. Let $f \in \mathrm{PF}^{\mathrm{NPMV}[k]}$ be computed by a deterministic oracle Turing-machine transducer M with $g \in \mathrm{NPMV}$ as the oracle. Let $h = s[M, g]$ in Proposition 3.1. Define D to be the machine with oracle h that on input x behaves as follows:

(1) D deterministically computes $w(0^k) = h(x, 0^k)$.
(2) For each $a \in \Sigma^k - \{0^k\}$, D sets $w(a)$ to the answer to $h(x, a)$. The number of queries to h is $2^k - 1$.
(3) D sets b to the largest a such that $w(a) \neq \perp$.
(4) If $w(b) = 0$, then D rejects x, and if $w(b) = 1y$ for some y, then D outputs y.

By Proposition 3.1, for every x, $D(x)$ rejects if and only if $x \notin dom(f)$, and if $D(x)$ outputs y, then $f(x) \mapsto y$. So D computes a refinement of f. Thus $f \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]}$. This proves that $\mathrm{PF}^{\mathrm{NPMV}[k]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]}$.

Now let $f \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]}$ be computed by a deterministic oracle Turing machine transducer M with $g \in \mathrm{NPMV}$ as the oracle. Let $h = s[M, g]$ in Proposition 3.2. Define D to be the machine with oracle h that on input x behaves as follows:

(1) D deterministically computes $w(0) = h(x, 0)$.

(2) By using a binary search over $[1, 2^k - 1]$ with oracle h, D computes $m = \max\{n \in \{0, \ldots, 2^k - 1\} \mid h(x, n) \text{ has an output}\}$. While doing this, D keeps the answers obtained from the oracle and sets $w(m)$ to the answer to $h(x, m)$.

(3) If $w(m) = 0$, then D rejects x, and if $w(m) = 1y$ for some y, then D outputs y.

By Proposition 3.2, for every x, $D(x)$ rejects if and only if $x \notin dom(f)$, and if $D(x)$ outputs y, then $f(x) \mapsto y$. So D computes a refinement of f. Since the number of queries to h is k, we have $f \in \mathrm{PF}^{\mathrm{NPMV}[k]}$. Thus $\mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]} \subseteq \mathrm{PF}^{\mathrm{NPMV}[k]}$, which proves the theorem. □

The above theorems yield the following corollary.

COROLLARY 3.6. *For every $k \geq 1$, $\mathrm{PF}^{\mathrm{NPMV}[k]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]} \subseteq_c \mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[k]}/_{sv}) \subseteq \mathrm{PF}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[2^{k+1}-1]}$.*

By Proposition 2.2, $\mathrm{PF}^{\mathrm{NPMV}[k]} \subseteq \mathrm{PF}^{\mathrm{NPMV}[k+1]}$. For general bounded query classes, it is not known whether $\mathrm{PF}^{\mathrm{NPMV}[k]} \subseteq \mathrm{PF}^{\mathrm{NP}[k]}$. But for reduction classes of sets, this type of equivalence holds.

THEOREM 3.7. *For every $k \geq 1$, $\mathrm{P}^{\mathrm{NPMV}[k]} = \mathrm{P}^{\mathrm{NP}[k]}$.*

Proof. Let $k \geq 1$. It suffices to show that $\mathrm{P}^{\mathrm{NPMV}[k]} \subseteq \mathrm{P}^{\mathrm{NP}[k]}$. Let $A \in \mathrm{P}^{\mathrm{NPMV}[k]}$. Then $f = \chi_A$ is in $\mathrm{PF}^{\mathrm{NPMV}[k]}$. Let M and $g \in \mathrm{NPMV}$ be a machine and a partial multivalued function witnessing this property, respectively. Informally, we will show that $f \in \mathrm{P}^{\mathrm{NP}[k]}$ by using Proposition 3.1 to compute f. Namely, by letting $h = s[M, g]$ be the NPMV function given in Proposition 3.1 and letting a_x be the largest a in Σ^k such that $h(x, a)$ is defined, we will show that k queries to an NP oracle suffice both to find a_x and to compute $h(x, a_x) = 1f(x)$. Assume without loss of generality that M always outputs exactly one bit for all oracles. For simplicity, we fix x in the following discussion.

Let a_x be the largest $a \in \Sigma^k$ such that $h(x, a)$ is defined. The function $\lambda x.[h(x, a_x)]$ is total and single-valued, $x \in A$ if and only if $h(x, a_x) = 11$, and $x \notin A$ if and only if $h(x, a_x) = 10$. Let $b \in \Sigma$ such that $1b = h(x, 0^k)$ and ρ_x be the largest $r \in \Sigma^{k-1}$ such that either $h(x, r0)$ or $h(x, r1)$ maps to $1b$. It is not hard to see that $h(x, a_x) = 1b$ if and only if

(a) for every $a > \rho_x 1$, $h(x, a)$ maps to neither 10 nor 11, and

(b) $h(x, \rho_x 1)$ does not map to $1b'$, where $b' = 1$ if $b = 0$ and 0 otherwise.

Since $\rho_x \geq 0^{k-1}$ and $graph(h) \in \mathrm{NP}$, it is easy to see that ρ_x is computed by making $k-1$ questions to an NP oracle: we perform a binary search over Σ^{k-1} in order to find the largest $r \in \Sigma^{k-1}$ such that either $h(x, r0) \mapsto 1b$ or $h(x, r1) \mapsto 1b$. After ρ_x is found, conditions (a) and (b) can be tested by a single question to an NP oracle. Therefore, by making k queries to an NP oracle, $h(x, a_x)$ is computed. Since $h(x, a_x) = 11$ if and only if $x \in A$, this implies that $A \in \mathrm{P}^{\mathrm{NP}[k]}$. This proves the theorem. □

Note that Theorem 3.7 holds even if k is replaced by any polynomially bounded function. This means, remarkably, that in *any* polynomial-time computation of a set

relative to NPMV, the queries to NPMV can be replaced one for one with queries to NP.

THEOREM 3.8. *For every* $k \geq 1$, $\mathrm{P}_{tt}^{\mathrm{NPMV}[k]} = \mathrm{P}_{tt}^{\mathrm{NP}[k]}$.

Proof. Let $k \geq 1$. It suffices to show that $\mathrm{P}_{tt}^{\mathrm{NPMV}[k]} \subseteq \mathrm{P}_{tt}^{\mathrm{NP}[k]}$. Let $A \in \mathrm{P}_{tt}^{\mathrm{NPMV}[k]}$. Then $f = \chi_A$ is in $\mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$. Let M and $g \in \mathrm{NPMV}$ be a machine and a partial multivalued function witnessing this property, respectively. We assume without loss of generality that M outputs 0 or 1 for all oracles and inputs. We will use Proposition 3.2 to compute f. Namely, we let $h = s[M, g]$ be the NPMV function given in Proposition 3.2, and fixing an input x, we let n_x be the largest n in $\{0, \ldots, k\}$ such that $h(x, n)$ is defined. We will show that k nonadaptive queries to an NP oracle suffice to compute $h(x, n_x) = 1f(x)$. The informal idea of the proof is as follows: we first compute b, the output of $M(x)$ where all query answers are $\perp$. Then we use nonadaptive NP queries to inspect sequences of query answers to $M(x)$ which are *plausible*, i.e., where all non-$\perp$ answers are verifiably legitimate outputs of the oracle function but where each $\perp$ answer may or may not be correct. We look for m, the largest number of non-$\perp$ query answers in any plausible sequence of query answers where M outputs b. Actually, computing m exacty uses all our allotted NP queries without telling us the real output of M, so instead we only compute which of the pairs $\{2i, 2i+1\}$ m belongs to. This uses only about $k/2$ nonadaptive queries. Simultaneously, for each pair $\{2i, 2i+1\}$, we ask NP if *either* there is any plausible sequence with more than $2i+1$ non-$\perp$ answers *or* M outputs $1-b$ for some plausible sequence containing exactly $2i+1$ non-$\perp$ answers. The answer to this NP question for the pair containing m immediately tells us whether M outputs b or $1-b$ for any correct sequence of query answers. This uses up the remaining $k/2$ NP queries.

We now present an exact version of the sketch above, showing that $f \in \mathrm{P}_{tt}^{\mathrm{NP}[k]}$. Note that the value of $h(x, 0)$ encodes the output of $M(x)$ with all queries answered with $\perp$, and $h(x, n)$ for $n > 0$ simply encodes the possible outputs of $M(x)$ over all plausible sequences of queries with at least n non-$\perp$ answers. Also, n_x is the exact number of non-$\perp$ entries in any sequence of correct query answers. We are assuming that M outputs exactly one bit for all possible sets of query answers, so h outputs nothing but 10 or 11.

By Proposition 3.2, $\lambda x.[h(x, n_x)]$ is total and single-valued, $x \in A$ if and only if $h(x, n_x) = 11$, and $x \notin A$ if and only if $h(x, n_x) = 10$. Let $b \in \Sigma$ be such that $1b = h(x, 0)$, and $d = \lfloor k/2 \rfloor$. Define two predicates S and T as follows:

$S(x, r)$ if and only if either $h(x, 2r)$ or $h(x, 2r+1)$ maps to $1b$.

$T(x, r)$ if and only if

- $h(x, n)$ is defined for some $n > 2r+1$, or
- $h(x, 2r+1)$ maps to $1b'$, where $b' = 1 - b$.

Note that S and T are NP-predicates, $S(x, 0) = \mathit{true}$, and if k is even, then $T(x, d) = \mathit{false}$. Define ρ_x to be the largest $r \in \{0, \ldots, d\}$ such that $S(x, r) = \mathit{true}$. It is not hard to see that $h(x, n_x) = 1b$ if and only if $T(x, \rho_x) = \mathit{false}$.

Our goal is to compute $h(x, n_x)$ without making more than k queries to some NP oracle. Our method to accomplish this is to partition the domain $\{0, \ldots, k\}$ into successive pairs. For each pair $\{2r, 2r+1\}$, we make two queries of the form "$S(x, r) = \mathit{true}$?" and "$T(x, r) = \mathit{true}$?". As observed above, $S(x, 0) = \mathit{true}$, and if k is even, $T(x, d) = \mathit{false}$. So we need exactly k queries to S or T. Since S and T are NP-predicates, a single set in NP can answer both types of questions. Thus by making k nonadaptive queries to an NP oracle, we determine whether $h(x, n_x) = 1b$ or not. Since $h(x, n_x) = 11$ if and only if $x \in A$, this implies that $A \in \mathrm{P}_{tt}^{\mathrm{NP}[k]}$. This

proves the theorem. □

Our next goal is to show that bounded query hierarchies probably do not collapse.

LEMMA 3.9. *Let $k \geq 1$. If $\mathrm{PF}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}^{\mathrm{NPMV}[k]}$, then for every $\ell \geq k$, $\mathrm{PF}^{\mathrm{NPMV}[\ell]} = \mathrm{PF}^{\mathrm{NPMV}[k]}$.*

Proof. Let k be as in the hypothesis, let $m \geq 0$, and let $f \in \mathrm{PF}^{\mathrm{NPMV}[k+1+m]}$ be computed by a deterministic oracle Turing-machine transducer M with $g \in \mathrm{NPMV}$ as the oracle. We will show that $f \in \mathrm{PF}^{\mathrm{NPMV}[k+m]}$.

Let N be an oracle transducer that on input x with oracle g outputs an ID of M on x just after obtaining the answer to its mth query to g. Clearly, N makes at most m queries to g, and if g is single-valued, then $N[g]$ is total and single-valued. Define D to be the machine such that, given an ID I of M, (1) D attempts to simulate the computation of M starting from ID I by making at most $k+1$ queries to its oracle, and (2) if M halts without an output, then so does D and if M outputs a string y, then so does D.

It is not hard to see that D with oracle g defines a function r such that for every x and single-valued refinement g' of g, $f(x)$ is defined if and only if r maps the ID $N[g'](x)$ to some output y. And if the latter holds, then $f(x)$ also maps to y. Moreover, it is clear that $r \in \mathrm{PF}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}^{\mathrm{NPMV}[k]}$; thus there is an oracle transducer Q such that for every single-valued refinement g' of g, $Q[g']$ computes a refinement of r making only k queries to g'. Now we combine the machines N and Q by taking the output of N as the input to Q. The resulting machine with oracle g' computes a refinement of f and makes at most $k+m$ queries to g'. Thus $f \in \mathrm{PF}^{\mathrm{NPMV}[k+m]}$ by Proposition 2.2.

By applying this argument repeatedly, we have $\mathrm{PF}^{\mathrm{NPMV}[\ell]} = \mathrm{PF}^{\mathrm{NPMV}[k]}$ for every $\ell \geq k$. □

LEMMA 3.10. *Let $k \geq 1$. If $\mathrm{PF}_{tt}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$, then for every $\ell \geq k$, $\mathrm{PF}_{tt}^{\mathrm{NPMV}[\ell]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$.*

Proof. Let $k \geq 1$ and suppose that $\mathrm{PF}_{tt}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$. Let $m \geq 0$ and $f \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[k+1+m]}$. We will show that $f \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[k+m]}$. By applying the argument repeatedly, for every $\ell \geq k$, we have $\mathrm{PF}_{tt}^{\mathrm{NPMV}[\ell]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$.

Let $f \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[k+1+m]}$ be computed by a deterministic oracle Turing-machine transducer M with $g \in \mathrm{NPMV}$ as the oracle. For x, let $q_1(x), \ldots, q_{k+1+m}(x)$ denote the queries of M on x. Define c to be the function that maps x to $(y_1, \ldots, y_{k+1})$ so that for every $i, 1 \leq i \leq k+1$, y_i is a value of $g(q_i(x))$ if $q_i(x) \in \mathit{dom}(g)$ and $y_i = \perp$ otherwise. Obviously, $c \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[k+1]}$, so by our supposition, $c \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$. Let c be computed by a deterministic oracle Turing-machine transducer N with $h \in \mathrm{NPMV}$ as the oracle. Then we can easily construct a machine that computes f by making k queries to h and m queries to g. Therefore, $f \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[k+m]}$. This proves the lemma. □

The Boolean hierarchy over NP is defined by Wagner and Wechsung [19] and has been studied extensively [6, 7, 8, 12]. We denote the kth level of the Boolean hierarchy as $\mathrm{NP}(k)$. By definition,

1. $\mathrm{NP}(1) = \mathrm{NP}$, and
2. for every $k \geq 2$, $\mathrm{NP}(k) = \mathrm{NP} - \mathrm{NP}(k-1)$.

The Boolean hierarchy over NP, denoted by BH, is the union of all $\mathrm{NP}(k)$, $k \geq 1$.

Kadin [12] proved that the Boolean hierarchy collapses only if the polynomial-time hierarchy collapses.

THEOREM 3.11. *Let $k \geq 1$. If $\mathrm{PF}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}^{\mathrm{NPMV}[k]}$, then* BH *collapses to*

its 2^kth level.

Proof. Suppose that $\mathrm{PF}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}^{\mathrm{NPMV}[k]}$. By Lemma 3.9 and Theorem 3.5, for every $m > k$, $\mathrm{PF}^{\mathrm{NPMV}[m]} \subseteq \mathrm{PF}^{\mathrm{NPMV}[k]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]}$. So by Theorem 3.7 and results by Köbler, Schöning, and Wagner [14], we have, for every $m > k$, $\mathrm{P}^{\mathrm{NP}[m]} = \mathrm{P}_{tt}^{\mathrm{NP}[2^k-1]} \subseteq \mathrm{NP}(2^k)$. Thus $\mathrm{BH} = \mathrm{NP}(2^k)$. □

The following theorem is proved in a similar manner.

THEOREM 3.12. *Let $k \geq 1$. If $\mathrm{PF}_{tt}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$, then* BH *collapses to its $(k+1)$st level.*

Proof. Suppose that $\mathrm{PF}_{tt}^{\mathrm{NPMV}[k+1]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$. By Lemma 3.10, for every $m > k$, $\mathrm{PF}_{tt}^{\mathrm{NPMV}[m]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$. So by Theorem 3.8 and results by Köbler, Schöning, and Wagner [14], we have, for every $m > k$, $\mathrm{P}_{tt}^{\mathrm{NP}[m]} = \mathrm{P}_{tt}^{\mathrm{NP}[k]} \subseteq \mathrm{NP}(k+1)$. Thus $\mathrm{BH} = \mathrm{NP}(k+1)$. □

Analogous to the theorems stated so far, the following theorems hold for reduction classes that make logarithmic many queries to partial functions in NPMV. We see in these theorems a different behavior when computing partial multivalued functions with bounded queries to NPMV than when computing partial functions with bounded queries to NP. To wit, in contrast to the following results, Selman [15] shows that $\mathrm{PF}^{\mathrm{NP}[\log]} = \mathrm{PF}_{tt}^{\mathrm{NP}}$ only if P = FewP and R = NP. The reason seems to be, as we showed in Theorems 3.3 and 3.5, that the mind-change argument [2, 19] works for $\mathrm{PF}^{\mathrm{NPMV}}$ (as it does for P^{NP}) but apparently does not work for $\mathrm{PF}^{\mathrm{NP}}$.

THEOREM 3.13.

1. $\mathrm{PF}^{\mathrm{NPMV}[\log]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}} \subseteq \mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[\log]} = \mathrm{NPMV} \circ \mathrm{PF}_{tt}^{\mathrm{NP}}$.
2. $\mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[\log]}/_{sv}) \subseteq \mathrm{PF}^{\mathrm{NPMV}[\log]}$.
3. $\mathrm{PF}^{\mathrm{NPMV}[\log]} \subseteq_c \mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[\log]}/_{sv})$.
4. $\mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[\log]}/_{sv}) \subseteq \mathrm{NPMV} \circ (\mathrm{PF}_{tt}^{\mathrm{NP}}/_{sv})$.
5. $\mathrm{NPMV} \circ (\mathrm{PF}_{tt}^{\mathrm{NP}}/_{sv}) \subseteq_c \mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[\log]}/_{sv})$.

Proof. Note that for any function t such that $t(x) \leq c \log |x|$, $2^{t(x)} - 1$ is polynomially bounded and that for any polynomial p, $\log p(|x|) \leq c \log |x|$ for some constant c. Therefore, a proof similar to that of Theorem 3.5 shows that $\mathrm{PF}^{\mathrm{NPMV}[\log]} = \mathrm{PF}_{tt}^{\mathrm{NPMV}}$, and a proof similar to Theorem 3.3(2) shows that $\mathrm{PF}^{\mathrm{NPMV}[\log]} \subseteq \mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[\log]}$. A proof similar to that of Theorem 3.4 yields $\mathrm{NPMV} \circ \mathrm{PF}^{\mathrm{NP}[\log]} = \mathrm{NPMV} \circ \mathrm{PF}_{tt}^{\mathrm{NP}}$. Thus (1) holds.

Inclusion (2) follows by a straightforward simulation, and (3) follows by adapting the proof of Theorem 3.3(1). Using a technique of Buss and Hay [4], for any set A, logarithmically many adaptive queries to A can be simulated by polynomially many nonadaptive queries to A, so $\mathrm{PF}^{\mathrm{NP}[\log]}/_{sv} \subseteq \mathrm{PF}_{tt}^{\mathrm{NP}}/_{sv}$. Thus $\mathrm{NPMV} \circ (\mathrm{PF}^{\mathrm{NP}[\log]}/_{sv}) \subseteq \mathrm{NPMV} \circ (\mathrm{PF}_{tt}^{\mathrm{NP}}/_{sv})$. Hence (4) holds. Inclusion (5) follows by adapting the remark following the proof of Theorem 3.4. □

THEOREM 3.14. $\mathrm{P}^{\mathrm{NPMV}[\log]} = \mathrm{P}_{tt}^{\mathrm{NPMV}} = \mathrm{P}^{\mathrm{NP}[\log]} = \mathrm{P}_{tt}^{\mathrm{NP}}$.

The proof is similar to those of Theorems 3.7 and 3.8.

4. Nondeterministic polynomial-time reductions. We define nondeterministic reductions between partial functions so that the access mechanism is identical to that for deterministic reductions. Namely, let g be a single-valued partial function and N be a polynomial-time nondeterministic oracle Turing machine. $N[g]$ denotes a multivalued partial function computed by N with oracle g in accordance with the following mechanism:

- when N asks about $y \in dom(g)$, g returns $g(y)$, and

- when N asks about $y \notin dom(g)$, g answers a special symbol $\perp$.

DEFINITION 4.1. *Let f and g be partial multivalued functions. We say that f* is nondeterministic polynomial-time Turing reducible to *g, denoted by $f \leq_{\mathrm{T}}^{\mathrm{NP}} g$, if there is a polynomial-time nondeterministic Turing machine N such that for every single-valued refinement g' of g, $N[g']$ is a refinement of f.*

Thus for every x and for every single-valued refinement g' of g,

- $x \in dom(f)$ if and only if $x \in dom(N[g'])$, and
- if $N[g']$ maps x to y, then f maps x to y.

Let $\mathcal{F}$ be a class of partial multivalued functions. $\mathrm{NPMV}^{\mathcal{F}}$ denotes the class of partial multivalued functions that are $\leq_{\mathrm{T}}^{\mathrm{NP}}$-reducible to some $g \in \mathcal{F}$. $\mathrm{NPMV}^{\mathcal{F}[k]}$ denotes the class of partial multivalued functions that are $\leq_{\mathrm{T}}^{\mathrm{NP}}$-reducible to some $g \in \mathcal{F}$ via a machine that makes k adaptive queries to its oracle.

For a class of sets $\mathcal{C}$, we write $\mathrm{NPMV}^{\mathcal{C}}$ to denote the class of multivalued partial functions that are computed by a nondeterministic Turing machine relative to an oracle in $\mathcal{C}$. $\mathrm{NPMV}^{\mathcal{C}[k]}$ is defined similarly.

It is easy to see that every nondeterministic polynomial-time reduction to partial functions is replaceable by a reduction that makes nonadaptive queries to its oracle and that preserves the number of queries. For this reason, we do not distinguish classes $\mathrm{NPMV}_{tt}^{\mathcal{F}}$ or $\mathrm{NPMV}_{tt}^{\mathcal{F}[k]}$.

For $k \geq 1$, $\Sigma\mathrm{MV}_k$ denotes $\underbrace{\mathrm{NPMV}^{\cdots^{\mathrm{NPMV}}}}_{k}$.

LEMMA 4.1. *For every $k \geq 1$, $\Sigma\mathrm{MV}_k = \mathrm{NPMV}^{\Sigma_{k-1}^{\mathrm{P}}[1]}$ and for every $f \in \Sigma\mathrm{MV}_k$, $dom(f) \in \Sigma_k^{\mathrm{P}}$.*

Proof. The proof is by an induction on k. The statement trivially holds for $k = 1$. Let $k = 2$. We show that $\mathrm{NPMV}^{\mathrm{NPMV}} \subseteq \mathrm{NPMV}^{\mathrm{NP}[1]}$. Let $f \in \mathrm{NPMV}^{\mathrm{NPMV}}$ via a machine M and a function $g \in \mathrm{NPMV}$. Let N be a machine witnessing $g \in \mathrm{NPMV}$. Define A to be the set of all $(y_1, \ldots, y_m)$, $m \geq 1$, such that $y_1, \ldots, y_m \notin dom(g)$. Obviously, $A \in$ co-NP. Consider a nondeterministic Turing machine T that on input x simulates M on x in the following way:

- When M queries about string w, T simulates N on w. If N on w outputs a string z, then T assumes that the answer from the oracle is z and if N on w does not accept, then T assumes that the answer from the oracle is $\perp$.
- When M enters a halting state, T enumerates all the queries w for which the answer from the oracle are assumed to be $\perp$. T sets $y_1, \ldots, y_m$ to the enumeration.
  - —If $(y_1, \ldots, y_m) \notin A$, then T rejects,
  - —if $(y_1, \ldots, y_m) \in A$ and M rejects, then T rejects, and
  - —if $(y_1, \ldots, y_m) \in A$ and M outputs some string z, then T outputs z.

It is not hard to see that for every x and z, $f(x)$ maps to z if and only if there is a computation of M relative to g that outputs z if and only if there is a computation of T relative to A that outputs z. Thus T relative to A computes a refinement of f. Furthermore, T is polynomial-time bounded and T makes only one question to A. So $f \in \mathrm{NPMV}^{\mathrm{NP}[1]}$. Thus $\mathrm{NPMV}^{\mathrm{NPMV}} = \mathrm{NPMV}^{\mathrm{NP}[1]}$. Moreover, by the above discussions, for every function $f \in \mathrm{NPMV}^{\mathrm{NPMV}}$, $dom(f) \in \Sigma_2^{\mathrm{P}}$.

Now let $k \geq 2$ and suppose that the claim holds for every $k' \leq k$. Since the above proof is relativizable, for any class $\mathcal{C}$ of sets, we have $\mathrm{NPMV}^{\mathrm{NPMV}^{\mathcal{C}}} \subseteq \mathrm{NPMV}^{\mathrm{NP}^{\mathcal{C}}[1]}$. So $\mathrm{NPMV}^{\Sigma\mathrm{MV}_k} = \mathrm{NPMV}^{\mathrm{NPMV}^{\Sigma_{k-1}^{\mathrm{P}}[1]}}$ (by induction hypothesis) $\subseteq \mathrm{NPMV}^{\mathrm{NPMV}^{\Sigma_{k-1}^{\mathrm{P}}}} \subseteq$

$\mathrm{NPMV}^{\mathrm{NP}^{\Sigma_{k-1}^{\mathrm{P}}}[1]} = \mathrm{NPMV}^{\Sigma_k^{\mathrm{P}}[1]}$. This proves the lemma. □

This lemma yields the following theorem.

THEOREM 4.2. *Let f be a partial multivalued function. For every $k \geq 1$, the following statements are equivalent:*

(i) *f is in $\Sigma\mathrm{MV}_k$;*

(ii) *f has a polynomially length-bounded refinement g such that $dom(g) \in \Sigma_k^{\mathrm{P}}$, and $graph(g) \in \Sigma_k^{\mathrm{P}}$;*

(iii) *f has a polynomial length-bounded refinement g such that $graph(g) \in \Sigma_k^{\mathrm{P}}$.*

THEOREM 4.3. *For every $k \geq 1$, $\Sigma\mathrm{MV}_{k+1} = \Sigma\mathrm{MV}_k$ if and only if $\Sigma_{k+1}^{\mathrm{P}} = \Sigma_k^{\mathrm{P}}$.*

Proof. First, suppose that $\Sigma_{k+1}^{\mathrm{P}} = \Sigma_k^{\mathrm{P}}$. Let f be any function in $\Sigma\mathrm{MV}_{k+1}$. By Theorem 4.2(ii), there is a polynomially length-bounded refinement g of f such that $dom(g), graph(g) \in \Sigma_{k+1}^{\mathrm{P}}$, so by our supposition, $dom(g), graph(g) \in \Sigma_k^{\mathrm{P}}$. Therefore, f is in $\Sigma\mathrm{MV}_k$. Hence $\Sigma\mathrm{MV}_{k+1} = \Sigma\mathrm{MV}_k$.

Next, suppose that $\Sigma\mathrm{MV}_{k+1} = \Sigma\mathrm{MV}_k$. Let A be $\leq_m^{\mathrm{P}}$-complete for $\Sigma_{k+1}^{\mathrm{P}}$. Define χ_A^0 to be the function that $\chi_A^0(x) = 1$ if $x \in A$ and undefined otherwise. Obviously, χ_A^0 is in $\Sigma\mathrm{MV}_{k+1}$, so by our supposition, $\chi_A^0 \in \Sigma\mathrm{MV}_k$. On the other hand, $dom(\chi_A^0) = A$. Thus we have $A \in \Sigma_k^{\mathrm{P}}$. Since A is complete for $\Sigma_{k+1}^{\mathrm{P}}$, we have $\Sigma_{k+1}^{\mathrm{P}} = \Sigma_k^{\mathrm{P}}$. □

Thus these classes form function analogues of the polynomial hierarchy, and, unless the polynomial hierarchy collapses, they form a proper hierarchy.

5. The difference hierarchy. Let $\mathcal{F}$ be a class of partial multivalued functions. A partial multivalued function f is in co$\mathcal{F}$ if there exist $g \in \mathcal{F}$ and a polynomial p such that for every x and y,

- $f(x)$ maps to y if and only if $|y| \leq p(|x|)$ and $g(x)$ does not map to y.

Let $\mathcal{F}$ and $\mathcal{G}$ be two classes of partial multivalued functions. A partial multivalued function h is in $\mathcal{F} \wedge \mathcal{G}$ if there exist partial multivalued functions $f \in \mathcal{F}$ and $g \in \mathcal{G}$ such that for every x and y,

- $h(x)$ maps to y if and only if $f(x)$ maps to y and $g(x)$ maps to y.

A partial multivalued function h is in $\mathcal{F} \vee \mathcal{G}$ if there exist partial multivalued functions $f \in \mathcal{F}$ and $g \in \mathcal{G}$ such that for every x and y,

- $h(x)$ maps to y if and only if $f(x)$ maps to y or $g(x)$ maps to y.

$\mathcal{F} - \mathcal{G}$ denotes $\mathcal{F} \wedge \mathrm{co}\mathcal{G}$.

NPMV(k) is the class of partial multivalued functions defined in the following way:

1. NPMV(1) = NPMV, and
2. for $k \geq 2$, NPMV(k) = NPMV − NPMV($k-1$).

LEMMA 5.1. *For every $k \geq 1$, $f \in \mathrm{NPMV}(k)$ if and only if f is polynomially length-bounded and $graph(f) \in \mathrm{NP}(k)$.*

Proof. The proof is by an induction on k. For $k = 1$, the claim trivially holds. Let $k \geq 2$ and suppose that the claim holds for all $k' < k$. Let $f \in \mathrm{NPMV}(k)$. There are functions $g \in \mathrm{NPMV}$ and $h \in \mathrm{NPMV}(k-1)$ such that for every x and y, f maps x to y if and only if g maps x to y but h does not map x to y. So $graph(f) = graph(g) - graph(h)$. By our hypothesis, $graph(h)$ belongs to NP($k-1$). Since $graph(g) \in \mathrm{NP}$, we have $graph(f) = \mathrm{NP}(k)$.

On the other hand, let f be a polynomially length-bounded function whose graph is in NP(k). There are sets $A \in \mathrm{NP}$ and $B \in \mathrm{NP}(k-1)$ such that $graph(f) = A - B$. Define g and h to be functions such that $graph(g) = A$ and $graph(h) = B$. By our hypothesis, $g \in \mathrm{NPMV}$ and $h \in \mathrm{NPMV}(k-1)$. Therefore, $f \in \mathrm{NPMV}(k)$. This proves the lemma. □

We use the above lemma to obtain the following theorem.

THEOREM 5.2. *For every* $k \geq 1$, $\mathrm{NPMV}(k+1) = \mathrm{NPMV}(k)$ *if and only if* $\mathrm{NP}(k+1) = \mathrm{NP}(k)$.

Despite the similarity in appearance, the difference hierarchy over NPMV is probably much stronger than both the difference hierarchy over NP and the bounded query hierarchy over NPMV. For example, it is well known that *maxsat* is complete for $\mathrm{PF}^{\mathrm{NP}} = \mathrm{PF}^{\mathrm{NPMV}}$ [13]. Nonetheless, we have the following.

PROPOSITION 5.3. $maxsat \in \mathrm{NPMV}(2)$.

Proof. Let $f \in \mathrm{NPMV}$ be the function that maps x to y if and only if there is a $z > y$ such that z is a satisfying assignment for x. Clearly, for all x, $maxsat(x) = y$ if and only if $sat(x)$ maps to y and $f(x)$ does not map to y. Therefore, $maxsat \in \mathrm{NPMV}(2)$. □

PROPOSITION 5.4. $\mathrm{co}(\text{co-NPMV}) = \mathrm{NPMV}$.

Proof. Let $f \in \mathrm{NPMV}$. Let p be a polynomial such that for every x and y, if $f(x)$ maps to y, then $|y| \leq p(|x|)$. Let g be the complement of f with respect to p such that for every x and y, $g(x)$ maps to y if and only if $f(x)$ does not map to y and $|y| \leq p(|x|)$. Furthermore, let h be the complement of g with respect to p such that for every x and y, $h(x)$ maps to y if and only if $g(x)$ does not map to y and $|y| \leq p(|x|)$. For every x and y, $h(x)$ maps to y if and only if $f(x)$ maps to y. This implies $h = f$. Therefore, $\mathrm{NPMV} \subseteq \mathrm{co}(\text{co-NPMV})$.

Conversely, let $f \in \mathrm{co}(\text{co-NPMV})$. There exist $g \in \text{co-NPMV}$ and a polynomial p such that for every x and y, $f(x)$ maps to y if and only if $g(x)$ does not map to y and $|y| \leq p(|x|)$. Moreover, since $g \in \text{co-NPMV}$, there exist $h \in \mathrm{NPMV}$ and a polynomial q such that for every x and y, $g(x)$ maps to y if and only if $h(x)$ does not map to y and $|y| \leq q(|x|)$. For every x and y, $f(x)$ maps to y if and only if either ($h(x)$ maps to y and $|y| \leq p(|x|)$) or ($q(|x|) < |y| \leq p(|x|)$). Let M be a nondeterministic Turing machine that computes h. Define N to be the machine that on input x (1) nondeterministically guesses $b \in \{0,1\}$, (2) if $b = 0$, then simulates M on input x and outputs y if M outputs y and $|y| \leq p(|x|)$, and (3) if $b = 1$, then nondeterministically guesses $y, q(|x|) < |y| \leq p(|x|)$ and outputs y. Clearly, N computes f. So $\mathrm{co}(\text{co-NPMV}) \subseteq \mathrm{NPMV}$. □

THEOREM 5.5. *The following statements are all equivalent.*

(a) NP = co-NP.
(b) NPMV ⊆ co-NPMV.
(c) co-NPMV ⊆ NPMV.

Proof. By Proposition 5.4, (b) is equivalent to (c). So it suffices to show that (a) is equivalent to (c). First, suppose that co-NPMV ⊆ NPMV. Define f to be the function that maps x to each of the three strings $\lambda, 0$, and 1 if $x \in \mathrm{SAT}$ and undefined otherwise. Obviously, $f \in \mathrm{NPMV}$. Let p be a polynomial such that $p(n) = 1$ for all n. By taking the complement of f with respect to p, we obtain a function $g \in \text{co-NPMV}$ that maps x to $\lambda, 0, 1$ if $x \notin \mathrm{SAT}$ and undefined otherwise. So $dom(g) = \overline{\mathrm{SAT}}$. Now by our supposition, we have $g \in \mathrm{NPMV}$, so $dom(g) \in \mathrm{NP}$. This implies $\overline{\mathrm{SAT}} \in \mathrm{NP}$, and thus NP = co-NP.

Conversely, suppose that NP = co-NP. Let $f \in \text{co-NPMV}$. There exist $g \in \mathrm{NPMV}$ and a polynomial p such that for every x and y, $f(x)$ maps to y if and only if $g(x)$ does not map to y and $|y| \leq p(|x|)$. The set of all (x, y) such that $g(x)$ does not map to y and $|y| \leq p(|x|)$ is in co-NP, so by our supposition, it is in NP. Thus $graph(f) \in \mathrm{NP}$, so $f \in \mathrm{NPMV}$. Hence co-NPMV ⊆ NPMV. □

Define a function f to be NPMV-*low* if $\mathrm{NPMV}^f \subseteq_c \mathrm{NPMV}$.

THEOREM 5.6. *A function f is* NPMV*-low if and only if $f \in_c$ NPMV with $dom(f) \in$ NP $\cap$ co-NP.*

Proof. Let f be NPMV-low. Since $f \in \text{NPMV}^f$ and $\text{NPMV}^f \subseteq_c$ NPMV, $f \in_c$ NPMV. So $dom(f) \in$ NP. Let $A = \overline{dom(f)}$. We wish to show that A belongs to NP. Define M to be a machine that on input x queries $f(x)$ and outputs 1 if $f(x) = \perp$ and 0 otherwise. The function h that M computes is the characteristic function of A. Since f is NPMV-low and h is single-valued, $h \in$ NPMV, so $A \in$ NP.

Conversely, let $f \in_c$ NPMV with $dom(f) \in \text{NP} \bigcap \text{co-NP}$. Define A to be the set of all $(y_1, \ldots, y_m), m \geq 1$, such that $y_1, \ldots, y_m \notin dom(f)$. By our supposition, $A \in \text{NP} \bigcap \text{co-NP}$. Let $g \in \text{NPMV}^f$ via a machine M. By the proof of Lemma 4.1, there is a polynomial-time nondeterministic Turing machine N that computes a refinement of g by making at most one query to A per computation path. Since $A \in \text{NP} \bigcap \text{co-NP}$, the query to A can be simulated nondeterministically. So $g \in_c$ NPMV. Therefore, f is NPMV-low. □

PROPOSITION 5.7. *For every $k \geq 1$ and $f \in \text{NPMV}(k)$, $dom(f) \in \Sigma_2^{\text{P}}$.*

Proof. By Lemma 5.1, f is polynomially length bounded and $graph(f) \in \text{NP}(k) \subseteq \Sigma_2^{\text{P}}$. There is a polynomial p such that for all x,

$$x \in dom(f) \leftrightarrow (\exists y)[|y| \leq p(|x|) \wedge y \in graph(f)].$$

Thus $dom(f) \in \Sigma_2^{\text{P}}$. □

We show next that Proposition 5.7 is tight, even when $k = 2$.

PROPOSITION 5.8. *Let A be in Σ_2^{P} via a polynomial p and a set B in* co-NP *so that for every x,*

$$x \in A \leftrightarrow (\exists y : |y| \leq p(|x|))[(x, y) \in B].$$

Let f be a function such that for every x and y,

$$f(x) \mapsto y \leftrightarrow |y| \leq p(|x|) \wedge (x, y) \in B.$$

Then $f \in \text{NPMV}(2)$. (Note that $dom(f) = A$.)

Proof. Let A, p, B, and f be as in the hypothesis. Define f_1 to be a function that maps x to each string y in $\Sigma^{\leq p(|x|)}$, and define f_2 to be a function that maps x to each string y in $\Sigma^{\leq p(|x|)}$ such that $(x, y) \notin B$. Obviously, $f_1, f_2 \in$ NPMV. For every x and y, $f(x)$ maps to y if and only if $f_1(x)$ maps to y and $f_2(x)$ does not map to y. So $f \in \text{NPMV}(2)$. □

By Theorem 5.2, the difference hierarchy for partial multivalued functions rises or falls in accordance with the difference hierarchy for sets. Since the difference hierarchy for sets sits entirely within Δ_2^{P}, one might anticipate that the NPMV(k) hierarchy lies within the second level of the polynomial-time hierarchy for partial multivalued functions. The following striking theorem shows that this can be true if and only if the polynomial-time hierarchy collapses.

THEOREM 5.9. $\text{NPMV}(2) \subseteq \text{PF}^{\text{NPMV}}$ *if and only if* $\Sigma_2^{\text{P}} = \Delta_2^{\text{P}}$.

Proof. First, suppose that $\text{NPMV}(2) \subseteq_c \text{PF}^{\text{NPMV}}$. Let A be in Σ_2^{P}. By the above proposition, there is a function $f \in \text{NPMV}(2)$ such that $dom(f) = A$. By our supposition, $f \in_c \text{PF}^{\text{NPMV}}$. So there exist a polynomial-time deterministic Turing machine M and a function $g \in$ NPMV such that for every x, $x \in A$ if and only if $M(x)$ relative to g has an output. By modifying M slightly, we have a machine M' such that for every x, $M'(x)$ relative to g outputs 1 if $x \in A$ and 0 otherwise. Thus $A \in \text{P}^{\text{NPMV}}$, and thus $A \in \text{P}^{\text{NP}}$. Hence $\Sigma_2^{\text{P}} = \Delta_2^{\text{P}}$.

Next, suppose that $\Sigma_2^{\mathrm{P}} = \Delta_2^{\mathrm{P}}$. Let $f \in \mathrm{NPMV}(2)$. By Lemma 5.1, there exist $f_1, f_2 \in \mathrm{NPMV}$ such that $graph(f) = graph(f_1) - graph(f_2)$. Define A to be the set of all pairs (x, y) for which there is some $z \geq y$ such that $(x, z) \in graph(f_1) - graph(f_2)$. Define g to be the partial function that maps x to the largest y such that $(x, y) \in A$ if $x \in dom(f)$. It is not hard to see that g is a single-valued refinement of f and g is polynomial-time computable with oracle A by an obvious binary search algorithm. By definition, $A \in \Sigma_2^{\mathrm{P}}$. So $A \in \Delta_2^{\mathrm{P}}$. Therefore, $g \in \mathrm{PF}^{\Delta_2^{\mathrm{P}}} = \mathrm{PF}^{\mathrm{P}^{\mathrm{NP}}} = \mathrm{PF}^{\mathrm{NP}}$. Thus $f \in_c \mathrm{PF}^{\mathrm{NPMV}}$. □

Theorem 5.9 raises a question: "How powerful is the difference hierarchy?" The following results provide some answers to this question.

LEMMA 5.10. $\mathrm{PF}_{tt}^{\mathrm{NPMV}[k]} \subseteq_c \mathrm{NPMV}(2k+1)$.

Proof. Let $f \in \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$ via a polynomial-time deterministic Turing machine M that makes nonadaptive queries to a function $g \in \mathrm{NPMV}$. Let p be a polynomial such that for every x, each query string of M on x is of length $\leq p(|x|)$. Let $h = s[M, g]$ defined in Proposition 3.2. For each x, let n_x be the largest $n \in \{0, \ldots, k\}$ such that $h(x, n)$ is defined. Then for every x,

- either $h(x, n_x)$ maps only to 0 or $h(x, n_x)$ maps only to strings of the form $1z$, and
- if $h(x, n_x)$ maps to 0, then $x \notin dom(f)$ and if $h(x, n_x)$ maps to some $1z$, then f maps x to z.

For each $i \in \{0, \ldots, k\}$, define H_i to be the function that maps x to y if and only if $h(x, i)$ maps to $1y$. For each $i \in \{1, \ldots, k\}$, define G_i to be the function that maps x to each string in $\Sigma^{\leq p(|x|)}$ if for some $j > i$, $(x, j) \in dom(h)$ and undefined otherwise. These functions are obviously in NPMV. Note the following:

(1) for every $i > n_x$, $H_i(x)$ is undefined;
(2) for every $i \geq n_x$, $G_i(x)$ is undefined; and
(3) for every $i < n_x$, $G_i(x)$ maps to every $y \in \Sigma^{\leq p(|x|)}$.

Define F by

$$graph(F) = graph(H_k) \cup \left(\bigcup\nolimits_{0 \leq i \leq k-1} (graph(H_i) - graph(G_i)) \right).$$

Then $dom(f) = dom(F)$ and if $F(x)$ maps to y, then $f(x)$ maps to y. Therefore, F is a refinement of f. Since H_i and G_i are in NPMV,

$$graph(F) \in \mathrm{NP} \vee \underbrace{\mathrm{NP}(2) \vee \cdots \vee \mathrm{NP}(2)}_{k}.$$

So $graph(F) \in \mathrm{NP}(2k+1)$. Therefore, $f \in_c \mathrm{NPMV}(2k+1)$. □

Since $\mathrm{PF}_{tt}^{\mathrm{NPMV}[2^k-1]} = \mathrm{PF}^{\mathrm{NPMV}[k]}$, we have the following theorem.

THEOREM 5.11. $\mathrm{PF}^{\mathrm{NPMV}[k]} \subseteq_c \mathrm{NPMV}(2^{k+1} - 1)$.

By Theorem 5.2, the levels of the difference hierarchy of partial functions are distinct if and only if the same levels of the Boolean hierarchy are distinct. Yet, whereas the Boolean hierarchy resides entirely within P^{NP}, by Theorem 5.9, this is unlikely to be true of the difference hierarchy of partial functions.

6. Reduction classes to NPSV. In this section, we set down some results about reduction classes to NPSV. With two notable exceptions, all of our results are corollaries of theorems that we already proved, and our interest is primarily in Corollaries 6.4 and 6.5 below, which demonstrate that bounded query hierarchies with oracles in NPSV do not collapse unless the Boolean hierarchy collapses.

The following proposition is easy to prove.

PROPOSITION 6.1.

1. $\mathrm{PF}^{\mathrm{NP}} = \mathrm{PF}^{\mathrm{NPSV}}$ (*by Theorem* 2.4).
2. $\mathrm{PF}^{\mathrm{NP}[k]} \subseteq \mathrm{PF}^{\mathrm{NPSV}[k]} \subseteq \mathrm{PF}^{\mathrm{NPMV}[k]}$ *and* $\mathrm{PF}^{\mathrm{NP}[\log]} \subseteq \mathrm{PF}^{\mathrm{NPSV}[\log]} \subseteq \mathrm{PF}^{\mathrm{NPMV}[\log]}$.
3. $\mathrm{PF}_{tt}^{\mathrm{NP}[k]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NPSV}[k]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NPMV}[k]}$.
4. $\mathrm{P}_{tt}^{\mathrm{NP}} = \mathrm{P}_{tt}^{\mathrm{NPSV}}$ (*by Theorem* 3.14).
5. $\mathrm{P}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{NPSV}}$ (*by Theorem* 2.4).
6. $\mathrm{P}^{\mathrm{NP}[k]} \subseteq \mathrm{P}^{\mathrm{NPSV}[k]} \subseteq \mathrm{P}^{\mathrm{NPMV}[k]}$ *and* $\mathrm{P}^{\mathrm{NP}[\log]} \subseteq \mathrm{P}^{\mathrm{NPSV}[\log]} \subseteq \mathrm{P}^{\mathrm{NPMV}[\log]}$.
7. $\mathrm{P}_{tt}^{\mathrm{NP}[k]} \subseteq \mathrm{P}_{tt}^{\mathrm{NPSV}[k]} \subseteq \mathrm{P}_{tt}^{\mathrm{NPMV}[k]}$.

COROLLARY 6.2. *For every* $k \geq 1$, $\mathrm{P}^{\mathrm{NPSV}[k]} = \mathrm{P}^{\mathrm{NP}[k]}$.

Proof. By Proposition 6.1(6), $\mathrm{P}^{\mathrm{NP}[k]} \subseteq \mathrm{P}^{\mathrm{NPSV}[k]} \subseteq \mathrm{P}^{\mathrm{NPMV}[k]}$. By Theorem 3.7, $\mathrm{P}^{\mathrm{NPMV}[k]} = \mathrm{P}^{\mathrm{NP}[k]}$. So $\mathrm{P}^{\mathrm{NPSV}[k]} = \mathrm{P}^{\mathrm{NP}[k]}$. □

COROLLARY 6.3. *For every* $k \geq 1$, $\mathrm{P}_{tt}^{\mathrm{NPSV}[k]} = \mathrm{P}_{tt}^{\mathrm{NP}[k]}$.

COROLLARY 6.4. *If* $\mathrm{PF}_{tt}^{\mathrm{NPSV}[k+1]} = \mathrm{PF}_{tt}^{\mathrm{NPSV}[k]}$ *for some* $k \geq 1$, *then* BH *collapses to its* $(k+1)$*st level.*

Proof. Suppose $\mathrm{PF}_{tt}^{\mathrm{NPSV}[k+1]} = \mathrm{PF}_{tt}^{\mathrm{NPSV}[k]}$. Then we have $\mathrm{P}_{tt}^{\mathrm{NPSV}[k+1]} = \mathrm{P}_{tt}^{\mathrm{NPSV}[k]}$. Therefore, by Corollary 6.3, we have $\mathrm{P}_{tt}^{\mathrm{NP}[m]} = \mathrm{P}_{tt}^{\mathrm{NP}[k]}$ for every m. Since $\mathrm{P}^{\mathrm{NP}[k]} \subseteq \mathrm{NP}(k+1)$, BH collapses to its $(k+1)$st level. □

COROLLARY 6.5. *If* $\mathrm{PF}^{\mathrm{NPSV}[k+1]} = \mathrm{PF}^{\mathrm{NPSV}[k]}$ *for some* $k \geq 1$, *then* BH *collapses to its* 2^k*th level.*

Proof. Suppose $\mathrm{PF}^{\mathrm{NPSV}[k+1]} = \mathrm{PF}^{\mathrm{NPSV}[k]}$. Then we have $\mathrm{P}^{\mathrm{NPSV}[k+1]} = \mathrm{P}^{\mathrm{NPSV}[k]}$. Therefore, by Corollary 6.2, we have $\mathrm{P}^{\mathrm{NP}[m]} = \mathrm{P}^{\mathrm{NP}[k]}$ for every m. Since $\mathrm{P}^{\mathrm{NP}[k]} \subseteq \mathrm{NP}(2^k)$, BH collapses to its 2^kth level. □

The following proposition, which is the NPSV version of Theorem 3.5, requires a somewhat different proof from the NPMV case. It is interesting that our techniques seem insufficient to prove the reverse inclusion.

PROPOSITION 6.6. *For every* $k \geq 1$, $\mathrm{PF}^{\mathrm{NPSV}[k]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NPSV}[2^k-1]}$.

Proof. Let $f \in \mathrm{PF}^{\mathrm{NPSV}[k]}$ via an oracle transducer M making k many queries to an oracle partial function $g \in \mathrm{NPSV}$ computed by an NPSV machine N. We may assume that f is single-valued. Let r be the NPMV function defined by the machine U in the proof of Proposition 3.1, where $t(x) = k$. Since g is single-valued, it follows that r is also single-valued, and hence $r \in \mathrm{NPSV}$. Moreover, we have that $\lambda x.[r(x, 0^k)]$ is total and polynomial-time computable, and if b_x is the lexicographically greatest string $b \in \Sigma^k$ on which $r(x, b)$ is defined, then

$$r(x, b_x) = \begin{cases} 0 & \text{if } x \notin \mathit{dom}(f), \\ 1y & \text{if } f(x) = y. \end{cases}$$

We compute $f(x)$ by querying r in parallel on the $2^k - 1$ values $\{(x, a) \mid a \neq 0^k\}$, recovering $f(x)$ from $r(x, a)$ as above, where a is lexicographically largest such that $r(x, a)$ returns a value. □

The equality in the following theorem depends heavily on the fact that NPSV functions are single-valued, and we do not believe it holds for oracles in NPMV. The first inclusion was found independently by E. Hemaspaandra [16].

THEOREM 6.7. $\mathrm{PF}^{\mathrm{NPSV}[\log]} \subseteq \mathrm{PF}_{tt}^{\mathrm{NPSV}} = \mathrm{PF}_{tt}^{\mathrm{NP}}$.

Proof. The first inclusion arises immediately from adapting the proof of Proposition 6.6. We now show that $\mathrm{PF}_{tt}^{\mathrm{NPSV}} \subseteq \mathrm{PF}_{tt}^{\mathrm{NP}}$. The reverse inclusion is obvious.

If h is any single-valued partial function, define $\mathit{code}(h)$ to be the set of all (x, i, b) such that the ith bit (left to right) of $h(x)$ exists and is b. Suppose $f \in \mathrm{PF}_{tt}^{\mathrm{NPSV}}$ is

computed by a deterministic oracle transducer M running in time $p(n)$ and making parallel queries to an oracle $g \in \mathrm{NPSV}$, which itself is computed by a machine N running in time $q(n)$ (p and q are polynomials). We may assume f is single-valued. Let g' be the function that maps x to $1y$ if $g(x)$ maps to y and is undefined otherwise. Clearly, $code(g') \in \mathrm{NP}$, and since g' is single-valued, at most one of the tuples $(x, i, 0)$ and $(x, i, 1)$ is in $code(g')$, for all x and i.

Given an input x, we compute $f(x)$ by making parallel queries to $code(g')$ as follows: let $q_1, \ldots, q_s$ be the oracle queries made by $M(x)$. We query $code(g')$ on all the tuples (q_i, j, b) for $1 \le i \le s$, $0 \le j \le q(p(|x|))$, and $b \in \{0, 1\}$. Since $dom(g') = dom(g)$ and $g'(q)$ never maps to the empty string, $q_i \in dom(g)$ if and only if one of $(q_i, 0, 0)$ and $(q_i, 0, 1)$ is in $code(g')$. For each $q_i \in dom(g)$, we recover $g(q_i)$ in the usual way by reading from $code(g')$ all but the 0th bit of $g'(q_i)$. Query answers in hand, we continue simulating M to obtain $f(x)$. □

The next corollary was proved independently from scratch by L. Hemaspaandra [10].

COROLLARY 6.8. $\mathrm{P}_{tt}^{\mathrm{NPSV}} = \mathrm{P}_{tt}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{NP[log]}} = \mathrm{P}^{\mathrm{NPSV[log]}}$.

Proof. The first equation follows from Theorem 6.7 by considering only characteristic functions. Also by Theorem 6.7, we have $\mathrm{P}^{\mathrm{NPSV[log]}} \subseteq \mathrm{P}_{tt}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{NP[log]}} \subseteq \mathrm{P}^{\mathrm{NPSV[log]}}$, so the last equation holds. □

Recall from the introduction that it is not known whether *sat* belongs to $\mathrm{PF}^{\mathrm{NPSV}[k]}$ for any k. We know that *maxsat* is complete for $\mathrm{PF}^{\mathrm{NPMV}}$ [13]. Thus by Corollary 6.5, if for any $k \ge 1$, $maxsat \in \mathrm{PF}^{\mathrm{NPSV}[k]}$, then the Boolean and polynomial hierarchies collapse.

Although $\mathrm{PF}^{\mathrm{NPMV[log]}} = \mathrm{PF}_{tt}^{\mathrm{NPMV}}$ (Theorem 3.13), we do not know whether $\mathrm{PF}^{\mathrm{NPSV[log]}}$ and $\mathrm{PF}_{tt}^{\mathrm{NPSV}}$ are equal. In particular, whereas, $\mathrm{PF}_{tt}^{\mathrm{NPSV}} = \mathrm{PF}_{tt}^{\mathrm{NP}}$ (Theorem 6.7) is easy to prove, apparently $\mathrm{PF}^{\mathrm{NPSV[log]}}$ and $\mathrm{PF}^{\mathrm{NP[log]}}$ are not equal, for $\mathrm{NPSV} \subseteq \mathrm{PF}^{\mathrm{NPSV}[1]} \subseteq \mathrm{PF}^{\mathrm{NPSV[log]}}$, while $\mathrm{NPSV} \subseteq \mathrm{PF}^{\mathrm{NP[log]}}$ implies P = UP [15]. Thus $\mathrm{PF}^{\mathrm{NPSV[log]}} \subseteq \mathrm{PF}^{\mathrm{NP[log]}}$ implies P = UP. Similarly, $\mathrm{PF}^{\mathrm{NPMV}[1]} \subseteq \mathrm{PF}^{\mathrm{NP[log]}}$ implies P = NP.

REFERENCES

[1] R. BEIGEL, *NP-Hard Sets Are P-Superterse unless R = NP*, Technical Report 88-04, Department of Computer Science, Johns Hopkins University, Baltimore, MD, 1988.

[2] R. BEIGEL, *Bounded queries to SAT and the Boolean hierarchy*, Theoret. Comput. Sci., 84 (1991), pp. 199–223.

[3] H. BUHRMAN, private communication, 1992.

[4] S. BUSS AND L. HAY, *On truth table reducibility to SAT*, Inform. and Comput., 91 (1991), pp. 86–102.

[5] R. BOOK, T. LONG, AND A. SELMAN, *Quantitative relativizations of complexity classes*, SIAM J. Comput., 13 (1984), pp. 461–487.

[6] J. CAI, T. GUNDERMANN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The boolean hierarchy* I: *Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[7] J. CAI, T. GUNDERMANN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The boolean hierarchy* II: *Applications*, SIAM J. Comput., 18 (1989), pp. 95–111.

[8] J. CAI AND L. HEMACHANDRA, *The Boolean hierarchy: Hardware over NP*, in Structure in Complexity Theory, Lecture Notes in Comput. Sci. 223, Springer-Verlag, Berlin, 1986, pp. 105–124.

[9] Z. CHEN AND S. TODA, *On the complexity of computing optimal solutions*, Internat. J. Found. Comput. Sci., 2 (1991), pp. 207–220.

[10] L. HEMASPAANDRA, private communication, 1993.

[11] L. Hemaspaandra, A. Naik, M. Ogihara, and A. Selman, *Computing solutions uniquely collapses the polynomial hierarchy*, SIAM J. Comput., 25 (1996), pp. 697–708.
[12] J. Kadin, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263–1282.
[13] M. Krentel, *The complexity of optimization problems*, J. Comput. Systems Sci., 36 (1988), pp. 490–509.
[14] J. Köbler, U. Schöning, and K. Wagner, *The difference and truth-table hierarchies for NP*, RAIRO Inform. Théor. Appl., 21 (1987), pp. 419–435.
[15] A. Selman, *A taxonomy of complexity classes of functions*, J. Comput. Systems Sci., 48 (1994), pp. 357–381.
[16] E. Hemaspaandra, private communication, 1993.
[17] H. Wareham, *On the Comptutational Complexity of Inferring Evolutionary Trees*, Master's thesis, Department of Computer Science, Memorial University of Newfoundland, St. John's, NF, Canada, 1992.
[18] O. Watanabe and S. Toda, *Structural analysis of the complexity of inverse functions*, Math. Systems Theory, 26 (1993), pp. 203–214.
[19] G. Wechsung and K. Wagner, *On the boolean closure of NP*, in Proc. International Conference on Fundamentals of Computation Theory, Lecture Notes in Comput. Sci. 199, Springer-Verlag, Berlin, 1985, pp. 485–493.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1066–1099, August 1997

008

FAST DISCRETE POLYNOMIAL TRANSFORMS WITH APPLICATIONS TO DATA ANALYSIS FOR DISTANCE TRANSITIVE GRAPHS*

J. R. DRISCOLL†, D. M. HEALY, JR.‡, AND D. N. ROCKMORE§

Abstract. Let $\mathcal{P} = \{P_0, \ldots, P_{n-1}\}$ denote a set of polynomials with complex coefficients. Let $\mathcal{Z} = \{z_0, \ldots, z_{n-1}\} \subset \mathbb{C}$ denote any set of *sample points*. For any $f = (f_0, \ldots, f_{n-1}) \in \mathbb{C}^n$, the *discrete polynomial transform* of f (with respect to $\mathcal{P}$ and $\mathcal{Z}$) is defined as the collection of sums, $\{\widehat{f}(P_0), \ldots, \widehat{f}(P_{n-1})\}$, where $\widehat{f}(P_j) = \langle f, P_j \rangle = \sum_{i=0}^{n-1} f_i P_j(z_i) w(i)$ for some associated weight function w. These sorts of transforms find important applications in areas such as medical imaging and signal processing.

In this paper, we present fast algorithms for computing discrete orthogonal polynomial transforms. For a system of N orthogonal polynomials of degree at most $N-1$, we give an $O(N \log^2 N)$ algorithm for computing a discrete polynomial transform at an arbitrary set of points instead of the N^2 operations required by direct evaluation. Our algorithm depends only on the fact that orthogonal polynomial sets satisfy a three-term recurrence and thus it may be applied to any such set of discretely sampled functions.

In particular, sampled orthogonal polynomials generate the vector space of functions on a distance transitive graph. As a direct application of our work, we are able to give a fast algorithm for computing subspace decompositions of this vector space which respect the action of the symmetry group of such a graph. This has direct applications to treating computational bottlenecks in the spectral analysis of data on distance transitive graphs, and we discuss this in some detail.

Key words. fast Fourier transform, FFT, discrete polynomial transform, orthogonal polynomials, three-term recurrence, distance transitive graph

AMS subject classifications. Primary, 42C05, 42C10, 42-04, 33C90; Secondary, 65T20, 62-04, 62-07, 05C99

PII. S0097539792240121

1. Introduction.

The efficient decomposition of a function into a linear combination of orthogonal polynomials is a fundamental tool which plays an important role in a wide variety of computational problems. Applied science abounds with computations using such decompositions, along with the related computational techniques for calculation of correlation or projection of data onto a family of polynomials. To cite just a few examples, this sort of approach is used in spectral methods for solving differential equations [Bo, Te], data analysis [D], signal and image processing [OS], and the construction of Gauss quadrature schemes [Ga1]. In most cases, the choice of a particular family of polynomials is determined by some special property or underlying symmetry of the problem under investigation.

*Received by the editors October 16, 1992; accepted for publication (in revised form) August 22, 1995.

http://www.siam.org/journals/sicomp/26-4/24012.html

†Driscoll Brewing, 81 Oakwood Drive, Murray Hill, NJ 07974 (driscoll@cs.dartmouth.edu). The research of this author was partially supported by DARPA as administered by the AFOSR under contract AFOSR-90-0292.

‡Departments of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755 (healy@cs.dartmouth.edu). The research of this author was partially supported by DARPA as administered by the AFOSR under contract AFOSR-90-0292 and by ARPA as administered by the AFOSR under contract DOD-F4960-93-056.

§Departments of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755 (rockmore@cs.dartmouth.edu). The research of this author was partially supported by an NSF Mathematical Sciences Postdoctoral Fellowship and by ARPA as administered by the AFOSR under contract DOD-F4960-93-056.

Perhaps the most familiar example is the representation of a discrete data sequence as a linear combination of phase polynomials. In this case, the decomposition is known as the discrete Fourier transform (DFT) and is accomplished both efficiently and reliably through the use of the well-known fast Fourier transform algorithms (FFT) (cf. [ER] and the references therein). The DFT is a particularly simple orthogonal polynomial transform which corresponds to the projection of a data sequence $f = (f_0, \ldots, f_{N-1})$ onto the family of monomials $P_l(x) = m_l(x) = x^l$ evaluated at the roots of unity $z_k = e^{2\pi ik/N}$, $k = 0, 1, \ldots, N-1$. Thus the DFT is the collection of sums

$$\widehat{f}(l) = \sum_{k=0}^{N-1} f_k P_l(z^k) = \sum_{k=0}^{N-1} f_k e^{2\pi ikl/N} \tag{1.1}$$

for the discrete frequencies $l = 0, 1, \ldots, N-1$. The monomials form an orthogonal set whose properties account for the well-documented usefulness and algorithmic efficiency of the FFT algorithms. In particular, these algorithms allow the projections in (1.1) to be computed in $O(N \log N)$ operations as opposed to the N^2 operations that a direct evaluation would require [ER]. (We assume a standard model in which a single complex multiplication and addition are defined as a single operation.)

In this paper, we are concerned with the development of efficient algorithms for computing more general discrete polynomial transforms. Specifically, let $\mathcal{P} = \{P_0, \ldots, P_{n-1}\}$ denote a set of polynomials with complex coefficients. Let $\mathcal{Z} = \{z_0, \ldots, z_{n-1}\} \subset \mathbb{C}$ denote any set of *sample points*. If $f = (f_0, \ldots, f_{n-1})$ is any data vector (often thought of as a function with known values at the sample points), then the *discrete polynomial transform* of f (with respect to $\mathcal{P}$ and $\mathcal{Z}$) is defined as the collection of sums, $\{\widehat{f}(P_0), \ldots, \widehat{f}(P_{n-1})\}$, where

$$\widehat{f}(P_j) = \langle f, P_j \rangle = \sum_{i=0}^{n-1} f_i P_j(z_i) w(i). \tag{1.2}$$

The function w is some associated weight function, often identically 1. Familiar examples of discrete polynomial transforms include the DFT (already mentioned) as well as the related discrete cosine transform (DCT). In fact, both may be obtained as particular cases of *discrete monomial transforms*—i.e., discrete polynomial transforms in which $P_j = m_j$ is the monomial of degree j. Beyond such special cases, we know of no prior general algorithm for computing discrete polynomial transforms which has complexity less than $O(n^2)$.

Inspection of equation (1.2) shows that direct computation of the discrete polynomial transform requires n^2 operations. For large n, this cost quickly becomes prohibitive. The main result of this paper is an algorithm which computes general discrete orthogonal polynomial transforms in $O(n \log^2 n)$ operations. This relies primarily on the three-term recurrences satisfied by any orthogonal polynomial system and as such our algorithms also obtain for computing transforms over any set of spanning functions which satisfy such a recurrence. Related techniques have already found a number of applications attacking computational bottlenecks in problems in areas such as medical imaging, geophysics, and matched filter design [DrH, MHR, HMR, HMMRT].

Our original motivation for studying these sorts of computations comes from problems which arise in performing spectral analysis of data on distance transitive graphs. This analysis is effectively the combinatorial analogue of the more familiar case of spectral analysis on continuous spaces like the circle or the 2-sphere. For instance,

functions defined on distance transitive graphs admit a spectral decomposition which mirrors that of integrable functions on the 2-sphere. In particular, recall that the algebra of functions on the 2-sphere is generated by functions constant on circles of fixed distance from the north pole (circles of latitude), the so-called "zonal spherical functions" for the 2-sphere [He]. For each nonnegative integer m, there is a uniquely defined (up to a constant) spherical function of degree m and the translates of this function under the action of $SO(3)$ (the symmetry group of the 2-sphere given by the group of rotations in 3-space) span a subspace of the vector space of functions on the 2-sphere which is invariant under the action of $SO(3)$. Similarly, distance transitive graphs have an associated symmetry group. After the choice of a distinguished vertex, analogous to the choice of a "north pole" on the 2-sphere, the algebra of functions on a distance transitive graph is generated by analogously defined spherical functions. (Here distance on the graph is the usual shortest path distance between vertices.) It turns out that these discrete functions are sampled orthogonal polynomials. Spectral analysis of data on a distance transitive graph, naturally viewed as a function on the graph, requires the expansion of the function in terms of a basis generated by the discrete spherical functions. The expansion may be reduced to the computation of discrete spherical transforms which are, in fact, discrete orthogonal polynomial transforms.

The spectral approach to data analysis, as described by Diaconis [D], is motivated by the observation that it is often appropriate and useful to view data as a function defined on an suitably chosen group or, more generally, some homogeneous space of a group. The choice of a "natural" group in any given situation depends on various symmetries of the problem. The group-theoretic setting of spectral analysis allows for the techniques of Fourier analysis to be applied. In particular, a data vector will have a natural decomposition into symmetry-invariant components which are calculated by computing the projections of the data vector into the various symmetry-invariant subspaces.

A familiar illustration of this approach comes from digital signal processing. Here the standard analysis of stationary signals proceeds by decomposing the signal as a sum of sines and cosines with coefficients determined by usual abelian FFT. The sines and cosines of a given frequency determine subspaces of functions which are invariant under translation of the origin.

For a possibly less familiar example (due to Diaconis [D]), consider the California Lottery game. Each player chooses a six-element subset of $\{1, \ldots, 49\}$. Every such subset corresponds to a coset of $S_{49}/(S_6 \times S_{43})$. (Here $S_6 \times S_{43}$ is identified with the subgroup of S_{49} that independently permutes the subsets $\{1, \ldots, 6\}$ and $\{7, \ldots, 49\}$.) The vector space of functions defined on the cosets $S_{49}/(S_6 \times S_{43})$ is denoted as $M^{(43,6)}$. Each "run" of the game gives rise to a function $f \in M^{(43,6)}$ such that $f(x)$ is the number of people picking 6-set x.

A spectral analysis approach to analyzing such a data vector is to decompose the vector into symmetry-invariant components, where here a natural choice of symmetry group is the symmetric group S_{49}. Standard analysis from the representation theory of the symmetric group shows that $M^{(43,6)}$ has a unique finest decomposition into seven S_{49}-invariant components, $M^{(43,6)} = S^{(49)} \oplus S^{(48,1)} \oplus \cdots \oplus S^{(43,6)}$. This decomposition has a natural data-analytic interpretation. The invariant subspace $S^{(49)}$ measures the constant contribution. The other invariant subspaces $S^{(49-j,j)}$ naturally measure the "pure" contribution of the popularity of the various j-sets (that is, the number of people including a given j-set in their 6-set). Computation of the projections onto

these subspaces can be reduced to computing the relevant spherical transforms, which in this case turn out to be certain discrete Hahn polynomial transforms. The methods of this paper allow these transforms to be computed efficiently.

The California Lottery example is, in fact, an example of data on a distance transitive graph. More generally, the k-sets of an n-set comprise a distance transitive graph by joining any two k-sets which differ by only a single element. This graph possesses certain Hahn polynomials as its spherical functions (cf. [St3, section II.3]). Other examples include the n-gon graph with dihedral group symmetry as well as the n-dimensional hypercube with hyperoctahedral group of symmetries. In the former case, the spherical functions are obtained from the Chebyshev polynomials, $T_n(x) = \cos(n \arccos(x))$ [Bi], and in the latter case, the Krawtchouk polynomials give the spherical functions (cf. [St3, section II.2]).

As we shall see in section 3, in general, the problem of finding an FFT for distance transitive graphs may be reduced to that of the efficient computation of the projection onto the spherical functions for the graph, which are an orthogonal family of special functions on the graph. In many important examples (cf. [St1, St3] and the many references therein), these spherical functions are actually sampled orthogonal polynomials, and the spherical transform amounts to projection onto these polynomials in a weighted ℓ^2 space.

The organization of the paper is as follows. Section 2 discusses fast orthogonal polynomial transforms, beginning with previously known results for monomial transforms and concluding with our main computational result, which is an efficient discrete orthogonal polynomial transform. This material is elementary and relies on nothing more than the recurrence relations satisfied by the polynomials in question. Section 3 treats our main application of interest: fast algorithms for projection onto spherical functions on distance transitive graphs. We include here the necessary group-theoretic background and notation and give explicit examples of the algorithm for spherical functions on several graphs of interest. The fast spherical transform algorithm may be modified in order to provide a fast inverse transform, and from this we also obtain a fast convolution algorithm for functions on distance transitive graphs. Section 4 discusses the connection of these results to the computation of isotypic projections required for spectral analysis. We close in section 5 with some final remarks.

2. Fast polynomial transforms. The goal of this section is to produce algorithms for fast evaluation of polynomial transforms with an eye to their eventual application to the efficient computation of spherical transforms. The general algorithm proceeds in two steps. The initial phase is an efficient projection onto the monomials. From here we are able to use the three-term recurrence to obtain a divide-and-conquer approach for relevant fast polynomial transforms. In general, our approach is to formulate the initial problem as a particular matrix–vector multiplication and then present the fast algorithm as a particular matrix factorization.

We proceed by first recalling the fast monomial transform. This is obtained by writing it as the transpose of multiple-point polynomial evaluation and then formulating a well-known efficient algorithm for the latter process (cf. [BM, Chapter 4]) as a structured matrix factorization. We explain the full algorithm next and then close this section with an example.

2.1. Fast monomial transforms. The simplest polynomial transform problem that we could consider is the projection of a vector $f = (f_0, \ldots, f_{n-1})$ onto the family

of monomials evaluated at the finite point set $\{z_0, \ldots, z_{n-1}\}$,

$$\widehat{f}(k) = \langle f, z^k \rangle = \sum_{\ell=0}^{n-1} f_\ell (z_\ell)^k \quad (k = 0, \ldots, n-1). \tag{2.1}$$

Note that viewed as a matrix–vector multiplication, this is the evaluation of multiplication of the suitably defined Vandermonde matrix times the vector of samples. That is,

$$\langle f, z^k \rangle = (V \cdot f)_k,$$

where

$$V = V(z_0, \ldots, z_{n-1}) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ z_0 & z_1 & \cdots & z_{n-1} \\ \vdots & & & \vdots \\ {z_0}^{n-1} & {z_1}^{n-1} & \cdots & {z_{n-1}}^{n-1} \end{pmatrix}. \tag{2.2}$$

The familiar example of the abelian DFT is obtained by taking the evaluation points to be the nth roots of unity in $\mathbb{C}$, $z_\ell = \exp(2\pi i \ell/n)$. This projection may be obtained by the familiar FFT divide and conquer strategy in $O(n \log n)$ operations as opposed to the obvious $O(n^2)$. General references include [BM, ER, N, TAL].

Notice that the abelian FFT gives rise to our first efficient spherical transform, corresponding to the n-gon graph. This is a fast discrete Chebyshev transform with samples at the Chebyshev points, $\cos 2\pi\ell/n$. This can be obtained by applying the usual FFT to a real-valued data sequence.

Our fast monomial transform is based on the formulation of the FFT which considers the *transpose* of projection and develops the algorithm as polynomial evaluation at the roots of unity,

$$\widehat{f}(k) = (V^t \cdot f)_k = \sum_{\ell=0}^{n-1} f_\ell (z_k)^\ell$$

for $z_k = \exp(2\pi i k/n)$. This version of the FFT is achieved by efficient recursive application of the division algorithm (cf. [BM]).

An advantage of this perspective is that it allows an easy generalization to the direct evaluation of polynomials at n real points. We now review a well-known $O(n \log^2 n)$ algorithm for polynomial evaluation which we may formulate as a factorization of the matrix V^t into block-diagonal matrices with Toeplitz blocks of geometrically decreasing size. It is this structure which permits the fast computation of the matrix–vector product. Consequently, we obtain a corresponding factorization of the transpose, V and hence, an algorithm for projection which also requires $O(n \log^2 n)$ operations. For ease of exposition, we assume n is a power of 2.

LEMMA 2.1. *Let $n = 2^k$ and let V be the Vandermonde matrix for the set of complex points $z_0, z_1, \ldots, z_{n-1}$, as defined in* (2.2). *The matrix–vector product $V^t \cdot f$, for $f \in \mathbb{C}^n$ (corresponding to the evaluation of the polynomial $f_0 + f_1 z + \cdots + f_{n-1} z^{n-1}$ at the points $z_0, z_1, \ldots, z_{n-1}$) may be accomplished in $O(n \log^2 n)$ operations. Likewise, the product $V \cdot f$ for $f \in \mathbb{C}^n$ (corresponding to projection of a sampled function f onto the monomials sampled at $z_0, z_1, \ldots, z_{n-1}$) may be accomplished in $O(n \log^2 n)$ operations.*

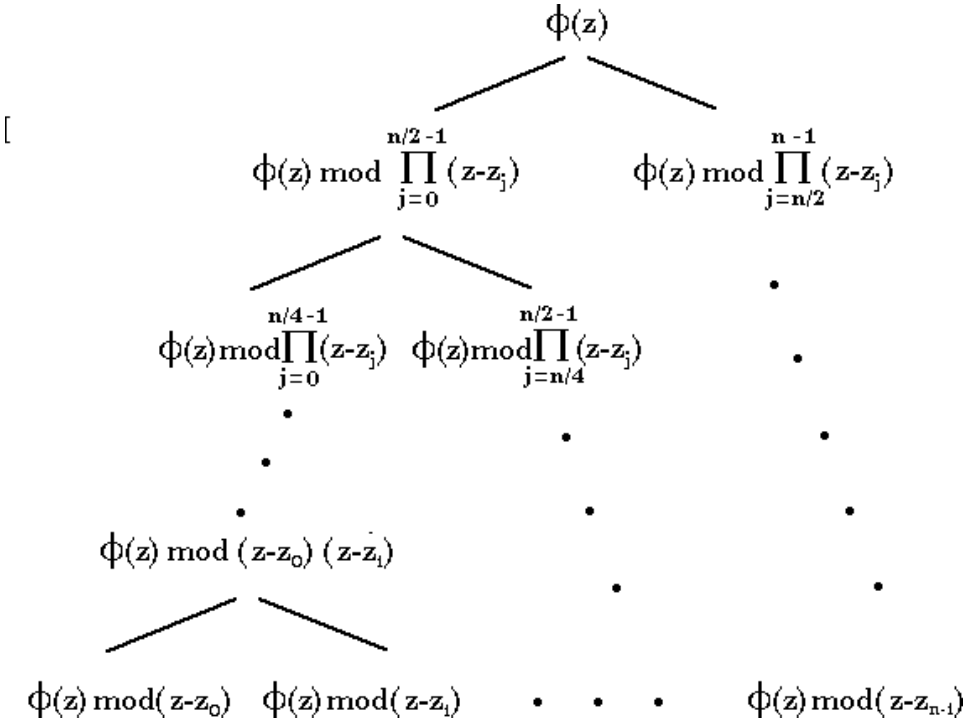


FIG. 2.1. *Tree for evaluating a polynomial $\phi(z)$ at $n = 2^k$ points. Note that it has $k = \log n$ levels.*

Proof. Let $\phi(z) = \sum_{i=0}^{n-1} f_i z^i$, $n = 2^k$, with $k \geq 0$. We may evaluate ϕ at any of the z_j, $j = 0, 1, \ldots, n-1$ by the division algorithm because $\phi(z_j) = \phi(z) \mod (z - z_j)$. The division may be done in $O(n \log n)$ for a given z_j, but to proceed this way for each of the z_j separately is prohibitively expensive.

Instead, we use a familiar divide-and-conquer strategy, simultaneously reducing the original polynomial modulo each linear factor $(z - z_j)$ in $k = \log n$ stages, as shown in Figure 2.1. Notice that fast polynomial arithmetic algorithms allow for the various moduli to be precomputed in $O(n \log^2 n)$ operations (cf. [BM, section 4.3]).

Each downward edge in the tree in Figure 2.1 represents the reduction of a polynomial $p(x)$ modulo another polynomial of form $m_S(z) = \Pi_{z_j \in S}(z - z_j)$, corresponding to a certain subset S of the evaluation points $z_0, \ldots, z_{n-1}$. To move down this edge of the tree, we need an algorithm to efficiently compute the remainder $r_S(x)$, and incidentally $q(x)$, in the division algorithm representation

$$p(x) = q(x) m_S(x) + r_S(x).$$

The input p is a remainder from a previous stage and has degree $d - 1$, where d is a power of 2. The precomputed modulus m_S has degree $d/2$. Therefore, r_S has degree $d/2 - 1$.

The key point is that in this tree, r_S is equivalent mod m_S not only to its immediate ancestor p but also to every ancestor of p, all the way back to the original polynomial ϕ. Indeed, p was itself obtained as a remainder modulo $m_{\tilde{S}}$ from *its* ances-

tor P, and Figure 2.1 shows that always $S \subset \tilde{S}$, so $m_S \,|\, m_{\tilde{S}}$. Therefore, $m_S|(r_S - P)$. So upon reaching the leaves of the tree, we have actually computed $\phi(z)$ modulo the linear factors $(z - z_j)$ as desired.

To see how to compute the basic reduction steps efficiently, we write the division algorithm representation $r = p - qm$ in matrix form. It is natural to split this equation into a high-order and a low-order part, due to the vanishing of the higher-order coefficients of r, corresponding to powers $d/2, \ldots, d-1$. The low-order equation involving the nonzero coefficients of r looks like

$$\begin{pmatrix} r_{\frac{d}{2}-1} \\ \vdots \\ \vdots \\ \vdots \\ r_0 \end{pmatrix} = \begin{pmatrix} p_{\frac{d}{2}-1} \\ \vdots \\ \vdots \\ \vdots \\ p_0 \end{pmatrix} - \begin{pmatrix} m_0 & m_1 & \cdots & \cdots & m_{\frac{d}{2}-1} \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & m_1 \\ 0 & \cdots & \cdots & 0 & m_0 \end{pmatrix} \begin{pmatrix} q_{\frac{d}{2}-1} \\ \vdots \\ \vdots \\ \vdots \\ q_0 \end{pmatrix}. \tag{2.3}$$

The upper triangular Toeplitz matrix in (2.3) is comprised of the lower-order coefficients of the polynomial m; for future reference, we call this matrix M.

Now the higher-order terms of r are zero, so the high-order equation reduces to

$$\begin{pmatrix} p_{d-1} \\ \vdots \\ \vdots \\ \vdots \\ p_{\frac{d}{2}} \end{pmatrix} = \begin{pmatrix} m_{\frac{d}{2}} & 0 & \cdots & \cdots & 0 \\ m_{\frac{d}{2}-1} & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ m_1 & \cdots & \cdots & m_{\frac{d}{2}-1} & m_{\frac{d}{2}} \end{pmatrix} \begin{pmatrix} q_{\frac{d}{2}-1} \\ \vdots \\ \vdots \\ \vdots \\ q_0 \end{pmatrix}. \tag{2.4}$$

Since $m_{d/2} = 1$, the lower triangular Toeplitz matrix in (2.4) is invertible. Its inverse, G, is also lower triangular and Toeplitz and may be computed in $O(d \log d)$ operations by a Newton iteration and then prestored (cf. [BM, Chapter 4] and Remark 3 following this proof). Insert the result into equation (2.3). This gives

$$\begin{pmatrix} r_{\frac{d}{2}-1} \\ \vdots \\ r_0 \end{pmatrix} = \begin{pmatrix} p_{\frac{d}{2}-1} \\ \vdots \\ p_0 \end{pmatrix} - MG \begin{pmatrix} p_{d-1} \\ \vdots \\ p_{\frac{d}{2}} \end{pmatrix}$$

$$= \left(\begin{array}{c|c} & \\ -MG & I_{d/2} \\ & \end{array}\right) \begin{pmatrix} p_{d-1} \\ \vdots \\ p_{\frac{d}{2}} \\ \hline p_{\frac{d}{2}-1} \\ \vdots \\ p_0 \end{pmatrix}. \tag{2.5}$$

Here $I_{d/2}$ is the $d/2 \times d/2$ identity matrix.

Here we must briefly recall that standard techniques using the FFT allow Toeplitz matrices of dimension b to be multiplied by an arbitrary vector of length b in at most

$O(b \log b)$ operations. This is done by framing the computation as the convolution of two sequences. More specifically, the Toeplitz matrix of order b is extended to a circulant matrix of order $2b$. A zero-padded version of the original data vector is then multiplied by this matrix in order to obtain the appropriate product. The new matrix–vector multiplication is precisely the circular convolution of two sequences of length $2b$ and as such is performed efficiently by computing the FFTs of each of the sequences, performing the pointwise multiplications of the resulting sequences and finally computing an inverse Fourier transform (requiring one more FFT) of this sequence. Thus a total of three FFTs are required as well as one pointwise multiplication of a sequence of length $2b$. If $2b = 2^r$, then an FFT of length $2b$ requires at most $3/2 \cdot 2b \cdot r = 3br$ operations (cf. [BM, p. 84]). Consequently, the multiplication of a Toeplitz matrix of order $b = 2^{r-1}$ by an arbitrary vector requires at most $3 \cdot 3br + 2b$ operations or $O(b \log b)$ operations.

Since M and G are both Toeplitz, the above discussion shows that the product

$$M \cdot G \cdot \begin{pmatrix} p_{d-1} \\ \vdots \\ p_{\frac{d}{2}} \end{pmatrix}$$

can be computed in at most $2(9 \cdot (d/2)r + 2 \cdot d/2) = 9dr + d$ operations, where $d = 2^r$. This is effected by first performing the multiplication

$$G \cdot \begin{pmatrix} p_{d-1} \\ \vdots \\ p_{\frac{d}{2}} \end{pmatrix}$$

and then multiplying this result by M. This means then that the multiplication in (2.5) may be accomplished in $9dr + d + d/2 = 9dr + 3d/2$ operations. The additional term of $d/2$ comes from the multiplication of the identity subblock against the low-order coefficients. Note that this is the cost of a single reduction in a single stage of the algorithm.

Looking back at the tree (see Figure 2.1), we see that the first stage of the algorithm consists of two reductions from order $n = 2^k$ to order $n/2$ by two polynomial divisions. Consequently, if we let $T(n)$ denote the number of operations required to compute the order n problem, then we obtain the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + 2 \cdot \left(9nk + \frac{3}{2}n\right) = 18n \log n + 3n. \tag{2.6}$$

Iteration of the recurrence (2.6) yields

$$T(n) \leq 18n \log^2 n + 3n \log n, \tag{2.7}$$

which shows that the entire computation can be performed in $O(n \log^2 n)$ operations.

This sequence of reductions can be encoded as a structured matrix factorization of V^t. Let $M_{l,i}$ denote the upper triangular Toeplitz matrix associated (in the sense of (2.3)) with the polynomial which gives the ith modulus at level l of the tree in Figure 2.1. Thus $M_{1,0}$ is associated with $(z - z_0) \cdots (z - z_{n/2-1})$, $M_{1,1}$ is associated with $(z - z_{n/2}) \cdots (z - z_{n-1})$, $M_{2,0}$ is associated with $(z - z_0) \cdots (z - z_{n/4-1})$, and, in general, $M_{l,i}$ is associated with the product $(z - z_{i(n/2^l)}) \cdots (z - z_{(i+1)(n/2^l)-1})$.

Similarly, let $G_{l,i}$ denote *inverse* of the lower triangular Toeplitz matrix associated (in the sense of (2.4)) with the polynomial which gives the ith modulus at level l of the tree in Figure 2.1. As discussed above, $G_{l,i}$ is itself lower triangular and Toeplitz.

Define the matrices $R_{l,i}$ as in (2.5) by

$$R_{l,i} = \begin{pmatrix} & | & \\ -M_{l,i}G_{l,i} & | & I_{2^{k-l}} \\ & | & \end{pmatrix} \begin{matrix} \uparrow \\ 2^{k-l} \\ \downarrow \end{matrix} \tag{2.8}$$
$$\leftarrow 2^{k-l} \rightarrow \quad \leftarrow 2^{k-l} \rightarrow$$

for $1 \le l \le k = \log n$ and $0 \le i < 2^l$. Then the previous discussion shows that V^t has a factorization into $k = \log n$ factors as the matrix product

$$\begin{pmatrix} \begin{pmatrix} R_{k,0} \\ - - - \\ R_{k,1} \end{pmatrix} & & & \bigcirc \\ & \begin{pmatrix} R_{k,2} \\ - - - \\ R_{k,3} \end{pmatrix} & & \\ & & \ddots & \\ \bigcirc & & & \begin{pmatrix} R_{k,n-2} \\ - - - \\ R_{k,n-1} \end{pmatrix} \end{pmatrix}$$
$$\cdots \begin{pmatrix} \begin{pmatrix} & R_{2,0} & \\ - & - & - \\ & R_{2,1} & \end{pmatrix} & & \bigcirc \\ & \ddots & \\ \bigcirc & & \begin{pmatrix} & R_{2,2} & \\ - & - & - \\ & R_{2,3} & \end{pmatrix} \end{pmatrix} \begin{pmatrix} & R_{1,0} & \\ & & \\ - & - & - \\ & & \\ & R_{1,1} & \end{pmatrix}.$$

By transposition, a similarly structured factorization of V is then also obtained. The reversal of order obviously does not change the complexity of the sequence of multiplications; each matrix is still block diagonal, with the blocks themselves comprised of products of triangular Toeplitz subblocks as before. □

Remarks. 1. As mentioned previously, Lemma 2.1 is a restatement of what is now a classical result of the complexity for polynomial evaluation. For variations on this algorithm as well as pointers to the more recent literature, we refer the reader to the survey article of Pan [P] and the extensive bibliography contained therein.

2. Our proof treats only the case of n equal to a power of 2 but may be extended to the general case in a straightforward manner with the same asymptotic result.

3. Notice that the above algorithm requires $O(n \log n)$ storage. To see this, recall that the matrix–vector multiplications involving the matrices $M_{l,i}$ and $G_{l,i}$ are effected by extending these matrices to the appropriate circulant matrices of twice the size and then performing the subsequent matrix–vector multiplications as circular convolutions using the FFTs of the associated sequences. The matrices $M_{l,i}$ and $G_{l,i}$

are of dimension 2^{k-l} (where $n = 2^k$) and thus are extended to circulants of size 2^{k-l+1}. We need only store the DFT of a single row of this circulant, so in total we require $2n \log n$ storage to keep the necessary data from the all of the $M_{l,i}$'s and $G_{l,i}$'s.

To generate this initial data structure, we require $O(n \log^2 n)$ operations. For this, we first note that to generate the necessary DFTs from all of the $M_{l,i}$'s and $G_{l,i}$'s we require at most $O(n \log^2 n)$ operations, assuming that we have constructed the $M_{l,i}$'s and $G_{l,i}$'s. The $M_{l,i}$'s and the $G_{l,i}^{-1}$'s are obtained from the polynomial coefficients of the various supermoduli in the division tree of Figure 2.1. These may be generated recursively from the bottom of the tree up using efficient polynomial multiplication routines which require $O(m \log m)$ operations to multiply two polynomials of degree m (cf. [BM, p. 86]). Thus at most $O(n \log^2 n)$ operations are needed to generate all of the $M_{l,i}$'s and $G_{l,i}^{-1}$'s (cf. [BM, p. 100]). Finally, to invert any particular $G_{l,i}^{-1}$ in order to obtain $G_{l,i}$, an additional $O(l\, 2^l)$ is needed (cf. [BM, p. 96]) so that in total we require $O(n \log^2 n)$ operations to precompute the necessary data structure.

4. Notice that one direct result of an efficient monomial transform is that we can obtain an FFT at nonuniformly spaced frequencies. This amounts to evaluating the polynomial above at n nonuniformly spaced points on the unit circle and can be accomplished in $O(n \log^2 n)$ operations. An application of this to fast scanning for MRI is discussed in [MHR], as are issues of stability of the fast algorithm.

Nonuniform FFTs also immediately provide an $O(n \log^2 n)$ Chebyshev polynomial transform on the uniform grid $\{k/n - 1 | k = 0, \ldots, 2n-1\}$ in $[-1, 1]$ by applying the nonuniform Fourier transform to a real data sequence f at the points $\exp(i\theta_k)$ with $\cos(\theta_k) = k/n$. This turns out to be useful, and we will explore it in more detail later.

Lemma 2.1 has many applications. We record some here for later use.

COROLLARY 2.2. *Each of the following three computations can be obtained in* $O(n \log^2 n)$ *operations:*

(1) *the* ℓ^2 *projections of a discrete function onto the monomials sampled at the points* $x_0, x_1, \ldots, x_{n-1}$ *in* $\mathbb{R}$,

$$\sum_{k=0}^{n-1} f_k {x_k}^l, \quad l = 0, \ldots, n-1,$$

(2) *the* ℓ^2 *projections of a discrete function onto the Chebyshev polynomials* $T_n(x)$ *sampled uniformly at the points*

$$\left\{ u_k = 2\frac{k}{n} - 1 \middle| k = 0, \ldots, n-1 \right\} \subset [-1, 1],$$

$$\sum_{k=0}^{n-1} f_k T_l(u_k), \quad l = 0, \ldots, n-1;$$

(3) *the* ℓ^2 *projections of a discrete function onto the shifted Chebyshev polynomials* $T_n^*(x) = T_n(2x-1)$ *on the regular grid*

$$\left\{ v_k = \frac{k}{n} \middle| k = 0, \ldots, n-1 \right\} \subset [0, 1].$$

Proof. (1) This is an immediate application of the lemma.

(2) Take $\theta_k = \arccos(u_k)$ in $[0, \pi]$, $k = 0, \ldots, n-1$. Define points $z_j, j = 0, \ldots, 2n-1$ in the unit circle by

$$z_j = \begin{cases} e^{i\theta_j} & \text{if } 0 \le j < n-1, \\ e^{i(\theta_{j-n} - \pi)} & \text{if } n \le j < 2n. \end{cases}$$

Then

$$\begin{aligned} \sum_{k=0}^{n-1} f_k T_l(u_k) &= \sum_{k=0}^{n-1} f_k \cos(l\theta_k) \\ &= \sum_{k=0}^{n-1} \frac{1}{2} f_k (z_k^l + \overline{z_k^l}) \\ &= \sum_{j=0}^{2n-1} F(z_j) z_j^l, \end{aligned}$$

where $F(z_0) = f_0$; $F(z_k) = (1/2)f_k$, $k = 1, \ldots, n-1$; $F(z_n) = 0$; and $F(\overline{z}) = F(z)$. The result follows by applying Lemma 2.1 to this last expression.

Alternatively, one may apply Lemma 2.1 separately to the real and imaginary parts of f, evaluating at the points $z_j, j = 0, \ldots, n-1$, and taking the real part.

(3) This follows from (2) by a change of variables. □

For reasons of numerical stability, the projections described in the last two parts of the corollary provide a useful alternative to projection onto monomials on uniform grids. Even though the Chebyshev polynomials are not discretely orthogonal on the uniform grid, they still are much better conditioned than the monomials [Ga2, Hi]. It should also be noted that certain modifications of the resulting algorithm for projection onto the Chebyshev polynomials are required for stable computation. These modifications do not affect the efficiency of the algorithm in any appreciable way (cf. [MHR]).

2.2. Three-term recurrence relations and fast projection. We wish to extend the results of section 2.1 to obtain an algorithm for the fast projection onto functions other than the monomials or the Chebyshev polynomials. In particular, we are interested in doing this for the spherical functions for distance transitive graphs. These functions satisfy three-term recurrence relations, which permits us to make an efficient change of basis from monomials or Chebyshev polynomials. The following theorem demonstrates this in a case of interest for the current paper. It is evident that the argument can be applied in more general situations.

THEOREM 2.3. *Let $n = 2^k$ and let $\Phi_i(x)$, $i = 0, \ldots, n-1$ comprise a family of functions defined at the positive integers $x = 0, 1, \ldots, n-1$ and satisfying a three-term recurrence there:*

$$\Phi_{l+1}(x) = (a_l x + b_l)\Phi_l(x) + c_l \Phi_{l-1}(x),$$

with initial conditions $\Phi_0 = 1$, $\Phi_{-1} = 0$. Then the projections of a data vector $f = (f_0, \ldots, f_{n-1})$ defined by

$$\widehat{f}(l) = \sum_{j=0}^{n-1} f_j \Phi_l(j) w_j = \langle f, \Phi_l \rangle,$$

where w is a weight function, can be computed for all $l < n$ in $O(n \log^2 n)$ operations.

Proof. Without loss of generality, we may assume that $w_i = 1$ for each i. (In the more general case, the weights could be absorbed immediately into f.) By Lemma 2.1, we can effect the projection onto the monomials of degree less than n sampled at the points $x = 0, 1, \ldots, n-1$ in $O(n \log^2 n)$ operations. We now use the three-term recurrence to transform these into the desired projections onto the Φ_l.

Define the sequence Z_l for each $l = 0, 1, \ldots, n-1$ by

$$Z_l(k) = \langle f, x^k \Phi_l \rangle = \sum_{j=0}^{n-1} f_j j^k \Phi_l(j) \tag{2.9}$$

for $k = 0, 1, \ldots, n-1$. Our goal is to obtain the values $Z_l(0) = \langle f, x^0 \Phi_l \rangle$. However, what we may compute efficiently from the initial data are the values $Z_0(k) = \langle f, x^k \rangle$.

In terms of the Z_l, the recurrence

$$\Phi_{l+1}(x) = (a_l x + b_l)\Phi_l(x) + c_l \Phi_{l-1}(x) \tag{2.10}$$

translates into

$$\begin{aligned} Z_{l+1}(k) &= a_l \langle f, x^{k+1} \Phi_l \rangle + b_l \langle f, x^k \Phi_l \rangle + c_l \langle f, x^k \Phi_{l-1} \rangle \\ &= a_l Z_l(k+1) + b_l Z_l(k) + c_l Z_{l-1}(k). \end{aligned} \tag{2.11}$$

Observe that the weights in (2.11) do not depend on the k index. That is, the sequence Z_{l+1} is obtained by adding scalar multiples of the sequences Z_l, Z_{l-1}, and a shifted version of Z_l.

According to Lemma 2.1 and, specifically, equation (2.7), we can compute the sequence Z_0 in $18n \log^2 n + 3n \log n$ operations. Setting $Z_{-1} = 0$, the recurrence (2.11) gives Z_1 in at most $2n$ additional operations. In particular, this gives the value $Z_1(0) = \widehat{f}(1)$. Proceeding in this direct fashion, one could successively build the sequences Z_l and the obtain the values $Z_l(0)$. Of course, this yields no savings, requiring n operations of length $2n$ and thus $O(n^2)$ in total.[1]

Instead, following [DrH], we are able to use a divide-and-conquer approach to solve the problem more efficiently. To explain this, it is instructive to view the computation graphically. For this, consider the coordinate grid in Figure 2.2 with the l-axis in the horizontal direction and the k-axis in the vertical direction. We can consider the function Z defined on the grid with values $Z(l, k) = Z_l(k)$. Using recurrence (2.11), one sees immediately that the computation of $Z_l(k)$ (for $k < n - l$) only requires the prior computation of $Z_i(j)$ for (i, j) in the triangle defined by the vertices $(l, k), (0, k), (0, l + k)$.

Our goal is to compute the values $Z_l(0)$ for $0 \le l \le n-1$. As discussed above, initial computation of the first two columns, $\{Z_j(k) \mid j = 0, 1 \text{ and } 0 \le k \le n-1\}$ can be obtained in $18n \log^2 n + 3n \log n + 2n$ operations. In particular, the values $Z_0(0)$ and $Z_1(0)$ are obtained.

To compute the remaining $Z_l(0)$'s we wish to rewrite the recurrence (2.11) as a matrix equation. For any complex numbers α, β, and γ, define a $2n \times 2n$ matrix

[1] Strictly speaking, the recurrence can be applied to the initial sequence Z_0 to obtain the correct values of $Z_l(k)$ for $k < n - l$. For example, to get $Z_1(n-1)$ with equation (2.11) requires the value of $Z_0(n)$, which we do not have. This "edge effect" propagates as l increases but does not affect the values of the sequences that we actually need for the algorithm.

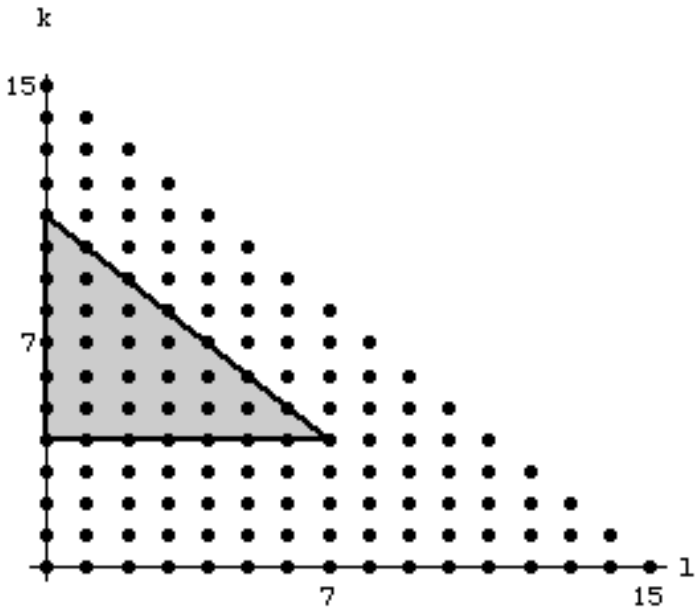


FIG. 2.2. *Computation of $Z_6(4)$ depends only on the computation of $Z_i(j)$ for (i,j) in the shaded triangle.*

$T_n(\alpha,\beta,\gamma)$ by

$$T_n(\alpha,\beta,\gamma) = \begin{pmatrix} 0 & I \\ \gamma I & \beta I + \alpha N \end{pmatrix},$$

where I denotes the $n \times n$ identity matrix and N denotes the $n \times n$ nilpotent matrix with ones on the superdiagonal and zeros elsewhere. Note that $T_n(\alpha,\beta,\gamma)$ is a block matrix composed of four $n \times n$ Toeplitz blocks. With this notation, recurrence (2.11) may be rewritten as

$$\begin{pmatrix} Z_l \\ Z_{l+1} \end{pmatrix} = T_n(a_l, b_l, c_l) \cdot \begin{pmatrix} Z_{l-1} \\ Z_l \end{pmatrix}. \tag{2.12}$$

Iteration of the recurrence is then realized as a product of such matrices, and so for any m,

$$\begin{pmatrix} Z_l \\ Z_{l+1} \end{pmatrix} = R_n(l-m-1, l) \cdot \begin{pmatrix} Z_{l-m-1} \\ Z_{l-m} \end{pmatrix}. \tag{2.13}$$

The product

$$R_n(l-m-1, l) = T_n(\alpha_l, \beta_l, \gamma_l) \cdots T_n(\alpha_{l-m-1}, \beta_{l-m-1}, \gamma_{l-m-1})$$

is still a block matrix made up of four $n \times n$ Toeplitz blocks. Consequently, the values of Z_{l-1} and Z_l can be computed from those of Z_{l-m-1} and Z_{l-m} by a single matrix–vector multiplication using a $2n \times 2n$ matrix with $n \times n$ Toeplitz blocks.

In particular,

$$\begin{pmatrix} Z_{\frac{n}{2}} \\ Z_{\frac{n}{2}+1} \end{pmatrix} = R_n\left(0, \frac{n}{2}\right) \cdot \begin{pmatrix} Z_0 \\ Z_1 \end{pmatrix}. \tag{2.14}$$

Recalling the discussion within the proof of Lemma 2.1, multiplication of an $n \times n$ Toeplitz matrix by a vector may be performed by standard FFT techniques using at most $9n(1 + \log n) + 2n$ operations. Four such matrix–vector multiplications are required to compute (2.14) so that an additional $36n(1 + \log n) + 8n$ operations are required to compute $Z_{\frac{n}{2}}$ and $Z_{\frac{n}{2}+1}$ and, in particular, $Z_{\frac{n}{2}}(0)$ and $Z_{\frac{n}{2}+1}(0)$ from our initial data of Z_0 and Z_1.

The point of this is to decompose the problem into two half-sized subproblems; we shall compute the Z_l for $l > n/2+1$ by applying equation (2.13) to $Z_{\frac{n}{2}}$ and $Z_{\frac{n}{2}+1}$. As we indicated previously in Figure 2.2, the values $Z_l(0)$ for $l > n/2+1$ depend only on the initial half-segments of the sequences $Z_{\frac{n}{2}}$ and $Z_{\frac{n}{2}+1}$. Similarly, the values $Z_l(0)$ for $l < n/2$ may be computed by applying the recurrence to the initial half-segments of the sequences Z_0 and Z_1. This is reexhibited in Figure 2.3, in which the diagonal lines display the dependence of the desired output $Z_l(0)$ on the various "subtriangles" in the grid for a problem of size $n = 16$.

Consequently, we see that to continue to obtain the remaining $Z_l(0)$'s, we need only keep $Z_j(k)$ for $0 \le k < n/2$ and $j = 0, 1, n/2$, and $n/2+1$. Thus step 2 proceeds by throwing away half of each of the sequences Z_0, Z_1, $Z_{\frac{n}{2}}$, and $Z_{\frac{n}{2}+1}$ and then computing $Z_{\frac{n}{4}}(k)$ and $Z_{\frac{n}{4}+1}(k)$ $(0 \le k < n/2)$ from the truncated sequences Z_0 and Z_1 and computing $Z_{\frac{3n}{4}}(k)$ and $Z_{\frac{3n}{4}+1}(k)$ $(0 \le k < n/2)$ similarly from the truncated sequences $Z_{\frac{n}{2}}$ and $Z_{\frac{n}{2}+1}$.

At the end of step 2, we own the first halves of the sequences Z_0, Z_1; $Z_{\frac{n}{4}}, Z_{\frac{n}{4}+1}$; $Z_{\frac{n}{2}}, Z_{\frac{n}{2}+1}$; $Z_{\frac{3n}{4}}, Z_{\frac{3n}{4}+1}$. Again, we throw away the latter halves of each (half-) sequence and continue by performing four multiplications by Toeplitz matrices of size $n/4$, and so on.

All of this is illustrated again by Figure 2.3 for a problem of size $n = 16$ in which we have indicated which values in the grid we have obtained after each step in the algorithm. Thus it shows that step 0 results in the sequences Z_0 and Z_1. After step 1, we have also obtained the sequences Z_8 and Z_9, which we then truncate in half while also cutting the sequences Z_0 and Z_1 in half. From this subset of data, we can then compute one quarter of the sequences Z_4, Z_5, Z_{12}, and Z_{13} and, after truncating each of the previous data sequences in half, a quarter of the sequences Z_0, Z_1, Z_8, and Z_9 as well. Finally, in the last step, we obtain the remaining pieces, one eighth of the sequences Z_2, Z_3, Z_6, Z_7, Z_{10}, Z_{11}, Z_{14}, and Z_{15}.

The complexity of the algorithm follows easily from an argument similar to that of Lemma 2.1. The process of "throwing away" is just a standard projection, so even if we include it in our estimate, it requires at most an additional $4n$ operations at the first step. Having cut the vectors Z_0, Z_1, $Z_{n/2}$, and $Z_{n/2+1}$ in half, we now have two identical subproblems that are half the size of the original problem, in which we computed $Z_{n/2}$ and $Z_{n/2+1}$ from Z_0 and Z_1. Thus if we let $T(n)$ denote the number of operations needed to compute the elements $Z_j(0)$ from the initial data of Z_0 and Z_1, we obtain the recurrence

$$T(n) = 36n(1 + \log n) + 8n + 4n + 2T\left(\frac{n}{2}\right) = 48n + 36n \log n + 2T\left(\frac{n}{2}\right). \tag{2.15}$$

Iterating (2.15) yields

$$T(n) \le 48n \log n + 36n \log^2 n. \tag{2.16}$$

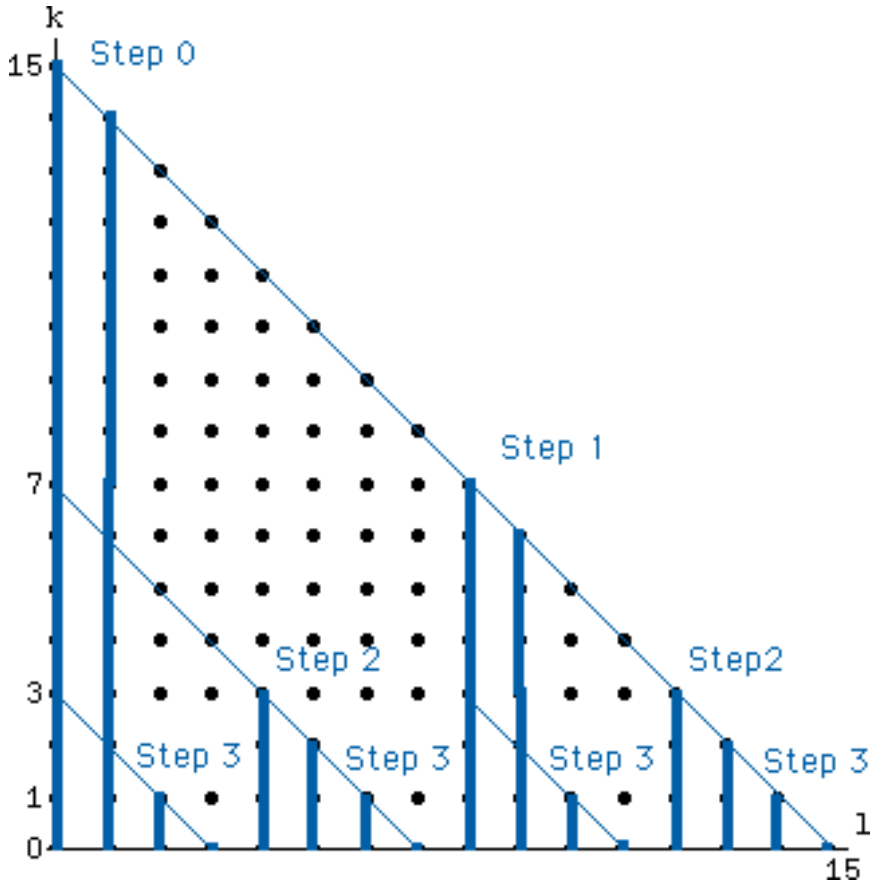


FIG. 2.3. *Computation of the $Z_l(0)$ for $l < n$ by a cascade of convolutions of decreasing size. The relevant ranges of Z are highlighted, and the step in which they are calculated is indicated.*

Finally, throwing in the operations needed to initially compute Z_0 and Z_1, we see that at most $48n \log n + 36n \log^2 n + 9n(1 + \log n) + 2n = O(n \log^2 n)$ operations are needed to compute the projections. □

Remarks. 1. As regards the storage requirements of the precomputed data structure, it is clear that all that is needed are the various Toeplitz matrices in the array:

$$
\begin{array}{ccccc}
 & & R_n\left(0, \frac{n}{2}\right) & & \\
 & R_{\frac{n}{2}}\left(0, \frac{n}{4}\right) & & R_{\frac{n}{2}}\left(\frac{n}{2}, \frac{3n}{4}\right) & \\
R_{\frac{n}{4}}\left(0, \frac{n}{8}\right) & R_{\frac{n}{4}}\left(\frac{n}{4}, \frac{3n}{8}\right) & R_{\frac{n}{4}}\left(\frac{n}{2}, \frac{5n}{8}\right) & R_{\frac{n}{4}}\left(\frac{3n}{4}, \frac{7n}{8}\right) & \\
 & & \vdots & & \\
R_4(0,2) & R_4(4,6) & \dots & \dots & R_4(n-4, n-2)\ .
\end{array}
$$

In fact, each should be suitably augmented to circulant matrices in order to effect an efficient matrix–vector multiplication. Analysis similar to that used in Remark 3

following the proof of Lemma 2.1 shows that this will require $O(n \log n)$ storage.

2. The Toeplitz matrices in the array above may be generated in $O(n \log^2 n)$ operations. The idea here is to build the larger R matrices at the top of this array from the smaller matrices lower down, which will have already been computed.

We start by filling in the bottom level of the array, building all of the matrices of the form $R_4(2j, 2j+2)$. Notice that we actually only need every other one of these for the lowest level of our data structure, but the rest are required for building the next level. These matrices may be combined pairwise, as detailed below, in order to obtain the matrices at the next level. Explicitly, we combine $R_4(4j, 4j+2)$ with $R_4(4j+2, 4j+4)$ to obtain $R_8(4j, 4j+4)$. Again, only half of these results are needed to fill out the second level of the data structure, and the rest are required for building the third level. Continuing in this fashion, we end up with all of the matrices that we need, up to $R_n(0, n/2)$.

The basic step is as follows: given matrices $R_p(j, j+m)$ and $R_p(j+m, j+2m)$, determine $R_{2p}(j, j+2m)$ efficiently. To see this, it is helpful to note that each of the four Toeplitz blocks of one of these matrices, say $R_p(j, j+m)$, may be written as a polynomial expression in the powers of the nilpotent matrix N of degree no more than m. The blocks are completely determined by the coefficients of these polynomials, and multiplication of a pair of R matrices corresponds to 2-by-2 matrix multiplication using polynomial arithmetic on the entries. Thus the entries of $R_{2p}(j, j+2m)$ may be computed from those of $R_p(j, j+m)$ and $R_p(j+m, j+2m)$ using fast polynomial arithmetic for polynomials of degree no more than m. Therefore, this may be done in $O(m \log m)$ operations.

The complexity of obtaining the entire array is now determined as in several similar calculations that we have done earlier; at the lth level (starting at the bottom) of $\log n$ levels, we have $n/2^l$ matrices to compute at $O(l2^l)$ complexity each. This leads to the given complexity of $O(n \log^2 n)$ for the entire array.

2.3. Some practical considerations and an example. The approach of section 2.2, while theoretically interesting, is, in fact, numerically rather suspect. Part of the problem comes from the step of first projecting the data vector onto the monomials. These functions, while linearly independent in exact arithmetic, are so close to being dependent as to be nearly useless in practice. See, for example, [Ga2] for a discussion of the condition number of expansions of polynomial functions in the monomial basis and other bases. Numerical experiments confirm that when the algorithm presented above is implemented in floating-point arithmetic, it can produce very unreliable answers for problems of modest size.

To treat this potential problem, we now prove a slight generalization of the recurrence technique of the last section that permits us to replace the monomials with other polynomial bases that satisfy simple constant coefficient recurrence relations. In particular, we have in mind the shifted Chebyshev polynomials. They satisfy the recurrence

$$T^*_{n+1} = (4x-2)T^*_n(x) - T^*_{n-1}(x). \tag{2.17}$$

Such a recurrence can be run in either the forward direction or the backward direction, in which case we obtain

$$T^*_{n-1}(x) = (4x-2)T^*_n(x) - T^*_{n+1}(x).$$

Consequently, we see that running the recurrence backwards from 0 allows the definition of $T^*_{-k}(x) = T^*_k(x)$ for all values of k. Notice that, in general, $T^*_k(x)$ is a

polynomial of degree $|k|$. Equality for negative and positive indices follows from the fact that the recurrence is the same in either direction.

More generally, any constant coefficient recurrence can be run in either direction, producing polynomials of degree $|k|$ for index k, assuming initial conditions that dictate polynomials of degree 0 and 1 for indices 0 and 1, respectively. For example, if a system satisfies the recurrence

$$p_{k+1}(x) = (\alpha x + \beta)p_k(x) + \gamma p_{k-1}(x),$$

then we obtain the "backward recurrence"

$$p_{k-1}(x) = -\frac{1}{\gamma}(\alpha x + \beta)p_k(x) + \frac{1}{\gamma}p_{k+1}(x).$$

This simple observation allows us to couple our algorithm with polynomials that satisfy such recurrences.

THEOREM 2.4. *Suppose that the polynomial families* $\{p_k(x)| - n \le k < n\}$ *and* $\{\Phi_l(x)|l = 0, \ldots, n-1\}$ *satisfy three-term recurrences*

$$p_{k+1}(x) = (\alpha x + \beta)p_k(x) + \gamma p_{k-1}(x), \tag{2.18}$$

$$\Phi_{l+1}(x) = (a_l x + b_l)\Phi_l(x) + c_l\Phi_{l-1}(x), \tag{2.19}$$

with $\deg(p_k) = |k|$ *and* $\Phi_0 = 1$, *and set* $\Phi_{-1} = 0$. *Suppose that the projections* $\langle f, p_k\rangle$ *are known,* $-n \le k < n$. *From this, the projections* $\langle f, \Phi_l\rangle$, $l = 0, \ldots, n-1$, *can be computed in* $O(n\log^2 n)$ *operations.*

Proof. Again, we may assume without loss of generality that the weight function is identically 1. Define the sequence Z_l for each $l = 0, 1, \ldots, n-1$ by

$$Z_l(k) = \langle f, p_k\Phi_l\rangle \tag{2.20}$$

for $k = -n, \ldots, 0, 1, \ldots, n-1$. Our goal is to obtain the values $Z_l(0) = \langle f, p_0\Phi_l\rangle$. Instead, we have the values $Z_0(k) = \langle f, p_k\Phi_0\rangle$. We hope to proceed by convolution as in Theorem 2.3 and push up from Z_0 to the higher sequences Z_l.

Recurrence (2.19) for the Φ_l shows that for $l > 0$,

$$\begin{aligned} Z_{l+1}(k) &= a_l\langle f, xp_k\Phi_l\rangle + b_l\langle f, p_k\Phi_l\rangle + c_l\langle f, p_k\Phi_{l-1}\rangle \\ &= a_l\langle f, xp_k\Phi_l\rangle + b_l Z_l(k) + c_l Z_{l-1}(k). \end{aligned}$$

Now use recurrence (2.18) for the p_k's to see that

$$\begin{aligned} \langle f, xp_k\Phi_l\rangle &= \frac{1}{\alpha}\langle f, p_{k+1}\Phi_l\rangle - \frac{\beta}{\alpha}\langle f, p_k\Phi_l\rangle - \frac{\gamma}{\alpha}\langle f, p_{k-1}\Phi_l\rangle \\ &= \frac{1}{\alpha}Z_l(k+1) - \frac{\beta}{\alpha}Z_l(k) - \frac{\gamma}{\alpha}Z_l(k-1). \end{aligned}$$

Therefore,

$$Z_{l+1}(k) = \frac{a_l}{\alpha}\left[Z_l(k+1) + \left(\frac{\alpha}{a_l}b_l - \beta\right)Z_l(k) - \gamma Z_l(k-1)\right] + c_l Z_{l-1}(k) \tag{2.21}$$

$$= u_l Z_l(k+1) + v_l Z_l(k) + w_l Z_l(k-1) + c_l Z_{l-1}(k). \tag{2.22}$$

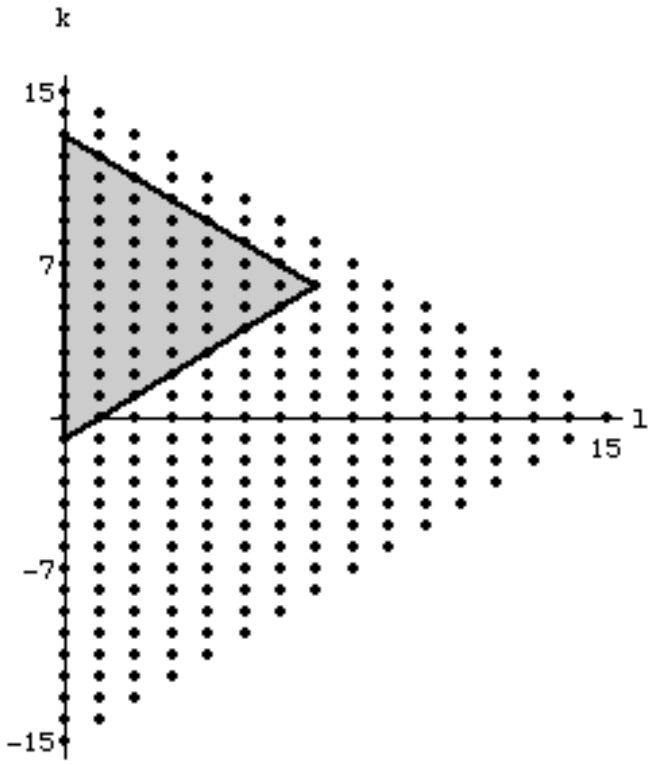


FIG. 2.4. *The computation of $Z_7(6)$ depends only on the computation of $Z_i(j)$ for (i, j) in the shaded triangle.*

Observe that the weights in expression (2.22) are independent of k. As in the case of Theorem 2.3, the sequence Z_{l+1} is obtained from the sequences Z_l and Z_{l-1} by convolving each with a fixed mask and then adding the resulting vectors. However, there is a difference. To describe this, it is again instructive to view the computation graphically. Following Theorem 2.3, we consider a function Z defined on a grid described by the l- and k-axes such that $Z(l, k) = Z_l(k)$. From this point of view, recurrence (2.22) indicates that a given value $Z_{l+1}(k)$ depends on knowing only any two adjacent vertical lines of data contained within the triangle determined by the vertices (l, k), $(0, k + l)$, and $(0, k - l)$. Figure 2.4 is an example.

Because in this case recurrence (2.22) "reaches" both down and up in k, slight modifications to the proof of Theorem 2.3 are required. Since the complexity counts are very similar, we will only point out the major changes and leave the details to the interested reader.

In analogy with Theorem 2.3, our goal is to express the full computation as a

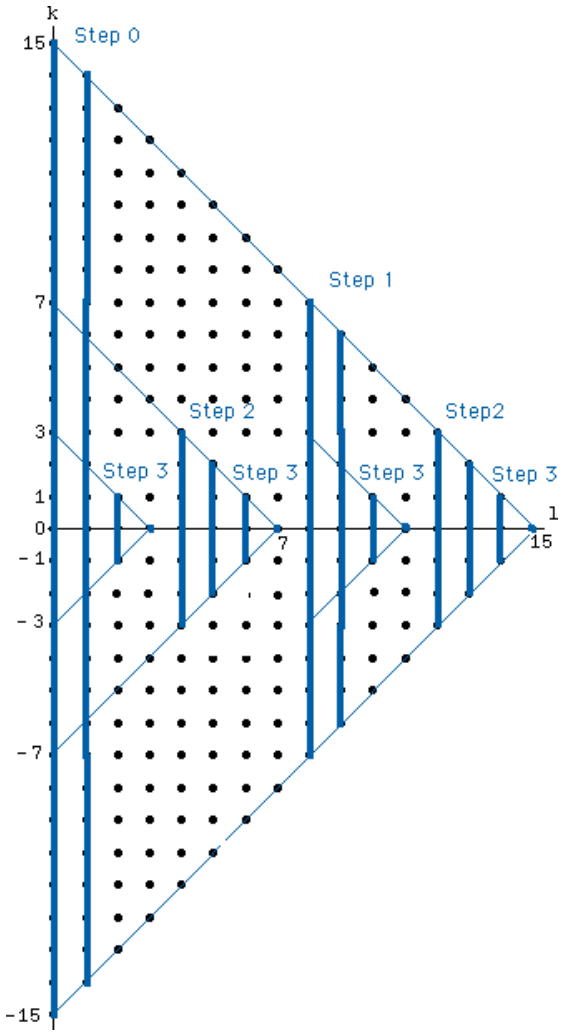


FIG. 2.5. *Computation of the $Z_l(0)$ for $l < 16$ by a cascade of convolutions of decreasing size. The relevant ranges of Z are highlighted, and the step in which they are calculated is indicated.*

divide-and-conquer algorithm. Figure 2.5 indicates how this can be accomplished. Starting with the full data of Z_0 and Z_1, we will construct $Z_{\frac{n}{2}}$ and $Z_{\frac{n}{2}+1}$. The values Z_j for $j < n/2 - 1$ depend on only half of the sequences Z_0 and Z_1, and similarly for $Z_{\frac{n}{2}+j}$, $Z_{\frac{n}{2}}$, and $Z_{\frac{n}{2}+1}$. Thus by keeping only the relevant values of these two pairs of sequences, we will have divided the original computation into two computations of half of the original's size. Continuing in this fashion, we ultimately obtain all values $Z_l(0)$. We need show only that the "divide" portion of the algorithm can be performed efficiently—that is, in $O(j \log j)$ operations for a problem of size j.

Again, we need the initial data of Z_0 and Z_1. We assume that Z_0 is given. By definition,

$$\begin{aligned} Z_1(k) &= \langle f, p_k \Phi_1 \rangle \\ &= \langle f, (a_0 x + b_0) p_k \rangle \\ &= a_0 \langle f, x p_k \rangle + b_0 Z_0(k). \end{aligned}$$

Notice that by using recurrence (2.18) for p_{k+1}, we may build the first summand out of at most three shifted copies of Z_0. Thus as a first step, a total of at most an additional $3n$ operations are needed to compute Z_1 from Z_0.

To compute the remaining Z_l's, we wish to rewrite recurrence (2.22) as a matrix equation. We can do this—up to "edge effects"—as we will now explain. For any complex numbers w, y, and z, let $C_n(w,y,z)$ denote the $2n \times 2n$ circulant matrix determined by setting the second row equal to $w, y, z, 0, \ldots, 0$,

$$C_n(w,y,z) = \begin{pmatrix} y & z & 0 & \cdots & 0 & 0 & w \\ w & y & z & \cdots & 0 & 0 & 0 \\ 0 & w & y & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & w & y & z \\ z & 0 & 0 & \cdots & 0 & w & y \end{pmatrix}. \tag{2.23}$$

Using the convention that

$$Z_l = \begin{pmatrix} Z_l(n-1) \\ \vdots \\ Z_l(0) \\ \vdots \\ Z_l(-n) \end{pmatrix},$$

consider the vectors Z'_l and Z'_{l+1} defined by the expression

$$\begin{pmatrix} Z'_l \\ Z'_{l+1} \end{pmatrix} = \begin{pmatrix} 0 & I_{2n} \\ c_l I_{2n} & C_n(w_l, v_l, u_l) \end{pmatrix} \cdot \begin{pmatrix} Z_{l-1} \\ Z_l \end{pmatrix}. \tag{2.24}$$

Notice that $Z'_l = Z_l$ but that Z'_{l+1} differs from Z_{l+1} by at worst the entries $Z_{l+1}(n-1)$ and $Z_{l+1}(-n)$. This is precisely the aforementioned "edge effect." If we define

$$A_n(l) = \begin{pmatrix} 0 & I_{2n} \\ c_l I_{2n} & C_n(w_l, v_l, u_l) \end{pmatrix}, \tag{2.25}$$

then we can make the following more general statement: a product of matrices of the form of $A_n(l+r)\cdots A_n(l)$ will still be composed of four $2n \times 2n$ circulant blocks, and the edge effects incurred when by multiplying this product by the vector $\left(\begin{smallmatrix} Z_l \\ Z_{l+1} \end{smallmatrix}\right)$ will still affect only (at most) the $r-1$ outermost values of Z_{l+r-1} and Z_{l+r}. More precisely, a simple inductive argument yields the following claim.

CLAIM. *Let $0 \le r < n$. Define Z'_{l+r-1} and Z'_{l+r} by*

$$\begin{pmatrix} Z'_{l+r-1} \\ Z'_{l+r} \end{pmatrix} = A_n(l+r)\cdots A_n(l) \cdot \begin{pmatrix} Z_{l-1} \\ Z_l \end{pmatrix}.$$

Then $Z'_i(j) = Z_i(j)$ for $j = -(n-r-2), \ldots, n-r-2$ and $i = l+r-1, l+r$.

The import of the claim is that if we compute the product $R \cdot \left(\begin{smallmatrix} Z_0 \\ Z_1 \end{smallmatrix}\right)$, where

$$R = A_n(0)\cdots A_n\left(\frac{n}{2}-1\right), \tag{2.26}$$

then we will correctly compute the values $Z_{\frac{n}{2+i}}(j)$ for $i = 0, 1$ and $-(n/2-1) \le j \le n/2-1$. Notice that this is precisely the data we need in order to effect the "divide"

portion of this algorithm (cf. Figures 2.4 and 2.5). The matrix R is composed of four $2n \times 2n$ circulant blocks, and thus the matrix–vector multiplication (2.26) requires at most $64n(1+\log n)+16n$ operations (cf. the discussion in the proof of Theorem 2.3). We continue by throwing away the upper and lower quarters of the vectors Z_0, Z_1, $Z'_{\frac{n}{2}}$, and $Z'_{\frac{n}{2}+1}$, forming two new subproblems of size $n/2$, and repeating the procedure. The analysis now follows that of the proof of Theorem 2.3. □

Remark. Notice that if initially only the projections onto the p_k's for positive k were given, then the projection onto the p_k's for negative k could obtained efficiently by using Theorem 2.3 applied to the backwards recurrence.

In particular, the shifted Chebyshev polynomials satisfy all of our requirements: they have a constant coefficient recurrence relation, fast projection is possible by Corollary 2.2, $T_k = T_{-k}$, and they possess relatively salutary numerical properties, even on a uniform grid. We have applied them in the case of the Hahn polynomial transforms with a great improvement in numerical accuracy over the method of Theorem 2.3.

Example 1: *Fast Hahn polynomial transform.* To illustrate Theorem 2.4 we proceed with an example and discuss its application to the specific case of the *Hahn polynomials*, a well-known discrete orthogonal polynomial family. This is, in fact, the relevant family of orthogonal polynomials for the California Lottery example discussed in section 1, and it provides the spherical functions for the k-sets of n-set graph. We begin by summarizing the relevant properties of the Hahn polynomials. We follow Stanton's notation of [St1], wherein a good bibliography for further sources is also contained. For general facts about orthogonal polynomials, Chihara's book [C] provides a friendly introduction to the subject.

The Hahn polynomials

$$Q_j(x;\alpha,\beta,N) = \sum_{i=0}^{j} \frac{(-j)_i\,(1+\alpha+\beta+j)_i\,(-x)_i}{i!\,(1+\alpha)_i\,(-N)_i}$$

are defined on the finite set $x = 0, 1, \ldots, N$ for $j = 0, 1, \ldots, N$. They are orthogonal with respect to the hypergeometric distribution

$$W(j;\alpha,\beta,N) = \binom{-1-\alpha}{-\beta+\mathrm{N}}\binom{-1-j}{\beta},$$

which is positive in the cases that we consider, $\alpha, \beta < -N$.

For the purpose of our calculations, we scale things so that all of the action takes place on the uniform grid in $[0,1]$, $\{k/(N+1) | k = 0, \ldots, N\}$. For fixed α, β, and N, define

$$Q_n^*(x) = Q_n(\,(N+1)x;\alpha,\beta,N)$$

for x in the grid. Then we have the three-term recurrence

$$Q_{n+1}^*(x) = \left(\frac{b_n + d_n - (N+1)x}{b_n}\right) Q_n^*(x) - \frac{d_n}{b_n} Q_{n-1}^*(x) \tag{2.27}$$

derived from that for Hahn polynomials, with

$$b_n = \frac{(n+\alpha+\beta+1)(n+\alpha+1)(N-n)}{(2n+\alpha+\beta+1)(2n+\alpha+\beta+2)}$$

and

$$d_n = \frac{n(n+\beta)(n+\alpha+\beta+N+1)}{(2n+\alpha+\beta)(2n+\alpha+\beta+1)}.$$

The calculation begins with projections of the data vector f onto the shifted Chebyshev polynomials $T_n^*(x)$. The projections are taken as l^2 inner products on the uniform grid in $[0,1]$, weighted by the hypergeometric distribution W on the grid. For the balance of this example, we will use this weighted l^2 inner product.

As described in Corollary 2.2, all of these projections may be done in $O(N \log^2 N)$ time. We now think of these as projections onto the functions $T_n^* Q_0^*$ and employ the techniques described in Theorem 2.4 for efficiently changing this information into the desired projections onto the Q_n^*'s.

We use the coefficients of recurrence (2.17) to construct the convolution masks described in Theorem 2.4. In practice, we also normalize the recurrences by the l^2 norms of the Hahn polynomials. Define

$$Z_l(k) = \frac{1}{\|Q_l^*\|} \langle f, T_k^* Q_l^* \rangle, \quad -N \le k < N,$$

with $T_k^* = T_{-k}^*$. Then

$$Z_l(k) = A_{l-1} Z_{l-1}(k) + B_{l-1} \{Z_{l-1}(k-1) + Z_{l-1}(k+1)\} + C_{l-1} Z_{l-2}(k),$$

with

$$A_l = \frac{b_l + d_l - \frac{N+1}{2}}{b_l} \frac{\|Q_l^*\|}{\|Q_{l+1}^*\|}.$$

One-step convolution coefficient masks for producing a sequence Z_l from lower index sequences Z_{l_0} and Z_{l_0+1} may be generated by an appropriate recursion. In the derivation that follows, M_j and N_j denote the one-step convolution masks for obtaining Z_{j+1} from Z_j and Z_{j-1}; in the present case, $M_j = \{\ldots, 0, B_j, A_j, B_j, 0, \ldots\}$ and $N_j = \{\ldots, 0, 0, C_j, 0, 0, \ldots\}$. Then

$$\begin{aligned}
Z_{l+1} &= M_l * Z_l + N_l * Z_{l-1} \\
&= M_l * (M_{l-1} * Z_{l-1} + N_{l-1} * Z_{l-2}) + N_l * Z_{l-1} \\
&= (M_l * M_{l-1} + N_l) * Z_{l-1} + (M_l * N_{l-1}) * Z_{l-2} \\
&= M_l(2) * Z_{l-1} + N_l(2) * Z_{l-2}.
\end{aligned}$$

Likewise we can continue all the way down to Z_{l_0} and Z_{l_0+1}:

$$Z_{l+1} = M_l(l - l_0) * Z_{l_0+1} + N_l(l - l_0) * Z_{l_0}$$

with multistep masks $M_l(j)$ and $N_l(j)$ defined recursively by

$$\begin{aligned}
M_l(j+1) &= M_l(j) * M_{l-j} + N_l(j), \\
N_l(j+1) &= M_l(j) * N_{l-j}
\end{aligned}$$

with initial conditions

$$M_l(1) = M_l, \qquad N_l(1) = N_l.$$

Using this, we generate and prestore the various multistep masks as described in Theorem 2.4, and run the tree of convolutions. Figure 2.6 shows the resulting steps of a small calculation, starting with Z_0 and Z_1, and then filling out the various other sequences in the tree.

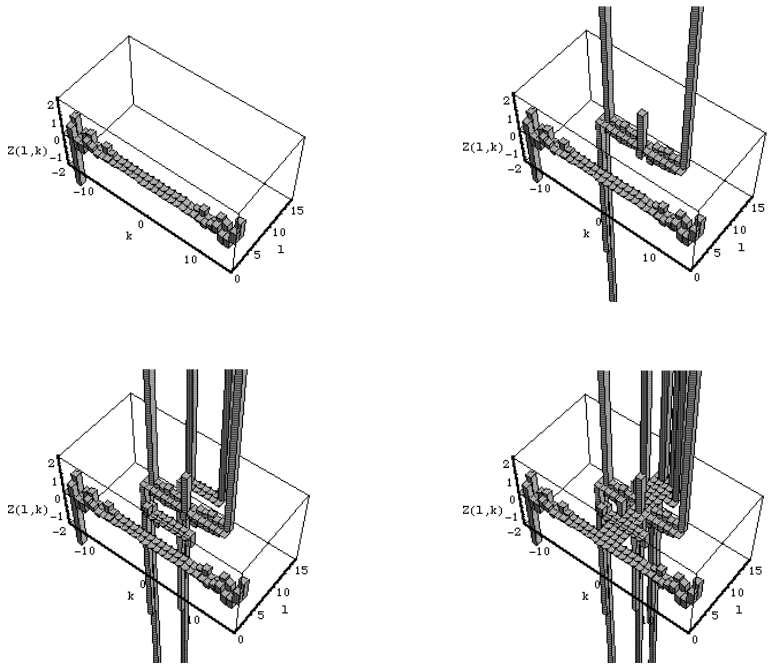


FIG. 2.6. *Stages in the computation of the Z_l for the Hahn polynomials. Z_0 is projection onto the Chebyshev polynomials. The desired transform values are the $Z_l(0)$'s. This example is the transform of Q_8^*.*

3. Fast spherical transforms for distance transitive graphs. We now employ the various results of the last section to obtain results leading up to a fast Fourier transform for distance transitive graphs. As mentioned, these results are of interest in several problems of data analysis, such as the California Lottery example mentioned in section 1.

Briefly, the setting is as follows. Let G be a finite group acting as isometries on a finite graph X with distance function $d(\cdot,\cdot)$ given by the usual measure of shortest path. Recall that X is *distance transitive* (for G) if given any two pairs of points (x,y), $(x',y') \in X$ such that $d(x,y) = d(x',y')$, there exists $s \in G$ such that $(sx,sy) = (x',y')$. Let $L^2(X)$ denote the vector space of complex-valued functions on X. Then $L^2(X)$ affords a linear representation of G by left translation. In this case, $L^2(X)$ may be decomposed into irreducible subspaces

$$L^2(X) = V_0 \oplus \cdots \oplus V_N,$$

where N is the maximum distance between two points in X.

Fix a basepoint $x_0 \in X$ and let H be the stabilizer of x_0 in G. Then X is in a natural 1:1 correspondence with G/H and $L^2(G/H)$ is the vector space of right H-invariant functions on G. If Ω_k denotes the sphere of radius k about x_0, then the algebra of functions constant on each Ω_k is isomorphic to the algebra of H-biinvariant functions on G, denoted $L^2(H\backslash G/H)$.

The fact that $L^2(X)$ is multiplicity free is equivalent to the existence in each V_k of a unique function ϕ_k, constant on each Ω_k and normalized by $\phi_k(x_0) = 1$. Classically,

the function ϕ_k is called the kth *spherical function* on X (cf. [He] and the references therein, as well as the remarks at the close of section 3.1.)

Let $x_j \in \Omega_j$. Then in analogy with the classical case (see, e.g., [He, Chapter 4]), we define for any function f constant on each Ω_k the *spherical transform* of f at ϕ_i to be the sum

$$\widehat{f}(\phi_i) = \sum_{j=0}^{N} f(x_j)\phi_i(x_j)|\Omega_j|.$$

The *discrete spherical transform* (DST) of f is the collection of transforms $\{\widehat{f}(\phi_i)\}_i$.

Direct computation of the DST requires $O(N^2)$ operations. For large N, this cost quickly becomes prohibitive. In this section, we give an algorithm that computes the DST for spherical functions from distance transitive graphs in $O(N \log^2 N)$ operations. By the same techniques, we may also invert the transform in the same number of operations and consequently obtain an $O(N \log^2 N)$ algorithm for convolution of two H-biinvariant functions.

Section 4 will show that the problem of finding an FFT for distance transitive graphs may be reduced to that of finding an efficient projection onto the spherical functions, an orthogonal family of special functions on the graph. This section discusses the fast DST. We begin by giving an expanded review of the group theoretic background (section 3.1), sufficient to present the fast algorithm in section 3.2.

3.1. Background and notation. In the interest of making this paper as self-contained as possible, we sketch the group-theoretic background and notation. We mainly follow Stanton's expositions [St1, St3], which are very accessible and provide a wealth of references. For the necessary graph-theoretic terminology—with special attention paid to distance transitive graphs—see Biggs [Bi]. Serre's book [S] provides a nice introduction to the representation theory of finite groups.

Throughout, X is a *distance transitive graph*. Thus X is a finite metric space with *integer-valued* distance d (taken to be the usual distance of the shortest path on the graph) with G a group of isometries of X (acting on the left) satisfying the property of *two-point homogeneity*:

If $x, y, x', y' \in X$ are such that $d(x, y) = d(x'y')$, then there exists $g \in G$ such that $gx = x'$ and $gy = y$.

In this case, X is also said to be a *two-point homogeneous space* (with respect to G and d).

It is perhaps instructive at this point to recall the analogy here with the usual 2-sphere in $\mathbf{R}^3$. In this case, we know that any pair of points on the sphere which are a fixed distance apart can be moved into any two other such points by a rotation—or isometry of $\mathbf{R}^3$. Thus the rotation group $SO(3)$ acts two-point homogeneously on the 2-sphere. Stanton's papers [St1, St3] do a terrific job of spelling out these analogies.

Thus since $d(x, x) = 0 = d(x', x')$ for every $x, x' \in X$, there is a $g \in G$ such that $gx = x'$, i.e., G acts *transitively* on X. Let $x_0 \in X$ denote a fixed *basepoint* and $H = \{g \in G \mid gx_0 = x_0\}$ denote the stabilizer subgroup of x_0. Since G acts transitively on X, any element $x \in X$ can be written as $x = sx_0$ for some $s \in G$ and if $sx_0 = x = tx_0$, then $s^{-1}tx_0 = x_0$ and hence $s^{-1}t \in H$ and $t \in sH$. Thus there is a natural correspondence between X and the coset space G/H, associating any element $x = sx_0$ with the coset sH. Keeping in mind the analogy with the 2-sphere, consider the subgroup $H = SO(2) < SO(3)$ of rotations that fix the north pole. Cosets are represented by circles of latitude, with coset representatives in 1:1 correspondence

with a choice of points at all possible latitudes. Any two points of the same latitude differ by only a rotation about the north pole.

Under the correspondence between points in X and cosets in G/H, the vector space of complex-valued functions on X, $L^2(X)$, is isomorphic to the vector space of complex-valued functions on G/H, $L^2(G/H)$, by defining

$$f(sH) \equiv f(sx_0).$$

Any function on G/H then naturally extends to $\tilde{f}$, a function defined on the entire group G, by declaring it to be constant on cosets,

$$\tilde{f}(s) \equiv f(sH).$$

It is not hard to check that $\tilde{f}$ is well defined and that

$$\tilde{f}(sh) = \tilde{f}(s). \tag{3.1}$$

Thus $\tilde{f}$ is a (*right*) *H-invariant function on G.* Conversely, it is easy to see that for any subgroup $H < G$, the subspace of $L^2(G)$ satisfying (3.1)) is equivalent to the $L^2(G/H)$. Following along the analogy with the 2-sphere, $L^2(SO(2)\backslash SO(3)/SO(2))$ can be identified with the subspace of functions on the 2-sphere which are constant on latitudes.

The action of G on X gives rise to a *representation* of G in $L^2(X)$ by translation. More precisely, for every $s \in G$ and $f \in L^2(X)$, a new function $\rho(s)f \in L^2(X)$ can be defined by

$$[\rho(s)f](x) \equiv f(s^{-1}x).$$

In this manner, each $\rho(s)$ defines a linear operator on $L^2(X)$ such that $\rho(st) = \rho(s)\rho(t)$ and thus is a *representation* of the group G. This representation is, in general, *reducible* in the sense that there exist proper subspaces of $W_1, \ldots, W_r$ such that each W_i is *G-invariant* (i.e., $\rho(s)W_i \subset W_i$ for all $s \in G$) and $L^2(X) = W_1 \oplus \cdots \oplus W_r$. A subspace W is *G-irreducible* if it contains no proper G-invariant subspaces. For $L^2(X)$, an irreducible decomposition can be understood by considering the action of H.

Let Ω_k denote the sphere of radius k about x_0,

$$\Omega_k = \{x \in X \mid d(x, x_0) = k\}. \tag{3.2}$$

Then the Ω_k's are exactly the H-orbits in X. That is, X is the disjoint union of the Ω_k's and if $x, y \in \Omega_k$, then $hx = y$ for some $h \in H$ and, conversely, if $hx = y$, then $x, y \in \Omega_k$ for some k. In the case of the 2-sphere, the associated "Ω_k's" are the circles of latitude.

A subspace W of $L^2(X)$ is *H-invariant* if for all $f \in W$, $\rho(h)f \in W$ for all $h \in H$. Thus a function constant on each of the Ω_k's is H-invariant and vice versa. Hence if N is the maximum distance occurring in X, then the subspace of H-invariant functions in $L^2(X)$ is of dimension $N+1$. This may be immediately translated into a statement about functions on G: under the association of $L^2(X)$ with right H-invariant functions on G, the H-invariant functions of $L^2(X)$ become functions which are *both* left and right H-invariant, i.e., functions $f \in L^2(G)$ such that

$$f(h_1 s h_2) = f(s)$$

for all $h_1, h_2 \in H$. Such H-biinvariant functions are then naturally associated with the space of functions constant on double cosets $H\backslash G/H$ and thus are denoted as $L^2(H\backslash G/H)$. Hence we see that $L^2(H\backslash G/H)$ is of dimension $N+1$.

Note that the subspaces $L^2(X)$ and $L^2(H\backslash G/H)$ are, in fact, algebras under convolution: if $f, g \in L^2(X)$, then define $f \star g \in L^2(X)$ by

$$f \star g(x) = \sum_{s \in G} \tilde{f}(s) g(s^{-1}x), \tag{3.3}$$

where $\tilde{f}$ is the function on G derived from f by extending it to be constant on cosets of H as in (3.1). It is easy to check that if f and g are H-invariant, then their convolution is as well.

As a complex representation space for a finite group G, $L^2(X)$ may be decomposed into G-irreducible subspaces (cf. [S, section 1.4, Theorem 2]). In the general situation of decomposing the permutation representation arising from a finite group G acting transitively on a set X, this irreducible decomposition need not be unique. However, under the assumption of distance transitivity, an irreducible decomposition is indeed unique.

THEOREM 3.1 ([St1, Theorem 2.6]). *Let all notation be as above. Then as a representation of G, the space $L^2(X)$ has a unique decomposition into irreducible subspaces as*

$$L^2(X) = V_0 \oplus V_1 \oplus \cdots \oplus V_N,$$

where the V_i are all pairwise inequivalent—that is, the representation of G in $L^2(X)$ is multiplicity-free.

The proof of this theorem is not crucial for the main results, but it is worth remarking that it depends only on the fact that G acts two-point homogeneously on X and as such is a general fact about permutation representations (cf. [W, Chapter 5, section 29]). In this context, a proof follows from the fact that Theorem 3.1 is equivalent to the statement that the *intertwining algebra* of the permutation representation is commutative. (The intertwining algebra is the algebra of linear operators T that commute with the permutation representation ρ—i.e., the set of T such that $T\rho(s) = \rho(s)T$ for all $s \in G$.) To show this commutativity, choose a basis for $L^2(X)$ consisting of "delta functions" or point masses concentrated at single points. For $0 \le k \le N$, define the $|X| \times |X|$ matrices D_k by

$$D_k(x,y) = \begin{cases} 1 & \text{if } d(x,y) = k, \\ 0, & \text{otherwise.} \end{cases} \tag{3.4}$$

Straightforward combinatorial arguments (e.g., [St1, St3]) show that the D_k's commute and span the intertwining algebra, which must then be of dimension $N+1$. Moreover, the algebra is generated by D_1 since the D_k's satisfy the following three-term recurrence [St1, St3],

$$D_1 D_i = c_{i+1} D_{i+1} + a_i D_i + b_{i-1} D_{i-1}, \tag{3.5}$$

where

$$\begin{aligned} c_{i+1} &= |\{z : d(x,z) = 1, d(y,z) = i\}| \quad \text{for any fixed } x, y \text{ with } d(x,y) = i+1, \\ a_i &= |\{z : d(x,z) = 1, d(y,z) = i\}| \quad \text{for any fixed } x, y \text{ with } d(x,y) = i, \\ b_{i-1} &= |\{z : d(x,z) = 1, d(y,z) = i\}| \quad \text{for any fixed } x, y \text{ with } d(x,y) = i-1. \end{aligned}$$

Consequently, D_i is a polynomial in D_1,

$$D_i = p_i(D_1). \tag{3.6}$$

Since D_1 is real symmetric and generates an algebra of dimension $N+1$, it has distinct real nonzero eigenvalues $\{\lambda_0 < \cdots < \lambda_N\}$. Also, since D_1 is in the intertwining algebra for the representation ρ and the intertwining algebra is commutative, the λ_i eigenspaces must be the G-irreducible subspaces.

The importance of Theorem 3.1 is that it shows that in this special case, the isotypic and irreducible decompositions coincide so that the irreducible decomposition is independent of the choice of basis. It is a direct consequence of Theorem 3.1 that in each V_i there exists a unique one-dimensional H-fixed subspace (e.g., see [D, p. 54, Theorem 9]). We choose a basis vector ϕ_i for this subspace by demanding that $\phi_i(x_0) = 1$. Note that this is possible since the previous reference—or Frobenious reciprocity (cf. [S])—shows the existence of some nonzero H-fixed element (hence constant on each of the Ω_k) $\phi_i \in V_i$. Since

$$D_1\phi_i = \lambda_i\phi_i \tag{3.7}$$

and ϕ_i is constant on each Ω_k, the fact that ϕ_i is not identically zero implies that $\phi_i(x_0) \neq 0$. Hence ϕ_i can be normalized so as to assume $\phi_i(x_0) = 1$.

Note that ϕ_i may be viewed as either an H-invariant function on X or an H-biinvariant function on G. As an H-invariant function on X, it is constant on the spheres Ω_k. We call ϕ_i the ith *spherical function.* By counting, we see that the spherical functions give a basis for the H-invariant functions on X.

For distance transitive graphs, the polynomial nature of the spherical functions is derived from the self-same property of the commuting algebra for the representation of G in $L^2(X)$.

As shown in [St1], by evaluating the eigenvalue equation (3.7) at Ω_k, we move recurrence (3.5) to the ϕ_i's:

$$\lambda_i\phi_i(\Omega_k) = \gamma_k\phi_i(\Omega_{k+1}) + \alpha_k\phi_i(\Omega_k) + \beta_k\phi_i(\Omega_{k-1}), \tag{3.8}$$

where for any $x \in \Omega_k$,

$$\begin{aligned}
\gamma_k &= |\{z : z \in \Omega_{k+1} \text{ and } d(x,z) = 1\}|,\\
\alpha_k &= |\{z : z \in \Omega_k \text{ and } d(x,z) = 1\}|, \quad \text{and}\\
\beta_k &= |\{z : z \in \Omega_{k-1} \text{ and } d(x,z) = 1\}|.
\end{aligned}$$

An examination of the combinatorics yields

$$|\Omega_k|\phi_i(\Omega_k) = p_k(\lambda_i). \tag{3.9}$$

The orthogonality relations for the spherical functions take on the form

$$\frac{1}{|X|}\sum_{k=0}^{N}\phi_i(\Omega_k)\phi_j(\Omega_k)|\Omega_k| = \delta_{ij}\frac{1}{\dim(V_j)}, \tag{3.10}$$

where $\phi_i(\Omega_k)$ makes sense since ϕ_i is constant on Ω_k. In addition, we have a "dual orthogonality relation,"

$$\frac{1}{|X|}\sum_{i=0}^{N}\phi_i(\Omega_k)\phi_i(\Omega_j)\dim(V_i) = \delta_{kj}\frac{1}{|\Omega_k|}. \tag{3.11}$$

We summarize this with the following theorem.

THEOREM 3.2. *Let X be a finite distance transitive graph with respect to a group of isometries G. Let*

$$L^2(X) = V_0 \oplus \cdots \oplus V_N$$

be the isotypic and hence irreducible decomposition of $L^2(X)$ so that N is the maximum distance occurring in X. Let ϕ_i be the spherical function for V_i and λ_i as in (3.7) *and Ω_k as in* (3.2). *Then*

$$|\Omega_k|\phi_i(\Omega_k) = p_k(\lambda_i)$$

for some set of orthogonal polynomials $\{p_k(x) \mid 0 \le k \le N\}$ such that p_k is of degree k and the polynomials satisfy a three-term recurrence (3.5).

Remarks. 1. It is worth noting that while the spherical functions ϕ_i are determined by the polynomial functions described above, Theorem 3.2 does not say that $\phi_i(\Omega_k)$ is polynomial in k (i.e., equal to some fixed polynomial evaluated at $N+1$ fixed points). Rather, this is a statement that the dual functions p_k are an orthogonal polynomial system with respect to the weights $\dim(V_i)$, although for many examples this will also be true of the spherical functions (cf. section 4). As a consequence, in general the inverse spherical transform is always a projection onto polynomials and would thus benefit from general results on fast projection onto polynomials. In fact, such an algorithm can then be transposed to obtain an algorithm for the direct transform with the same complexity, whether or not the direct transform is obtained by projection onto polynomials.

2. The existence of spherical functions depended only on the fact that the permutation representation of G on $L^2(G/H)$ was multiplicity free. This is often summarized by saying that (G, H) form a *Gelfand pair*. Gelfand pairs have been much studied of late. See Gross [Gr] for a survey with applications to number theory and Diaconis [D] for applications to statistics and probability as well as an extensive bibliography. Helgason's book [He] gives a thorough introduction to the study of spherical functions for compact and locally compact groups with a full bibliography.

As remarked, the polynomial nature follows from the polynomial relation of the D_i's. This is true in a slightly more general setting than distance transitive graphs. It can be extended to finite two-point homogeneous spaces in which the metric satisfies some technical properties (cf. [St1, p. 90]).

Spherical functions may also be computed by character-theoretic methods. Travis [Tr] generalizes this approach to construct "generalized" spherical functions attached to any pair of characters ψ and χ for representations of a finite group G and subgroup H, respectively.

3. While the existence of the spherical functions depends only on the multiplicity-free nature of the representation, their expression as certain sampled values of orthogonal polynomial sets and consequent relations via the recurrence (3.8) use the integer-valued property of the metric.

4. We should point out that many of the problems and results discussed here may be phrased in the language of association schemes. Bannai and Ito [BI] give a beautiful treatment of this subject. We have not pursued this connection.

3.2. Fast spherical transforms on distance transitive graphs. The terminology is that of section 3.1. In particular, recall that $H < G$ is the isotropy group of a fixed basepoint $x_0 \in X$ so that $L^2(H\backslash G/H)$ is identified with the subspace of $L^2(X)$ of functions constant on spheres around x_0.

THEOREM 3.3. *For distance transitive graphs with maximum distance N, the spherical transform and its inverse can be computed in at most $O(N \log^2 N)$ operations.*

Proof. Let $f \in L^2(H\backslash G/H)$. Then the components of $\widehat{f}$ are

$$\widehat{f}(\phi_i) = \sum_{k=0}^{N} f(\Omega_k)\phi_i(\Omega_k)|\Omega_k| = \sum_{k=0}^{N} f(x_k)p_k(\lambda_i) \tag{3.12}$$

using

$$|\Omega_k|\phi_i(\Omega_k) = p_k(\lambda_i)$$

from section 3.1.

This has the form of polynomial evaluation, or multiplication by the transpose of the generalized Vandermonde matrix associated with the polynomials p_k and the evaluation points λ_i:

$$\begin{pmatrix} \widehat{f}(\phi_0) \\ \widehat{f}(\phi_1) \\ \vdots \\ \widehat{f}(\phi_N) \end{pmatrix} = \begin{pmatrix} p_0(\lambda_0) & p_1(\lambda_0) & \cdots & p_N(\lambda_0) \\ p_0(\lambda_1) & p_1(\lambda_1) & \cdots & p_N(\lambda_1) \\ \vdots & \vdots & \cdots & \vdots \\ p_0(\lambda_N) & p_1(\lambda_N) & \cdots & p_N(\lambda_N) \end{pmatrix} \begin{pmatrix} f(\Omega_0)|\Omega_0| \\ f(\Omega_1)|\Omega_1| \\ \vdots \\ f(\Omega_N)|\Omega_N| \end{pmatrix} = P(\Lambda)^t \mathbf{f}_\Omega.$$

Likewise, the inverse spherical transform can be written in terms of $(P(\Lambda)^t)^t = P(\Lambda)$ itself; by the dual orthogonality equation (3.11),

$$f(\Omega_k) = \frac{1}{|X|} \sum_{i=0}^{N} \widehat{f}(\phi_i)\phi_i(\Omega_k) \dim(V_i).$$

Using equation (3.10), we rewrite this as

$$|\Omega_k| f(\Omega_k) = \frac{|\Omega_k|}{|X|} \sum_{i=0}^{N} \widehat{f}(\phi_i) p_k(\lambda_i) \dim(V_i).$$

Up to scaling, this has the form of projection onto the polynomials p_k, or multiplication by the generalized Vandermonde matrix $P(\Lambda)$. The three-term recurrence relation is already known explicitly; consequently, the methods of section 2 apply directly to this computation. Thus the inverse spherical transform can be done in $O(N \log^2 N)$ operations.

We can also conclude that the transpose problem, the direct spherical transform, is also fast. This follows, for instance, from the results of Bshouty et al. [BKK]. Roughly speaking, they observe that if a straight-line algorithm can compute a matrix product $M \cdot \mathbf{v}$ in time $O(T(n))$ (where M is an $n \times n$ matrix and $\mathbf{v}$ is a column vector), then there exists an algorithm computing the transposed product $M^t\mathbf{v}$ in the same time. They note that any such algorithm may be encoded in a directed graph and in this context the "transposed" algorithm is essentially given by reversing all arrows on the graph.

In our case, this has a concrete interpretation in matrix language. Namely, the inverse spherical transform algorithm amounts to a factorization of the matrix $P(\Lambda)$. The order reversed and transposed factors comprise a factorization of $P(\Lambda)^t$, which

effects the direct transform by matrix multiplication. The individual factors have block structure with Toeplitz blocks, and this does not change upon their transposition. Therefore, the direct transform is computed with the same complexity as the inverse. □

Finally, consider the convolution of two functions $f, g \in L^2(H\backslash G/H)$. Recall that this is simply the convolution over G (cf. equation (3.3)) of H-biinvariant functions which is again H-biinvariant. As such, it makes sense to compute the DST of the convolution. It can be shown that for such functions,

$$|X|\widehat{f \star g}(\phi_i) = \widehat{f}(\phi_i)\widehat{g}(\phi_i),$$

with a quick proof using the multiplicative properties of a Fourier transform on a finite group and the fact that a spherical function is a particular matrix coefficient for the symmetry group (cf. [D, pp. 54–56]). Thus we obtain the following result.

THEOREM 3.4. *Let $f, g \in L^2(H\backslash G/H)$. Given as initial input the spherical transforms $\{\widehat{f}(\phi_i)\}_i$, the function f may be recovered in $O(N \log^2 N)$ operations. The convolution $f \star g$ can be computed in at most $O(N \log^2 N)$ operations. The implied constants here are universal and depend only on the universal constant for the FFT.*

Example 2: *Two particular distance transitive graphs.*

1. *K-sets of an N-set.* This is the collection of size-K subsets $x \subset \{1, 2, \ldots, N\}$, $|x| = K$, with metric $d(x, y) = K - |x \cap y|$, S_N as the symmetry group. The K-sets are made into a graph in the usual way: put edges between those K-sets whose distance is 1. Assuming that $2K < N$, the largest distance is K, occurring for disjoint K-sets. Picking a basepoint K-set x_0, the stabilizer is $\cong S_K \times S_{N-K}$, so we may identify this graph with $S_N/(S_K \times S_{N-K})$.

This is a distance transitive graph of size $\binom{N}{K}$. The weights in the spherical function orthogonality relations are the sizes of the spheres at fixed distances from the basepoint: $|\Omega_j| = \binom{N-K}{j}$. Recall that the spherical functions satisfy

$$|\Omega_j|\phi_i(\Omega_j) = p_j(\lambda_i)$$

for a family of orthogonal polynomials defined on the collection of eigenvalues for the adjacency operator. We saw that the spherical functions are eigenvectors of the adjacency operator and that this provides the three-term recurrence from which the λ_i's and p_j's may be determined:

$$\begin{aligned}\lambda_i p_j(\lambda_i) = j^2 p_{j-1}(\lambda_i) + (K-j)(N-K-j)p_{j+1}(\lambda_i) \\ + [K(N-K) - j^2 - (K-j)(N-K-j)]p_j(\lambda_i).\end{aligned}$$

This is the recurrence for the Eberlein or dual Hahn polynomials. From the case where $j = 1$, we get $\lambda_i = K(N-K) - i(N+1-i)$. We can now compute the spherical functions as

$$\phi_i(\Omega_j) = c \sum_{l=0}^{i} \frac{(N-K-i+1)_l}{(-K)_s} \binom{K-j}{l} \binom{j}{-i-l}.$$

This is the Hahn polynomial $Q_i(j; K-N-1, -K-1, K)$ [KM, St1]. As we have already seen, these are orthogonal polynomials satisfying a three-term recurrence (2.27). The norms can be determined from $\dim V_i = \binom{N}{i} - \binom{N}{i-1}$.

In this case, we have the recurrence relation needed to make the forward spherical transform fast, as discussed in Example 1 of the last section. On the other hand, we

know that in every case we have the recurrence relation for the p_k's needed to make the inverse transform fast.

2. *The hypercube*: The hypercube $\mathbb{Z}_2^N$ has the Hamming metric and symmetries consisting of the hyperoctohedral group, the semidirect product $\mathbb{Z}_2$ wr S_N. This is a distance transitive graph. The three-term recurrence from the adjacency operator eigenequation is

$$\lambda_i p_j(\lambda_i) = j\lambda_i p_{j-1}(\lambda_i) + (N-j)p_{j+1}(\lambda_i),$$

from which we determine the eigenvalues $\lambda_i = N - 2i$ and the spherical functions

$$\phi_i(\Omega_j) = c\sum_{l=0}^{i} \binom{j}{l}\binom{N-j}{i-l}(-1)^{i-l}.$$

Again, these are orthogonal polynomials with respect to weights $|\Omega_j| = \binom{N}{j} = \dim V_j$, known as the Krawtchouk polynomials $K_i(j, 1/2, N)$. This is a particularly nice case in that the dual polynomials and the spherical polynomials are the same up to a constant. Thus the forward spherical transform and its inverse are effectively the same, and can be done fast using the three-term recurrence as before.

4. Computation of isotypic projections. As remarked in section 1, a fast DST algorithm has applications in spectral analysis for data on distance transitive graphs. In this section, we wish to explain this in a little more detail. Diaconis' book [D, especially Chapter 8] is an excellent introduction to these ideas and also gives many pointers to the existing literature. (See [DR] for a more thorough account of the following discussion as well as for other approaches to this problem.)

In general, let G be a finite group acting transitively on a set

$$X = \{x_0, x_1, \ldots, x_n\}.$$

The action of G on X then determines the associated permutation representation ρ of G in $L^2(X)$ given by translation,

$$(\rho(s)f)(x) = f(s^{-1}x).$$

If η and η' are two representations of G, then we will write $\eta \sim \eta'$ if η is equivalent to η'. Recall that the *isotypic decomposition* of $L^2(X)$,

$$L^2(X) = V_0 \oplus \cdots \oplus V_N,$$

is defined by the following:

(1) *Each V_i is G-invariant.*

(2) *If $\rho^{(i)} := \rho\,|_{V_i}$, then*

$$\rho^{(i)} \sim m_i\eta_i,$$

where the η_i's are irreducible representations of G, m_i is some positive integer, and $i \neq j$ implies that $\eta_i \not\sim \eta_j$.

The isotypic decomposition is canonical in the sense that it is independent of the choice of basis for $L^2(X)$.

In appropriate settings, data on a such a finite homogeneous space X is viewed as a vector $f \in L^2(X)$. The relevant statistics then become the projections of f onto the

isotypic components. To state things a bit more sharply, the computational problem is as follows:

Given as input $f = (f(x_0), \dots, f(x_n))$, compute for $i = 0, \dots, N$ the projection of f into the ith isotypic, denoted $f^{(i)} (\in V_i)$, as

$$f^{(i)} = (f^{(i)}(x_0), \dots, f^{(i)}(x_n)).$$

One way to proceed here is via character theory. Let χ_i be the character corresponding to η_i. Then [S, Theorem 2.7],

$$f^{(i)}(x) = \frac{\chi_i(1)}{|G|} \sum_{s \in G} \chi_i(s) f(s^{-1}x). \tag{4.1}$$

This gives a naïve upper bound of $|X||G|$ operations to compute all projections. Note that in the example of the California Lottery of section 1, this would give $49!\binom{49}{6}$ operations, which is beyond the capabilities of any machine.

Careful analysis of this formula permits significant speedups, even in the general case.

THEOREM 4.1 ([DR, Theorem 2.4]). *For any fixed i, the projection onto the ith isotypic can be computed in at most $|X|^2$ operations. Consequently, all projections can be computed in at most $(N+1)|X|^2$ operations.*

Let us now specialize the case of interest, in which $L^2(X)$ has a multiplicity-free decomposition so that the isotypic decomposition is actually an irreducible decomposition. As previously, let H denote the isotropy subgroup of the chosen basepoint x_0 and let $\{s_0 = 1, \dots, s_n\}$ denote a fixed set of coset representatives. Let $\{\phi_i\}_{i=0}^N$ denote the corresponding spherical functions. In this case, the character formula (4.1) can be rewritten as

$$f^{(i)}(x_k) = \frac{|H|}{|G|} \chi_i(1) \sum_{j=0}^{N} f_k(\Omega_j) \phi_i(\Omega_j) |\Omega_j|, \tag{4.2}$$

where

$$f_k(\Omega_j) = \frac{1}{|\Omega_j|} \sum_{x \in \Omega_j} f(s_k^{-1} x). \tag{4.3}$$

The computation of the f_k is a preprocessing of the original data whose complexity will depend on the geometry of X. In any event, this may always be done in at most $|X|^2$ additions.

Finally, using the notation of the previous sections, we rewrite (4.2) as

$$f^{(i)}(x_k) = \frac{|H|}{|G|} \chi_i(1) \widehat{f}_k(\phi_i).$$

Consequently, using the results of section 3, we have the following result.

THEOREM 4.2. *Let X be a distance transitive graph for G with maximum distance N. Then the set of projections $f^{(i)} \in V_i$ $(i = 0, \dots, N)$ may be computed in at most*

$$O(|X|^2 + |X| N \log^2 N)$$

operations.

Remark. In some cases, the projections can also be computed combinatorially using variations of the discrete Radon transform (cf. [D, DR, BDRR]).

5. Final remarks. Of course, distance transitive graphs are not the only source of orthogonal polynomials. Another example closely related to this setting is the construction of orthogonal polynomial systems from group actions on posets [St2]. If P is a ranked poset, then $L^2(P)$ has a natural decomposition into "harmonics." In [St2], Stanton shows that under certain assumptions about the automorphism group G of P, the space of functions on the maximal elements of P gives a multiplicity-free representation of G. Again these functions can be written in terms of discretely sampled orthogonal polynomials.

More generally, one might consider other special function systems satisfying recurrence relations that arise in a continuum setting. Results similar to those of this paper can be obtained, provided that an appropriate sampling theorem is available to reduce the computations to finite ones. Some initial work along this line for the case of the homogeneous space $SO(3)/SO(2)$ may be found in [DrH]. Maslen has recently extended these ideas to more general compact groups [M].

Beyond the example of spectral analysis, we are actively seeking other applications for the techniques presented here. A recent book by Nikiforov, Suslov, and Uvarov [NSU] cites a large number of tantalizing possibilities.

Acknowledgments. We would like to thank the referees for their suggestions following a careful reading of the manuscript. We also thank Peter Kostelec and Doug Warner for their help with the final typesetting.

REFERENCES

[BI] E. Bannai and T. Ito, *Algebraic Combinatorics* I: *Association Schemes*, Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1984.

[BDRR] R. Bailey, P. Diaconis, D. Rockmore, and C. Rowley, *Representation Theory and ANOVA*, Technical Report, Department of Mathematics, Harvard University, Cambridge, MA, 1994.

[Bi] N. Biggs, *Algebraic Graph Theory*, Cambridge Tracts in Mathematics 67, Cambridge University Press, Cambridge, UK, 1974.

[BM] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, Elsevier, New York, 1975.

[Bo] J. Boyd, *Chebyshev and Fourier Spectral Methods*, Springer-Verlag, New York, 1989.

[BKK] N. Bshouty, M. Kaminski, and D. Kirkpatrick, *Addition requirements for matrix and transposed matrix products*, J. Algorithms, 9 (1988), pp. 354–364.

[C] T. S. Chihara, *An Introduction to Orthogonal Polynomials*, Gordon and Breach, New York, 1978.

[D] P. Diaconis, *Group Representations in Probability and Statistics*, Institute for Mathematical Statistics, Hayward, CA, 1988.

[DR] P. Diaconis and D. Rockmore, *Efficient computation of isotropic projections for the symmetric group*, in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 11, L. Finkelstein and W. Kantor, eds., AMS, Providence, RI, 1992, pp. 87–104.

[DrH] J. R. Driscoll and D. Healy, *Computing Fourier transforms and convolutions on the* 2*-sphere*, Adv. Appl. Math., 15 (1994), pp. 202–250.

[ER] D. F. Elliott and K. R. Rao, *Fast Transforms: Algorithms, Analyses, Applications*, Academic Press, New York, 1982.

[Ga1] W. Gautschi, *On the construction of Gaussian quadrature rules from modified moments*, Math. Comp., 24 (1970), pp. 245–260.

[Ga2] W. Gautschi, *Questions of numerical condition related to polynomials*, in Studies in Numerical Analysis, G. Golub, ed., MAA, Washington, DC, 1984, pp. 140–177.

[Gr] B. Gross, *Some applications of Gelfand pairs to number theory*, Bull. Amer. Math. Soc., 24 (1991), pp. 277–301.

[HMR] D. Healy, S. Moore, and D. Rockmore, *Efficiency and Stability Issues in the Numerical Computation of Fourier Transforms on the* 2*-Sphere*, Technical Report

PCS-TR94-222, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1993.

[HMMRT] D. Healy, D. Maslen, S. Moore, D. Rockmore, and M. Taylor, *Applications of FFT's on the* 2*-sphere*, manuscript.

[He] S. Helgason, *Groups and Geometric Analysis*, Academic Press, New York, 1984.

[Hi] N. J. Higham, *Fast solution of Vandermonde-like systems involving orthogonal polynomials*, IMA J. Numer. Anal., 8 (1988), pp. 473–486.

[KM] S. Karlin and J. L. McGregor, *The Hahn polynomials, formulas and an application*, Scripta Math., 26 (1961), pp. 33–46.

[M] D. Maslen, *Fast Transforms and Sampling for Compact Groups*, Ph.D. thesis, Department of Mathematics, Harvard University, Cambridge, MA, 1993.

[MHR] S. Moore, D. Healy, and D. Rockmore, *Symmetry stabilization for polynomial evaluation and interpolation*, Linear Algebra Appl., 192 (1993), pp. 249–299.

[NSU] A. F. Nikiforov, S. K. Suslov, and V. B. Uvarov, *Classical Orthogonal Polynomials of a Discrete Variable*, Springer-Verlag, New York, 1991.

[N] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, New York, 1982.

[OS] A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice–Hall, Englewood Cliffs, NJ, 1989.

[P] V. Pan, *Matrix and polynomial computations*, SIAM Review, 34 (1992), pp. 225–262.

[S] J. P. Serre, *Linear Representations of Finite Groups*, Springer-Verlag, New York, 1986.

[St1] D. Stanton, *Orthogonal polynomials and Chevalley groups*, in Special Functions: Group Theoretical Aspects and Applications, R. Askey, T. Koornwinder, and W. Schempp, eds., Reidel, Dordrecht, The Netherlands, 1984, pp. 87–128.

[St2] D. Stanton, *Harmonics on posets*, J. Combin. Theory Ser. A, 40 (1985), pp. 136–149.

[St3] D. Stanton, *An introduction to group representations and orthogonal polynomials*, in Orthogonal Polynomials, P. Nevai, ed., Kluwer Academic Publishers, Norwell, MA, 1990, pp. 419–433.

[Te] C. Temperton, *On scalar and vector transform methods for global spectral models*, Monthly Weather Review, 119 (1991), pp. 1303–1307.

[TAL] R. Tolimieri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transform and Convolution*, Springer-Verlag, New York, 1989.

[Tr] D. Travis, *Spherical functions on finite groups*, J. Algebra, 29 (1974), pp. 65–76.

[W] H. Wielandt, *Finite Permutation Groups*, Academic Press, New York, 1964.

SIAM J. COMPUT
Vol. 26, No. 4, pp. 1100–1119, August 1997

009

DOUBLY LOGARITHMIC COMMUNICATION ALGORITHMS FOR OPTICAL-COMMUNICATION PARALLEL COMPUTERS*

LESLIE ANN GOLDBERG†, MARK JERRUM‡, TOM LEIGHTON§, AND SATISH RAO¶

Abstract. In this paper, we consider the problem of interprocessor communication on parallel computers that have optical communication networks. We consider the *completely connected optical-communication parallel computer* (OCPC), which has a completely connected optical network, and also the *mesh-of-optical-buses parallel computer* (MOB-PC), which has a mesh of optical buses as its communication network. The particular communication problem that we study is that of realizing an *h-relation.* In this problem, each processor has at most h messages to send and at most h messages to receive. It is clear that any 1-relation can be realized in one communication step on an OCPC. However, the best previously known p-processor OCPC algorithm for realizing an arbitrary h-relation for $h > 1$ requires $\Theta(h + \log p)$ expected communication steps. (This algorithm is due to Valiant and is based on earlier work of Anderson and Miller.) Valiant's algorithm is optimal only for $h = \Omega(\log p)$, and it is an open question of Geréb-Graus and Tsantilas whether there is a faster algorithm for $h = o(\log p)$. In this paper, we answer this question in the affirmative and we extend the range of optimality by considering the case in which $h \leq \log p$. In particular, we present a $\Theta(h + \log\log p)$-communication-step randomized algorithm that realizes an arbitrary h-relation on a p-processor OCPC. We show that if $h \leq \log p$, then the failure probability can be made as small as $p^{-\alpha}$ for any positive constant α. We use the OCPC algorithm as a subroutine in a $\Theta(h + \log\log p)$-communication-step randomized algorithm that realizes an arbitrary h-relation on a $p \times p$-processor MOB-PC. Once again, we show that if $h \leq \log p$, then the failure probability can be made as small as $p^{-\alpha}$ for any positive constant α.

Key words. parallel algorithms, randomized algorithms, routing, optical networks

AMS subject classifications. 68Q22, 68R05

PII. S0097539793259483

1. Introduction.

The p-processor *completely connected optical-communication parallel computer* (p-OCPC) consists of p processors, each of which has its own local memory. The p processors can perform local computations and can communicate with each other by message passing. A *computation* on this computer consists of a sequence of *communication steps.* During each communication step, each processor can perform some local computation and then send one message to any other processor. If a given processor is sent one message during a communication step, then it receives this

*Received by the editors December 9, 1993; accepted for publication (in revised form) August 22, 1995. A preliminary version of this paper appeared in *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM, New York, 1993, pp. 300–309.
http://www.siam.org/journals/sicomp/26-4/25948.html

†Department of Computer Science, University of Warwick, Coventry CV4 7AL, England (leslie@dcs.warwick.ac.uk). The research of this author was performed at Sandia National Laboratories and was supported by Department of Energy contract DE-AC04-76DP00789.

‡Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, United Kingdom (mrj@dcs.ed.ac.uk). This work was performed while this author was visiting the NEC Research Institute, Princeton, NJ. The research of this author was supported by UK Science and Engineering Research Council grant GR/F 90363 and by the ESPRIT Working Group "RAND."

§Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (ftl@math.mit.edu). The research of this author was supported by Air Force contract AFOSR-F49620-92-J-0125 and DARPA contracts N00014-91-J-1698 and N00014-92-J-1799.

¶NEC Research Institute, 4 Independence Way, Princeton, NJ 08540 (satish@research.nj.nec.com).

message successfully, but if it is sent more than one message, then the transmissions are garbled and it does not receive any of the messages.

Eshaghian [Esh88], [Esh91] first studied the computational aspects of parallel architectures with complete optical interconnection networks. The OCPC model is an abstract model of computation which formalizes important properties of such architectures. It was first introduced by Anderson and Miller [AM88] and Eshaghian and Kumar [EK88] and has subsequently been studied by several authors, including Valiant [Val90], Geréb-Graus and Tsantilas [GT92], and Gerbessiotis and Valiant [GV92] (though not always under the name OCPC). The feasibility of the OCPC from an engineering point of view is discussed in [AM88], [GT92], and [Rao92]. See also the references in [McC93].

In the first part of this paper, we study the problem of interprocessor communication on an OCPC. In particular, we study the problem of realizing *h-relations*. This problem arises in both the direct implementation of specific parallel algorithms [AM88], and the simulation of shared-memory models, such as the PRAM, on more realistic distributed-memory models [Val90]. An h-relation [Val90] is a communication problem in which each processor has up to h messages that it wishes to send to other processors (assumed distinct). The destinations of these messages can be arbitrary except that each processor is the destination of at most h messages. The goal is to design a fast p-OCPC algorithm that can realize an arbitrary h-relation. Anderson and Miller [AM88] have observed that an h-relation can easily be realized in h communication steps if all of the processors are given *total* information about the h-relation to be realized.[1] A more interesting (and perhaps more realistic) situation arises if we assume that initially each processor knows about only the messages that it wants to send and the processors learn about the h-relation only by receiving messages from other processors. This is the usual assumption, and it is the one that will be made here.

An OCPC algorithm for realizing h-relations is said to be *direct* if it has the property that the only messages that are exchanged by the processors are the original messages of the h-relation and these messages are sent only to their destinations. In this paper, we prove the following:

1. The expected number of communication steps taken by *any* direct algorithm for realizing h-relations on a p-OCPC is $\Omega(h + \log p)$.

2. An arbitrary h-relation can be realized on a p-OCPC in $\Theta(h + \log\log p)$ communication steps with high probability. (Valiant has shown that an arbitrary h-relation can be realized in $\Theta(h + \log p)$ communication steps, which deals with the case where $h \geq \log p$; our algorithm (see Theorem 1) covers the case of sublogarithmic h.)

It is easy to see that any 1-relation can be realized in one communication step on an OCPC. Anderson and Miller [AM88] were the first to consider the problem of realizing h-relations for $h > 1$. They discovered a direct p-OCPC algorithm that runs for $\Theta(h)$ communication steps and delivers most of the messages in an arbitrary h-relation. In particular, the expected number of messages remaining after Anderson and Miller's algorithm is run is $O(p)$. Anderson and Miller were interested in the special class of h-relations in which each of the messages with a given destination has a unique label ℓ in the range $1 \leq \ell \leq h$. For this class of h-relations, Anderson and Miller also discovered a deterministic $\Theta(h + \log p)$-communication-step algorithm that

[1] To see this, model the communications between the p processors viewed as sources and the p processors viewed as destinations as the edges of a bipartite graph of order $2p$. Since the graph has maximum degree h, it is edge colorable with h colors, which can be interpreted as time steps.

delivers all of the messages in any h-relation that contains only $O(p)$ messages. Thus their algorithms can be combined to obtain an algorithm that realizes an arbitrary h-relation from their special class in $\Theta(h + \log p)$ expected communication steps.

Valiant [Val90] considered the general problem of realizing h-relations for $h > 1$. He discovered a $\Theta(h + \log p)$-expected-communication-step p-OCPC algorithm that realizes an arbitrary h-relation. Valiant's algorithm consists of the first phase of Anderson and Miller's algorithm followed by a second phase which redistributes the remaining $O(p)$ messages using parallel prefix, sorts them, and then sends them to the correct destinations. The second phase of Valiant's algorithm takes $\Theta(h + \log p)$ communication steps.

Prior to this work, Valiant's algorithm was the fastest known OCPC algorithm that can realize an arbitrary h-relation for $h > 1$. It is not direct, however. The fastest known *direct* OCPC algorithm for realizing arbitrary h-relations is due to Geréb-Graus and Tsantilas [GT92] and runs in $\Theta(h + \log p \log\log p)$ expected communication steps. In this paper, we show that every direct OCPC algorithm for realizing h-relations takes $\Omega(h + \log p)$ expected communication steps. Furthermore, we describe a $\Theta(h + \log\log p)$-communication-step p-OCPC algorithm that can realize an arbitrary h-relation and we show that if $h \leq \log p$, then the failure probability can be made as small as $p^{-\alpha}$ for any positive constant α. (The Θ notation does not hide any large constants in the running time of our algorithm.)

In this paper we also consider a model of computation known as the *mesh-of-optical-buses parallel computer* (MOB-PC). The $p \times p$ MOB-PC consists of p^2 processors, organized in a $p \times p$ array. The processors can perform local computations and can communicate with each other by message passing. As in the case of the OCPC, a computation on this computer consists of a sequence of communication steps. During each communication step, each processor can perform some local computation and then send one message. Unlike the OCPC, the MOB-PC has the restriction that the destination of each message must be in the row or the column of its sender. (The reason for considering the MOB-PC is that this restriction makes it much easier to build than a p-OCPC (see [Rao92]).) As in the case of the OCPC, if a given processor is sent one message during a communication step, then it receives this message successfully, but if it is sent more than one message, then the transmissions are garbled and it does not receive any of the messages.

The $p \times p$ mesh of optical buses is a member of a class of networks studied by Wittie [Wit81] and suggested by Dowd as a method for optical interconnects [Dow91]. Rao studied the MOB-PC in [Rao92] and used a result of Leighton and Maggs to show that for $h \geq \log p$, an arbitrary h-relation can be realized on a $p \times p$ MOB-PC in $\theta(h)$ communication steps. In this paper, we describe a $\Theta(h + \log\log p)$-communication-step randomized algorithm that realizes an arbitrary h-relation on a $p \times p$ MOB-PC, and we show that if $h \leq \log p$, then the failure probability can be made as small as $p^{-\alpha}$ for any positive constant α.

The following experiment gives the intuition underlying our lower bound for direct OCPC algorithms and our OCPC algorithm (which is a subroutine in our MOB-PC algorithm). Suppose that two processors P_i and P_j of an OCPC are both trying to send messages to a third processor P_d and that they adopt the following direct strategy. During each communication step, processors P_i and P_j both flip fair coins. If P_i's coin comes up "heads," then P_i sends its message to P_d. Similarly, if P_j's coin comes up "heads," then P_j sends its message to P_d. On any given communication step, P_d has probability $\frac{1}{2}$ of successfully receiving a message. Therefore, the probability

that P_d has not received any messages after t communication steps is 2^{-t}. Now suppose that we use a similar strategy to realize a 2-relation in which each processor is the destination of two messages. After t communication steps, we will expect to have $p\,2^{-t}$ processors that have received no messages at all. Therefore, it will take $\Omega(\log p)$ communication steps to realize the 2-relation.

Intuitively, the reason that so much time is needed is that the events are "too independent." In particular, the fact that most of the other messages are already delivered will not make it easier for P_i and P_j to send their messages to P_d. In order to obtain a sublogarithmic OCPC algorithm, we adopt the following strategy. We divide the set of p destinations into disjoint "target groups." During the first part of our algorithm, we send each message in the h-relation to a randomly chosen processor within the target group containing its destination. As more and more messages are delivered to a given target group, the probability that any remaining message is successfully delivered to the group in one communication step increases. Once all of the messages have been delivered to their target groups, we solve the smaller problem of realizing an h-relation within each target group.

Our OCPC algorithm consists of four procedures. The first three procedures deliver the messages to their target groups and the last procedure realizes smaller h-relations within the target groups.

The methods that we use to deliver messages to target groups rely upon the fact that the number of messages being sent to each group is small compared to the size of the group. The first procedure of our algorithm (the "thinning" procedure) establishes this condition by delivering most of the messages in the h-relation to their final destinations. The thinning procedure is a direct OCPC algorithm and it is based on Anderson and Miller's algorithm. Proving that it satisfies the appropriate conditions requires a probabilistic analysis of dependent events. To do the analysis, we use the "method of bounded differences" [McD89], [Bol88]. Note that Matias and Vishkin's "thinning out" procedure [MV91] is similar to one step of our "thinning" procedure; in their case, just one step suffices since the density of messages is much lower.

After the thinning procedure has terminated the number of messages remaining will be $O(p/(h \log\log p))$ with high probability. The purpose of the second procedure (the "spreading" procedure) is to redistribute these messages so that each sender has at most one message to send. After the spreading procedure terminates, the third procedure delivers the remaining messages to their target groups. The bulk of the messages are delivered using a probabilistic tool called "approximate compaction." After the approximate compaction terminates, the number of messages that have not been delivered to their target groups will be $O(p/\log^2 p)$ with high probability. Each remaining message is copied $\log p$ times and the processors are reallocated so that $\log p$ processors can work together to send each message to its target group. (The approximate compaction technique and the copying technique were first used in PRAM algorithms such as those described in [CDHR89] and in [GM91] and [MV91]. In this work, we require a smaller failure probability for approximate compaction than previous authors because our target groups are only polylogarithmic in size and we need to bound the probability of failure in any group.)

At the end of the third procedure, the communication problem that remains consists of one h-relation within each target group. These h-relations could be realized in $\Theta(h + \log\log p)$ communication steps by simultaneously running the second phase of Valiant's algorithm within each target group, substituting a deterministic EREW

sorting algorithm such as Cole's parallel merge sort (see [Col88]) for the randomized sorting algorithm that Valiant uses.

Our fourth procedure is an alternative algorithm for realizing the h-relations within the target groups. It does not rely on efficient deterministic $O(\log p)$-time EREW sorting and it is therefore likely to be faster in practice. The algorithm is as follows. Each target group is subdivided into disjoint subgroups. Our "thinning," "spreading," and "deliver to target group" procedures are run simultaneously in each target group to deliver the messages in that group to the appropriate subgroups. The communication problem remaining with each subgroup is an h-relation, and this h-relation is realized using the second phase of Valiant's algorithm, in which the sorting is done by Bitonic sort. With high probability, the proportion of target groups for which this strategy delivers all of the messages is at least $1 - 1/\log^c p$ for a sufficiently large constant c. The processors from these target groups are then reallocated and used to help the unsuccessful target groups finish realizing their h-relations. After the processors are reallocated, each unsuccessful target group sorts its messages using an enumeration sort due to Muller and Preparata [MP75] which is fast in practice as well as in theory. The sorted messages are then delivered to their destinations.

The structure of this paper is as follows. In section 2, we describe the OCPC algorithm in detail. We demonstrate that it uses $\Theta(h + \log\log p)$ communication steps, and we prove that if $h \leq \log p$, then the probability that any messages are left undelivered can be made as small as $p^{-\alpha}$ for any positive constant α. In section 3, we give the proof of the lower bound for direct OCPC algorithms. Finally, in section 4, we describe the MOB-PC algorithm. We demonstrate that it uses $\Theta(h + \log\log p)$ communication steps, and we prove that if $h \leq \log p$, then the probability that any messages are left undelivered can be made as small as $p^{-\alpha}$ for any positive constant α.

2. The OCPC algorithm. In this section, we prove the following theorem.

THEOREM 1. *Suppose that $h \leq \log p$. There is a randomized algorithm that realizes an arbitrary h-relation on a p-OCPC in $\theta(h + \log\log p)$ communication steps. The failure probability of the algorithm is $p^{-\alpha}$. The constant $\alpha > 0$ may be chosen arbitrarily.*

Before we can define the OCPC algorithm, we must describe the partition of the set $\{P_1, \ldots, P_p\}$ of processors into disjoint "target groups." The size of each target group will be a polynomial in $\log(p)$. To be precise, let c_1 denote a sufficiently large integer (the size of c_1 will depend upon the failure probability that we wish to obtain) and let k denote $\lceil \log^{c_1} p \rceil$. We will divide the p processors into approximately p/k target groups, each of size about k. To simplify the presentation, we will assume that k divides p[2] and we will define the ℓth target group for ℓ in the range $0 \leq \ell < p/k$ to be the set $\{P_{k\ell}, \ldots, P_{k\ell+k-1}\}$. We will define the target group of any given message to be the target group containing the destination of the message, and we will say that the message is destined for that target group.

The algorithm consists of the following four procedures:

- *Thinning.* At the beginning of the algorithm, the number of messages destined for any given target group may be as high as hk. The goal of the thinning procedure is to deliver most of the messages to their final destinations so that by the end of the procedure the number of undelivered messages destined for any given target group is at most $k/(h\lceil c_2 \log\log p \rceil)$ for a sufficiently large

[2] The case in which k does not divide p presents no real difficulty. In this case, the target groups should be defined in such a way that all but one of the groups has size k and the size of the remaining group is between k and $2k$.

constant c_2. If $h \leq \log p$, then this can be done in $\Theta(h + \log(h) \log\log\log(p))$ steps with probability at least $1 - p^{-\alpha}$, where the constant in the running time depends upon α and c_2.

- *Spreading.* At the end of the thinning procedure, there will be only $O(p/(h \log\log p))$ undelivered messages. However, some senders may have as many as h undelivered messages. The spreading procedure spreads these out so that each sender has at most one to send. This can be done in $\Theta(h + \log\log p)$ communication steps with probability at least $1 - p^{-\alpha}$, where the constant in the running time depends upon α.
- *Deliver to target groups.* This procedure delivers all of the undelivered messages to their target groups. After it terminates, each sender will have at most two undelivered messages to send and the destination of each undelivered message will be within the target group containing its sender. The procedure can be implemented in $\Theta(\log\log p)$ communication steps with probability at least $1 - p^{-\alpha}$, where the constant in the running time depends upon α.
- *Deliver within target groups.* This procedure delivers all messages to their final destinations. It can be implemented deterministically in $\Theta(h + \log\log p)$ steps by running the second phase of Valiant's algorithm twice in each target group. However, this implementation may be slow in practice. In section 2.4, we describe an alternate implementation which runs in $\Theta(h + \log\log p)$ communication steps and succeeds with probability at least $1 - p^{-\alpha}$. (The constant in the running time depends upon α.)

We will use the following tool in the implementation of our algorithm. (For similar tools, see [CDHR89], [GM91], and [MV91].)

DEFINITION 1. *The* (s, β, Δ) approximate compaction problem *is defined as follows. Given*

- *a p-OCPC in which at most s senders each have one message to send and*
- *a set of* βs *receivers which is known to all of the senders,*

deliver all but up to Δ *of the messages to the set of receivers in such a way that each receiver receives at most one message. (During the delivery, messages may only be sent from the original senders to the* βs *receivers.)*

LEMMA 1. *For any positive constant* α*, there is a positive constant* c_2 *such that the* $(s, \lceil c_2 \log\log p \rceil, \Delta)$ *approximate compaction problem can be solved in* $O(\log\log p)$ *communication steps with failure probability at most* $\alpha^{-\sqrt{s}} + s^{-\alpha(\Delta+1)}$.[3]

Using the (s, β, Δ) approximate compaction algorithm, we can accomplish a variety of tasks. For example (following [CDHR89] and [GM91]), we use the algorithm to allocate $\lfloor \log p \rfloor$ processors to each message once the number of undelivered messages is reduced to $p/\lfloor \log p \rfloor^2$. In the proof of Lemma 1, we use Lemma 2 (see below) and the following definition.

DEFINITION 2. *The* (s, β, Δ) approximate collection problem *is defined to be the same as the* (s, β, Δ) *approximate compaction problem except that we remove the requirement that each receiver receives at most one message.*

LEMMA 2. *For any positive constant* α*, there is a positive constant* c_2' *such that the* $(s, 36, \Delta)$ *approximate collection problem can be solved in at most* $\lceil c_2' \log\log p \rceil$ *communication steps with failure probability at most* $\alpha^{-\sqrt{s}} + s^{-\alpha(\Delta+1)}$.

Proof of Lemma 1. Let α be any positive constant and let $c_2 = 36c_2' + 1$, where

[3] In fact, there is a positive constant c_2 such that the (s, c_2, Δ) approximate compaction problem can be solved in $O(\log\log s)$ communication steps with small failure probability, but Lemma 1 is sufficient for our purposes.

c_2' is the constant associated with α in Lemma 2. Suppose that we are given an instance of the $(s, \lceil c_2 \log\log p\rceil, \Delta)$ approximate compaction problem. Partition the set of receivers into $\lceil c_2' \log\log p\rceil$ disjoint sets $R_1, R_2, \ldots$, each of size at least $36s$. Since by Lemma 2 the $(s, 36, \Delta)$ approximate collection problem can be solved in at most $\lceil c_2' \log\log p\rceil$ communication steps with failure probability at most $\alpha^{-\sqrt{s}} + s^{-\alpha(\Delta+1)}$, there is an algorithm with this failure probability that delivers all but up to Δ of the messages to the receivers in R_1 in only $\lceil c_2' \log\log p\rceil$ steps. To solve the $(s, \lceil c_2 \log\log p\rceil, \Delta)$ approximate compaction problem, simply run this algorithm, substituting the set R_i for R_1 on the ith communication step of the algorithm, thus guaranteeing that each receiver receives at most one message. □

Proof of Lemma 2. We say a sender is *active* initially if it contains a message. Our algorithm proceeds in a number of similar communication steps, where in step i each active sender sends its message to a random location in the set of receivers. Each sender that successfully transmitted a message is considered *inactive.*

Let m denote $36s$. We must show that there are at most Δ active messages when the algorithm terminates. We use the following claim.

CLAIM 1. *Let c be a positive integer. If there are at most m/r active senders left at step i, then the probability that there will be $f = \max\{\lceil m/r^{3/2}\rceil, \Delta + 1\}$ or more active senders left at step $i + 2c$ is at most $(2\mathrm{e}/\sqrt{r}\,)^{cf}$.*

We prove Claim 1 by imagining that in a certain step the m/r active senders make their random choice of destination in some fixed order. For there to be f active senders that do not transmit their message, there must be $\lceil f/2\rceil$ times at which a sender chooses the same receiver as one chosen by a previous sender in this order. The probability of choosing the same receiver as a previous sender is at most $(m/r)/m = 1/r$. Thus the probability of $\lceil f/2\rceil$ such events occurring is bounded above by

$$\begin{aligned}
\binom{\lfloor m/r\rfloor}{\lceil f/2\rceil}\left(\frac{1}{r}\right)^{\lceil f/2\rceil} &\le \left(\frac{2em}{rf}\right)^{\lceil f/2\rceil}\left(\frac{1}{r}\right)^{\lceil f/2\rceil} \\
&\le \left(\frac{2em}{r^2 \max\left\{\lceil m/r^{3/2}\rceil,\, \Delta+1\right\}}\right)^{\lceil f/2\rceil} \\
&\le \left(\frac{2em}{r^2(m/r^{3/2})}\right)^{f/2} \\
&\le \left(\frac{2e}{\sqrt{r}}\right)^{f/2}.
\end{aligned}$$

We proceed by computing the probability that f active senders remain after $2c$ steps. It is easy to verify that the probability that f senders remain active after $2c$ steps in our algorithm is less than the probability that f senders remain active if each of the $2c$ successive steps is implemented by sending from all of the processors that were active at the initial step. In this situation, the successive steps are independent; thus the probability that there are f senders that never got a message through on any of the steps is at most the probability above raised to the $2c$th power. This proves Claim 1.

Now we define $r_0 = 36$, $r_j = r_{j-1}^{3/2}$, $f_j = \max\left\{\lceil m/r_j^{3/2}\rceil, \Delta + 1\right\}$, and $t = \min\{j : f_j = \Delta + 1\}$. The algorithm will run for $t + 1$ "supersteps" $0, 1, \ldots, t$, each superstep consisting of $2c$ steps as described above, with c a constant to be chosen later. Observe that the number of supersteps, and hence the total number of steps, is $\mathrm{O}(\log\log s)$ and is therefore $\mathrm{O}(\log\log p)$.

We say that superstep j is *successful* if, starting with at most m/r_j active senders, it finishes with (strictly) fewer than f_j active senders. Note that if supersteps $0, 1, \ldots, j$ are all successful, then the number of active senders remaining at the end of superstep j is strictly less than f_j. If all $t+1$ supersteps are successful, then the number of active senders remaining at the end is at most Δ, as required.

Using Claim 1, we can bound the probability that some superstep fails by

$$\sum_{j=0}^{t} \Big(\frac{2\mathrm{e}}{\sqrt{r_j}}\Big)^{cf_j}.$$

Notice that each term where $r_j \le m^{1/3}$ is at most $(\mathrm{e}/3)^{6c\sqrt{s}}$, and every other term is at most $(16\mathrm{e}^6/(9s))^{c(\Delta+1)/6}$. Thus the probability that *some* superstep fails is at most

$$(t+1)\Big\{\Big(\frac{\mathrm{e}}{3}\Big)^{6c\sqrt{s}} + \Big(\frac{16\mathrm{e}^6}{9s}\Big)^{c(\Delta+1)/6}\Big\}.$$

Observe that $t+1 = \mathrm{O}(\log\log s)$, so if c is chosen to be big enough relative to α, this is at most $\alpha^{-\sqrt{s}} + s^{-\alpha(\Delta+1)}$, as required. □

We proceed by describing the implementation of the various steps of the algorithm.

2.1. Thinning. The thinning procedure is a direct OCPC algorithm which is based on Anderson and Miller's algorithm [AM88]. It consists of $\log h + 1$ phases. Intuitively, the goal of the ith phase is to reduce the problem of realizing an $h/2^{i-1}$-relation to the problem of realizing an $h/2^i$-relation. That is, the ith phase should get so many of the messages delivered that the remaining communication problem is "essentially" an $h/2^i$-relation. After the last phase, the h-relation will be mostly realized, except that there will be a small number (at most $k/(h\lceil c_2 \log\log p\rceil)$) of undelivered messages destined for each target group. (Recall that k denotes $\lceil \log^{c_1} p\rceil$.)

Let c_3 be a sufficiently large constant (depending on c_1 and c_2 and the constant α in the desired failure probability) and let t_i denote $c_3\lceil h/2^{i-1} + \log h + \log\log\log p\rceil$. ($t_i$ denotes the number of communication steps in phase i.) Before phase i, it will be the case that each participating sender has at most $h/2^{i-1}$ undelivered messages to send. During phase i, each participating sender executes the following communication step t_i times:

Choose an integer j uniformly at random from the set $\{1, \ldots, h/2^{i-1}\}$.
If there are at least j undelivered messages to be sent,
send the jth undelivered message to its destination.

After each communication step, there is an acknowledgment step in which every receiver that receives a message sends an acknowledgment back to the sender indicating that the message was delivered successfully. At the end of phase i, any sender that has more than $h/2^i$ undelivered messages left to send stops participating.

We will prove the following theorem.

THEOREM 2. *Suppose that $h \le \log p$. Then with probability at least $1 - p^{-\alpha}$, the number of undelivered messages destined for any given target group is at most $k/(h\lceil c_2 \log\log p\rceil)$ after the thinning procedure terminates.*

In order to prove Theorem 2, we will use the following notation. We will say that a given message is "participating" at any point in time if it is undelivered at that time and its sender is participating. We will say that a receiver is "overloaded" in phase i if at the start of phase i the number of participating messages with that destination

is more than $h/2^{i-1}$. We will say that the receiver becomes overloaded in phase i if it is not overloaded in phases 1 through i but it is overloaded in phase $i+1$. We will say that a sender is "good" in phase i if it does not have a message to send to an overloaded receiver. For every target group T, let $S(T)$ denote the set containing all senders in the h-relation with messages destined for T and let $N(T)$ denote the set containing all destinations of messages from processors in $S(T)$. Finally, let $S(N(T))$ be the set containing all senders with messages destined for members of $N(T)$. (Note that $|S(T)| \leq h\,|T|$, $|N(T)| \leq h^2|T|$, and $|S(N(T))| \leq h^3|T|$.) The theorem follows from the following lemma.

LEMMA 3. *Suppose that $h \leq \log p$. Let i be an arbitrary phase of the thinning procedure and let T be any target group. With probability at least $1 - p^{-}(\alpha+1)$,*

1. *at most $|N(T)|/(h^6\lceil c_2 \log\log p\rceil)$ receivers in $N(T)$ become overloaded in phase i;*
2. *at most $|S(T)|/(h^6\lceil c_2 \log\log p\rceil)$ good senders in $S(T)$ stop participating at the end of phase i.*

Proof of Theorem 2. To see that the theorem follows from Lemma 3, note that the number of target groups is at most p/k and the number of phases is $O(\log h)$, so with probability at least $1 - p^{-\alpha}$, conditions 1 and 2 hold for all phases i and target groups T. Suppose that this is the case and consider any particular target group T. A message that is destined for T will be delivered by the thinning procedure unless either (1) there is a phase in which its sender is not good (in which case the sender could possibly stop participating) or (2) its sender stops participating even though it is good. The number of messages that are destined for T and are not delivered is therefore at most

$$\log(h) \times (h^2|N(T)|/(h^6\lceil c_2 \log\log p\rceil) + h|S(T)|/(h^6\lceil c_2 \log\log p\rceil)).$$

This is at most $k/(h\lceil c_2 \log\log p\rceil)$. □

The proof of Lemma 3 will use the following "independent bounded differences inequality" of McDiarmid [McD89]. (The inequality is a development of the "Azuma martingale inequality"; a similar formulation was also derived by Bollobás as [Bol88].)

THEOREM 3 (McDiarmid). *Let $x_1, \ldots, x_n$ be independent random variables, with x_i taking values in a set A_i for each i. Suppose that the (measurable) function $f : \prod A_i \to \mathbb{R}$ satisfies $|f(\overline{x}) - f(\overline{x}')| \leq c_i$ whenever the vectors $\overline{x}$ and $\overline{x}'$ differ only in the ith coordinate. Let Y be the random variable $f(x_1, \ldots, x_n)$. Then for any $t > 0$,*

$$\Pr\left(|Y - \mathrm{E}(Y)| \geq t\right) \leq 2\exp\left(-2t^2 / \textstyle\sum_{i=1}^{n} c_i^2\right). \qquad \square$$

Proof of Lemma 3. Suppose that $h \leq \log p$, let i be an arbitrary phase of the thinning procedure, and let T be any target group. Let x_a denote the sequence of integers randomly chosen by processor P_a during phase i.

We will start by proving that with probability at least $1 - p^{-(\alpha+2)}$, at most $|N(T)|/(h^6\lceil c_2 \log\log p\rceil)$ receivers in $N(T)$ become overloaded in phase i.

Let $Y = f(\{x_a \mid P_a \in S(N(T))\})$ be the number of receivers in $N(T)$ that become overloaded during phase i. Let R be any receiver in $N(T)$ that is not overloaded in phases 1 through i, and let s_j denote the number of participating messages that are destined for R at the jth communication step of phase i. (Note that these messages are not necessarily *sent* on the jth communication step.) The probability that R receives a message on this step is at least $s_j\,(2^{i-1}/h)\,(1 - 2^{i-1}/h)^{s_j - 1}$. There is a positive constant ρ such that this probability is greater than or equal to ρ for every s_j that

is greater than or equal to $h/2^i$. (Note that R cannot become overloaded in phase i if s_j is ever less than $h/2^i$.) Therefore, the probability that R becomes overloaded is at most

$$\sum_{b=0}^{h/2^i-1} \binom{t_i}{b} \rho^b (1-\rho)^{t_i-b}.$$

Furthermore, as long as c_3 is sufficiently large (i.e., t_i is sufficiently large compared to b), there is a constant $c_4 > 1$ such that the above sum is at most $c_4^{-t_i}$. Therefore, the expected number of processors in $N(T)$ that become overloaded in phase i is at most $|N(T)|\, c_4^{-t_i}$, which is at most $|N(T)|/(2h^6\lceil c_2 \log\log p\rceil)$ as long as c_3 is sufficiently large.

If the value of x_a changes for any a, then Y changes by at most h. Therefore, by the bounded differences inequality of Theorem 3, the probability that Y is greater than $|N(T)|/(h^6\lceil c_2 \log\log p\rceil)$ is at most

$$2\exp(-2\,|N(T)|^2/(4\,h^{12}\,\lceil c_2 \log\log p\rceil^2\,|S(N(T))|\,h^2)).$$

This is at most $p^{-(\alpha+2)}$ as long as the constant c_1 is sufficiently large (i.e., the target groups are sufficiently large). (Here we use the fact that $h \le \log p$.)

We now prove that with probability at least $1 - p^{-(\alpha+2)}$, at most

$$|S(T)|/(h^6\lceil c_2 \log\log p\rceil)$$

good senders in $S(T)$ stop participating at the end of phase i.

Let $Y = f(\{x_a \mid P_a \in S(N(T))\})$ be the number of good senders in $S(T)$ that stop participating at the end of phase i.

Let S be any good sender in $S(T)$ that participates in phase i, and let s_j denote the number of participating messages that S has to send at the jth communication step of phase i. Let $d_{\ell,j}$ denote the number of participating messages at the jth communication step that have the same destination as the ℓth message that S has to send. (Since S is good, each $d_{\ell,j}$ is less than or equal to $h/2^{i-1}$.) The probability that S sends a message successfully on the jth communication step is at least $\sum_{\ell=1}^{s_j}(2^{i-1}/h)\,(1-2^{i-1}/h)^{d_{\ell,j}-1}$. As before, there is a positive constant ρ such that this probability is greater than or equal to ρ for every s_j that is greater than or equal to $h/2^i$. Therefore, the probability that S stops participating is at most

$$\sum_{b=0}^{h/2^i-1} \binom{t_i}{b} \rho^b (1-\rho)^{t_i-b}.$$

As in the proof of the first part of the lemma, we conclude that the expected number of good senders in $S(T)$ that stop participating at the end of phase i is at most $|S(T)|/(2h^6\lceil c_2 \log\log p\rceil)$.

If the value of x_a changes for any a, then Y changes by at most h^2. Therefore, by the bounded differences inequality of Theorem 3, the probability that Y is greater than $|S(T)|/(h^6\lceil c_2 \log\log p\rceil)$ is at most

$$2\exp(-2\,|S(T)|^2/(4\,h^{12}\,\lceil c_2 \log\log p\rceil^2\,|S(N(T))|\,h^4)).$$

This is at most $p^{-(\alpha+2)}$ as long as the constant c_1 is sufficiently large (i.e., the target groups are sufficiently large). (Once again, we use the fact that $h \le \log p$.) □

2.2. Spreading. Let α be any positive constant and let c_2 be the constant associated with α that is defined in Lemma 1. At the end of the thinning procedure, there will be at most $p/(h\lceil c_2 \log\log p\rceil)$ undelivered messages. We wish to spread these out so that each sender has at most one to send. To do this, we observe that there are at most $p/(h\lceil c_2 \log\log p\rceil)$ senders with undelivered messages. Suppose (without loss of generality) that h divides p and partition the set of p receivers into h disjoint sets $R_1, \ldots, R_h$ of size p/h. Perform a $(p/(h\lceil c_2 \log\log p\rceil), \lceil c_2 \log\log p\rceil, 0)$ approximate compaction to send the first message from each sender to a unique processor in R_1. (The probability that this will succeed is at least

$$1 - \alpha^{-\sqrt{p/(h\lceil c_2 \log\log p\rceil)}} - (p/(h\lceil c_2 \log\log p\rceil))^{-\alpha}.)$$

Finally, send the remaining messages to $R_2, \ldots, R_h$ in $\Theta(h)$ communication steps with no contention using the following strategy. If the first message of sender i was sent to the jth cell of R_1 by the approximate compaction, then send the ℓth message of sender i to the jth cell of R_ℓ for $1 < \ell \leq h$.

2.3. Deliver to target groups. Let α be any positive constant and let c_2 be the constant associated with α that is defined in Lemma 1. At the end of the spreading procedure, each sender will have at most one undelivered message to send and each target group will have at most $k/(h\lceil c_2 \log\log p\rceil)$ undelivered messages to receive. (Recall that $k = \lceil \log^{c_1} p\rceil$.) Our goal is to deliver the messages to the target groups. After this procedure terminates, each processor will have at most two undelivered messages to send and the destination of each undelivered message will be within the target group containing its sender.

We have two methods for implementing this procedure in $\Theta(\log\log p)$ communication steps. The simpler method (which we describe here) involves making copies of messages but the other method does not. The simpler of the two methods consists of two phases.

We first describe phase 1. Consider any target group T. At the start of the procedure, there are at most $k/\lceil c_2 \log\log p\rceil$ senders, each of which has one message to send to the target group. Let ℓ denote $\lfloor \log p \rfloor$. We send all but up to k/ℓ^2 of these messages to T in $O(\log\log p)$ steps by doing a $(k/\lceil c_2 \log\log p\rceil, \lceil c_2 \log\log p\rceil, k/\ell^2)$ approximate compaction. We can do this in parallel for each target group, and the probability that it fails for any target group is at most

$$\tfrac{p}{k}(\alpha^{-\sqrt{k/\lceil c_2 \log\log p\rceil}} + (k/\lceil c_2 \log\log p\rceil)^{-\alpha(k/\ell^2+1)}),$$

which is sufficiently small as long as the constant c_1 in the definition of k is sufficiently large.

We will use the phrase "completely undelivered" to describe all messages that were undelivered before phase 1 and were not delivered to their target groups during phase 1. At the end of phase 1, each sender has at most one completely undelivered message to send, each member of each target group has received at most one message, and the number of completely undelivered messages is at most p/ℓ^2. Choose ℓ disjoint sets $R_1, \ldots, R_\ell$ of size $\lfloor p/\ell \rfloor$ from the set of p receivers and let Q_j denote the set consisting of the jth receiver from each of $R_1, \ldots, R_\ell$. Next, send all of the completely undelivered messages to R_1 by performing a $(p/\ell^2, \lceil c_2 \log\log p\rceil, 0)$ approximate compaction. (This fails with probability at most $\alpha^{-\sqrt{p/\ell^2}} + (p/\ell^2)^{-\alpha}$.) Finally (for each j in parallel), the processors in Q_j copy the message received at the jth receiver in R_1

(if there is one) to the other processors in Q_j. (This takes $\Theta(\log\log p)$ communication steps.)

At this point, each completely undelivered message is stored at each of the ℓ processors in Q_j (for some j) and each processor stores at most one completely undelivered message. The following communication step is now performed in parallel by all processors. If the ith processor in Q_j has a completely undelivered message to send, then it chooses an integer uniformly at random from the set $\{\gamma \mid (1 \le \gamma \le k)$ and $(\gamma \bmod \ell = i)\}$ and it sends the message to the γth processor in its target group. The probability that the ith processor in Q_j is unsuccessful is at most $1/\ell$, and this probability is independent of the probability that the other processors in Q_j succeed, so the probability that there is a completely undelivered message that is not delivered at least once to its target group in this communication step is at most $p\ell^{-\ell}$, which is sufficiently small.

For each j, in parallel the processors in Q_j perform parallel prefix to select one of the delivered copies. They then send messages "cancelling" any other copies that were delivered to their target group. This takes $\Theta(\log\log p)$ communication steps. Note that each processor receives at most two messages during the procedure — one in phase 1 and one in phase 2.

2.4. Deliver within target groups. When this procedure begins, each sender has at most two undelivered messages to send and the destination of each undelivered message is within the target group containing its sender. Our goal is to deliver all of the undelivered messages.

This procedure can be implemented deterministically in $\Theta(h + \log\log p)$ steps by running the second phase of Valiant's algorithm [Val90] twice within each target group. The algorithm within each target group is as follows. First, we consider only one undelivered message per sender. These messages are sorted by destination in $\Theta(\log\log p)$ communication steps using an EREW sorting algorithm such as Cole's parallel merge sort [Col88].[4] Then the sorted messages are delivered to their destinations without contention in $\Theta(h)$ communication steps. Next, the process is repeated for the remaining undelivered messages.

In this section, we describe an alternative implementation of the procedure. It does not rely on efficient deterministic $O(\log p)$-time EREW sorting and it is therefore likely to be faster in practice.

The main idea is as follows. We start by subdividing each target group into target subgroups. We then run the "thinning," "spreading," and "deliver to target group" procedures within each target group to deliver the messages to their target subgroups. If these three procedures succeed within a target group, then each sender in the group will have at most two undelivered messages to send and the destination of each undelivered message will be within the target *subgroup* of its sender. We can now run the second phase of Valiant's algorithm twice within each target subgroup to deliver the messages in the target group to their final destinations. Since the subgroups are very small, we can use Bitonic sort (which is fast in practice) to do the sorting. With high probability, the proportion of target groups for which the "thinning," "spreading," or "deliver to target group" procedures fail will be $O(k^{-3})$. We now allocate a group of k^2 extra processors to each of these target groups, and

[4] Valiant uses a randomized parallel sorting algorithm instead of using parallel merge sort. We cannot do that here because we want to be able to claim that (with high probability) the messages are successfully (and quickly) sorted in all of our target groups.

we use these extra processors to sort the messages using a counting sort that is fast in practice as well as in theory.

We now describe the procedure in more detail. The communication problem within each target group can be viewed as the problem of realizing an h-relation on a k-OCPC. Therefore, we can run the "thinning," "spreading," and "deliver to target group" procedures simultaneously *within* each target group. Before we can do that, we must partition each target group into target subgroups. Let the size of the target subgroups be $k' = \lceil \log^{c_5} k \rceil$, where c_5 is a constant that is sufficiently large that the probability that the "thinning," "spreading," and "deliver to target group" procedures fail within a target group is at most k^{-3}. (In order to simplify the presentation, in this section, we will assume that k' divides k. The case in which k' does not divide k is no more difficult—it is simply messier. Similarly, we will assume that k^3 divides p.) After the "deliver to target group" procedure terminates within each target group, run the second phase of Valiant's algorithm twice within each target subgroup, using Bitonic sort to do the sorting. (This takes $\Theta(h + \log^2 k')$ communication steps.) If the "thinning," "spreading," and "deliver to target group" procedures succeeded within a target group then all of its messages are now delivered. (This will happen with probability at least $1 - k^{-3}$.)

We now describe the second part of the procedure — the allocation of extra processors to help target groups that have not finished. Partition the set of target groups into p/k^2 disjoint sets $S_1, \ldots, S_{p/k^2}$. Each set S_ℓ contains k target groups and is called a *target supergroup*. Partition the set of target supergroups into k disjoint sets $\mathcal{C}_1, \ldots, \mathcal{C}_k$. Each set $\mathcal{C}_\ell$ contains p/k^3 target supergroups (and therefore p/k^2 target groups) and is called a *collection* of target supergroups. Note that with probability at least $1 - k \exp(-p/3k^5)$, each collection of target supergroups contains at most $2p/k^5$ unfinished target groups. Suppose that this is the case. Each target group and each target supergroup performs a parallel prefix to determine whether or not it has finished. (This takes $\Theta(\log \log p)$ communication steps.) Next, each processor that is part of an unfinished target group attempts to find a finished target supergroup. In particular, if the processor is the jth member of the target group, then it chooses a target supergroup uniformly at random from $\mathcal{C}_j$ and it sends a message to the first processor in the target supergroup asking whether the target supergroup is finished. The probability that a given member of a given unfinished target group fails to find a finished supergroup is at most $3/k$. (The probability that the supergroup chosen is not finished is at most $2/k^2$ and the probability that the query is sent to the same destination as some other query is at most $2/k$.) Furthermore, the queries from any given target group are independent of each other, so the probability that every processor in a given unfinished target group fails to find a finished supergroup is at most $(3/k)^k$ and the probability that there exists an unfinished target group that fails to find a finished supergroup is at most $p(3/k)^k$, which is sufficiently small. Each unfinished target group then performs a parallel prefix to choose a single finished supergroup.

At this point, each unfinished target group has identified a single finished supergroup containing k^2 processors. Consider the k^2 processors to be organized in a k by k matrix. We now run Valiant's algorithm twice in each unfinished target group. The messages are sorted using Muller and Preparata's algorithm [MP75], which works as follows. The ith processor of the unfinished target group sends its message (if it has one) to all of the processors in the ith row. (This takes $\Theta(\log \log p)$ communication steps.) If the processor in the ith row of the ith column gets a message, then it sends this message to all of the processors in the ith column and the processors in the ith

column perform parallel prefix to determine its rank. (Again, this takes $\Theta(\log\log p)$ communication steps.) Finally (in one communication step), the message with rank i is sent to the ith processor in the unfinished target group.

3. A lower bound for direct OCPC algorithms. The algorithm described in the previous section often sends a message to a processor other than its final destination, i.e., the algorithm is not *direct*. Using a nondirect strategy in a network that allows direct routing may seem strange at first, and one might question its necessity. In this section, we prove a lower bound that demonstrates that any sublogarithmic OCPC algorithm must necessarily use nondirect routing.

THEOREM 4. *Let $\mathcal{A}$ be any direct (randomized) OCPC algorithm that can realize any 2-relation with success probability at least $\frac{1}{2}$. Then there is a 2-relation which $\mathcal{A}$ takes $\Omega(\log p)$ communication steps to realize.*

Proof. Consider any direct randomized OCPC algorithm that runs for $t \leq \lfloor \frac{1}{3}\log p \rfloor$ steps. We shall construct a 2-relation ρ such that the probability that the algorithm successfully realizes ρ is exponentially small (in p). In the 2-relation ρ, each processor has at most one message to send.

Consider a processor P_i that is not itself the destination of any messages and has a single message to send to P_d but is blocked every time it attempts to transmit. Since P_i receives no external stimulus, we can imagine that P_i selects its transmission strategy at random in advance of the first time step. A strategy for P_i to transmit to P_d (under the blocking regime) can be coded as a binary word of length t, where a 1 in position t' indicates that P_i is to attempt to send its message at time step t'.

For convenience, assume that p is divisible by 4. The 2-relation ρ is the union of $p/4$ subrelations, each consisting of a pair of sending processors attempting to send a single message each to a common destination. The $3p/4$ processors in the 2-relation are distinct. The $p/4$ subrelations will be selected sequentially. Note that at any stage, there will be $f \geq p/4$ "free" processors from which the next pair of senders may be selected. To make the selection, first choose a free destination processor P_d. Observe that since the number of possible transmission strategies is 2^t, there must exist a strategy $\sigma \in \{0,1\}^t$ such that the expected number of free senders that choose strategy σ to send to P_d under the blocking regime is at least $f2^{-t}$. Thus there is a free sender, say P_i, that chooses strategy σ with probability at least $2^{-t} \geq p^{-1/3}$ and a different free sender, say P_j, that chooses σ with probability at least

$$(f2^{-t}-1)/(f-1) \geq 2^{-t} - f^{-1} \geq p^{-1/3} - 4p^{-1},$$

which is at least $\frac{1}{2}p^{-1/3}$ for $p \geq 24$. Now add to ρ the subrelation that requires P_i and P_j each to send a single message to P_d.

Note that P_i and P_j select strategies independently, so the probability that they both select σ is at least $\frac{1}{2}p^{-2/3}$; thus the probability that P_i and P_j fail to get rid of their messages is also at least $\frac{1}{2}p^{-2/3}$. Since there are $p/4$ subrelations forming ρ, the probability that ρ is successfully realized is at most $(1-\frac{1}{2}p^{-2/3})^{p/4}$, which is less than $\exp(-p^{1/3}/8)$. □

It may be observed from the proof that a direct algorithm requires a logarithmic number of steps to achieve even inverse polynomial success probability.

4. The MOB-PC algorithm. In this section, we describe a $\theta(h+\log\log p)$-communication-step algorithm that realizes an arbitrary h-relation on a $p \times p$ MOB-PC. We show that if $h \leq \log p$, then the failure probability can be made as small as $p^{-\alpha}$ for any positive constant α.

Since each row and each column of a $p \times p$ MOB-PC is itself a p-OCPC, we start by considering a p-OCPC. As in section 2, we divide the p processors into target groups of size $k = \lceil \log^{c_1} p \rceil$. A *target-group h-relation* is defined to be a communication problem in which each processor has up to h messages that it wishes to send. The destinations of these messages are target groups, and each target group is the destination of at most hk messages. We will use the following lemma.

LEMMA 4. *Suppose that $h \leq \log p$ and let α be any positive constant. Then there is a p-OCPC algorithm that can realize an arbitrary target-group h-relation in $O(h + \log\log p)$ steps with failure probability $3p^{-\alpha}$.*

Proof. Suppose that we are given a target-group h-relation. As in section 2, we will let $S(T)$ denote the set containing all senders that have messages destined for target group T. Let $M(S(T))$ denote the set of messages that are to be sent by these senders. Let each message choose a destination uniformly at random from within its target group. Let $h' = 8eh + \log\log p$ and let $h'' = h'/2$. We will say that a message is *externally bad* with respect to a target group T if the message has the same destination as at least h'' other messages that are not sent from senders in $S(T)$. We will say that a message is *internally bad* with respect to a target group T if it has the same destination as at least h'' other messages that are sent from senders in $S(T)$. We will say that a sender is *initially good* unless one or more of its messages is (externally or internally) bad. We will prove the following claim.

CLAIM 2. *With probability at least $1 - p^{-\alpha}$, every set $S(T)$ contains at most $k/(2h'^2\lceil c_2 \log\log p \rceil)$ senders that are not initially good.*

Suppose that every set $S(T)$ contains at most $k/(2h'^2\lceil c_2 \log\log p \rceil)$ senders that are not initially good and that we start to deliver messages to their destinations by running the thinning procedure from section 2.1 using h' as the value of the variable "h". It is easy to see that we can modify the proof of Lemma 3 to obtain the following. (The following lemma is the same as Lemma 3 except for the factor of 2 in the denominator.)

LEMMA 3′. *Suppose that $h' \leq \log p$. Let i be an arbitrary phase of the thinning procedure and let T be any target group. With probability at least $1 - p^{-(\alpha+1)}$,*

1. *at most $|N(T)|/(2\,h'^6\lceil c_2 \log\log p \rceil)$ receivers in $N(T)$ become overloaded in phase i;*
2. *at most $|S(T)|/(2\,h'^6\lceil c_2 \log\log p \rceil)$ good senders in $S(T)$ stop participating at the end of phase i.*

We conclude that with probability at least $1 - 2p^{-\alpha}$, the number of messages that are not delivered to a given target group is at most the sum of

1. $k/(2h'\lceil c_2 \log\log p \rceil)$ (these messages may not be delivered because their sender is not initially good);
2. $k/(2h'\lceil c_2 \log\log p \rceil)$ (these messages may not be delivered because their sender stops participating or stops being good during the thinning).

We conclude that with probability at least $1 - 2p^{-\alpha}$, the number of undelivered messages destined for any given target group is at most $k/(h'\lceil c_2 \log\log p \rceil)$ after the thinning procedure terminates. Therefore, we can deliver the rest of the messages to their target groups using the "spreading" procedure from section 2.2 and the "deliver to target groups" procedure from section 2.3. In the remainder of this section, it will be important to have our algorithm for realizing target-group h-relations behave symmetrically with respect to the different destinations within a target group. We can achieve this goal by modifying the "deliver to target group" procedure from section 2.3 as follows.

1. In the first part of the procedure, we deliver messages to their target groups using "approximate collection" rather than "approximate compaction."

2. In the second part of the procedure (the part involving copies), the "winner" is chosen uniformly at random (rather than arbitrarily) from amongst the successfully delivered copies.

We now finish the proof of Lemma 4 by proving Claim 2. Let T be any target group. We will show that the probability that $M(S(T))$ contains more than $k/(4h'^2\lceil c_2 \log\log p\rceil)$ externally bad messages is at most $\frac{1}{2}p^{-\alpha}\,(k/p)$. Then we will show that the probability that $M(S(T))$ contains more than $k/(4h'^2\lceil c_2 \log\log p\rceil)$ internally bad messages is at most $\frac{1}{2}p^{-\alpha}\,(k/p)$.

First, we consider externally bad messages. We will say that a processor P is *externally crowded* with respect to a target group T if there are at least h'' messages which are not in $M(S(T))$ and have destination P. A set of b members of a target group are all externally crowded only if at least bh'' messages have destinations in the set. Therefore, the probability that there is a set of b members of a target group that are all externally crowded is at most

$$\binom{p}{k}\binom{k}{b}\binom{kh}{bh''}\left(\frac{b}{k}\right)^{bh''}.$$

We can use Stirling's approximation to show that for $b = k/h''^6$ this quantity is at most $(p/k)2^{-k/h''^5}$. Therefore, with probability at least $1-(p/k)2^{-k/h''^5}$, every target group has at most k/h''^6 processors which are externally crowded with respect to T. Suppose that this is the case. Then the probability that a message in $M(S(T))$ chooses a destination which is externally crowded with respect to T is at most h''^{-6}. Using a Chernoff bound, we see that with probability at least $1-\exp(-|M(S(T))|\,/\,(3\times h''^6))$, at most $2\,|M(S(T))|/h''^6$ messages in $M(S(T))$ choose a destination which is externally crowded with respect to T. Note that as long as p is sufficiently large, $2\,|M(S(T))|/h''^6 \le k/(4h'^2\lceil c_2\log\log p\rceil)$. Also, as long as $|M(S(T))| \ge k/(4h'^2\lceil c_2\log\log p\rceil)$, $h' \le \log p$, and the constant c_1 is sufficiently large, the sum of $(p/k)2^{-k/h''^5}$ and $\exp(-|M(S(T))|\,/\,(3\times h''^6))$ is at most $\frac{1}{2}p^{-\alpha}\,(k/p)$.

We now consider internally bad messages. We start by calculating an upper bound on the probability that a message is internally bad. This probability is at most

$$\sum_{j=h''}^{hk-1}\binom{hk-1}{j}\frac{1}{k^j}\left(1-\frac{1}{k}\right)^{hk-1-j}.$$

We can use Stirling's approximation to show that this sum is $\mathrm{O}(2^{-h''})$. So the expected number of messages in $M(S(T))$ which are internally bad is $\mathrm{O}(|M(S(T))|2^{-h''})$.

Let x_i be a random variable which denotes the destination of the ith message in $M(S(T))$ and let Y be a random variable denoting the number of internally bad messages in $M(S(T))$. (Y is a function of $x_1,\ldots,x_{|M(S(T))|}$.) If we change one of the x_i's, then we change Y by at most $h''+1$. Therefore, by Theorem 3 (the bounded differences inequality),

$$\begin{aligned}&\Pr(Y \ge k/(4h'^2\lceil c_2\log\log p\rceil))\\ &\quad\le 2\exp\left(-2\left(\frac{k}{4h'^2\lceil c_2\log\log p\rceil}-\mathrm{E}(Y)\right)^2\Big/\left(|M(S(T))|\,(h''+1)^2\right)\right).\end{aligned}$$

Since $\mathrm{E}(Y) \leq k/(8{h'}^2\lceil c_2 \log\log p\rceil)$ (for big enough p), the probability is at most

$$2\exp(-k/(32{h'}^4\lceil c_2 \log\log p\rceil^2 h^2(h''+1)^2)).$$

This quantity is at most $\frac{1}{2}p^{-\alpha}\,(k/p)$ as long as c_1 is sufficiently large and h' is at most $\log p$. □

Now that we have proved Lemma 4, we are ready to describe our algorithm for realizing h-relations on a $p \times p$ MOB-PC. We have already observed that each row and each column of a MOB-PC is a p-OCPC. We will divide each row and each column of the MOB-PC into target groups of size k. A *block* of the MOB-PC is defined to be a $k \times k$ sub-MOB-PC in which each row is a row target group of the original MOB-PC and each column is a column target group of the original MOB-PC. We will use the phrase *column of blocks* to refer to a collection of p/k blocks which together make up k columns of the MOB. Finally, we will subdivide each column of blocks into p/k^2 *superblocks* in which each superblock consists of k blocks. (As in section 2.4, we will simplify the presentation by assuming that k^3 divides p. We will also assume that $h \leq \log p$.)

The algorithm has five steps:

1. *On each row:* Each message picks a random row target group and the messages are routed to the target groups.

2. *On each column:* Each message chooses as its immediate destination the column target group that intersects the row of its final destination. The messages are routed to the target groups.

3. *Within each block:* Each message chooses an immediate destination uniformly at random from its row. The messages are routed to their immediate destinations using the OCPC algorithm in each row. Each message that was successfully delivered to its immediate destination chooses as its new immediate destination the processor which is in its column and in the row of its final destination. The messages are routed to their immediate destinations using the OCPC algorithm in each column. If the block contains a message that was not successfully delivered to the row of its final destination, then we say that the block *failed.* Every processor in the block is notified of the failure.

4. If any block of a superblock failed, then we say that the superblock failed. Every processor in the superblock is notified of the failure. Each failed block attempts to allocate a superblock which has not failed from within its column of blocks. After allocating a superblock, the failed block copies all of its messages to each of the blocks in the superblock. Each of these blocks then repeats step 3. If there is a block in the superblock which does not fail, then the first such block copies the (delivered) messages back to the original failed block.

5. *On each row:* Each message is routed to its final destination.

We will conclude the section by considering each of the five steps. For each step, we will discuss the method that is used to implement the step and also the failure probability of the method.

At the beginning of step 1, each processor has at most h messages. Each message then picks a random row target group. Using a Chernoff bound, we see that the probability that a given target group is the destination of more than $2hk$ messages is at most $e^{-hk/3}$, so the probability that there is such a target group is at most $p \times (p/k) \times e^{-hk/3}$, which is at most $p^{-\alpha}$ as long as c_1 is sufficiently large. Suppose that every target group is the destination of at most $2hk$ messages. Then we can use the method described in the proof of Lemma 4 to deliver the messages to their target

groups in $O(h + \log\log p)$ steps. The probability that this method fails is at most $3p^{-\alpha}$ for any positive constant α.

At the beginning of step 2, each processor has at most h_2 messages, where h_2 is h plus the number of time steps used in step 1. If it is also true that every target group is the destination of $O(hk)$ messages in step 2, then we can use the method described in the proof of Lemma 4 to deliver the messages to the target groups in $O(h + \log\log p)$ steps. We will conclude our discussion of step 2 by showing that with high probability each target group is the destination of $O(hk)$ messages.

Let T be any column target group and let C be the column of T. There are at most hkp messages which have final destinations in rows which intersect T. These are the only messages which could be destined for T in step 2. We will refer to them as the set of "potentially relevant" messages. Each potentially relevant message will be destined for T in step 2 if and only if it is delivered to column C in step 1. Therefore, our goal is to prove that with high probability only $O(hk)$ of the potentially relevant messages are delivered to column C in step 1.

We start out by using a Chernoff bound to prove that with probability at least $1 - \exp(-hk^2/3)$, only $2hk^2$ of the potentially relevant messages select target groups that intersect C in step 1. We refer to these messages as "relevant" messages. Our goal is to prove that with high probability only $O(hk)$ of the relevant messages are delivered to column C in step 1.

We will use the following theorem of Hoeffding, which is included in McDiarmid's paper [McD89].

THEOREM 5 (Hoeffding). *Let the random variables $X_1, \ldots, X_p$ be independent, with $0 \le X_i \le 1$ for each i. Let $\overline{X} = \frac{1}{p}\sum_i X_i$ and $\mu = \mathrm{E}[\overline{X}]$. Then for $0 \le t < 1 - \mu$,*

$$\Pr(\overline{X} \ge t + \mu) \le \left[\left[\frac{\mu}{\mu + t}\right]^{\mu+t}\left[\frac{1-\mu}{1-\mu-t}\right]^{1-\mu-t}\right]^{p}. \qquad \square$$

To apply Hoeffding's inequality, let X_i be h_2^{-1} times the number of relevant messages that are delivered to row i of column C in step 1. Observe that $0 \le X_i \le 1$ and that the X_i's are independent. Note that $\overline{X}$ is $(h_2 p)^{-1}$ times the number of relevant messages that are delivered to column C in step 1. Recall that the algorithm for realizing target-group h-relations behaves symmetrically with respect to the destinations forming a particular target group; thus the expected number of relevant messages delivered to column C in step 1 is k^{-1} times the expected number of relevant messages. Therefore, μ is at most $2hk/(h_2 p)$. Let t denote $4hk/(h_2 p)$. Observe that $t \ge 2\mu$ and that $0 \le t < 1 - \mu$. By Hoeffding's inequality, the probability that $\overline{X}$ is at least $6hk/h_2 p$ is at most

$$\left[\left[\frac{\mu}{\mu + t}\right]^{\mu+t}\left[\frac{1-\mu}{1-\mu-t}\right]^{1-\mu-t}\right]^{p} \le 3^{-tp}e^{tp} = e^{-\Omega(tp)}.$$

We conclude that with high probability at most $6hk$ messages are destined for any target group during step 2. In this case, the messages can be delivered in $O(h + \log\log p)$ steps using the method described in the proof of Lemma 4.

At the beginning of step 3, each processor has at most h_3 messages, where h_3 is h plus the number of time steps used in steps 1 and 2. Using a Chernoff bound (as in step 1), we see that with probability at least $1 - p \times (p/k) \times e^{-hk/3}$, each row of

each block is the destination of at most $2hk$ messages in step 3. We now consider each particular block. Following Rao [Rao92], we can use a Chernoff bound to show that with probability at least $1 - k^2 \exp(-h_3/3)$, the communication problem on each row is a $2h_3$ relation. Similarly, with high probability the communication problem on each column is a $2h_3$-relation. Therefore, the probability of failure can be made as small as k^{-3}. The processors in the block use parallel prefix to notify each other of failure. Similarly, the processors in each superblock use parallel prefix to notify each other of failure.

The implementation and analysis of step 4 closely follows that of section 2.4. The probability that there is a failed block that fails to allocate a superblock is at most $(p^2/k^2)(3/k)^k$. The probability that there is a superblock in which every block fails when it repeats step 3 is at most $(p^2/k^3)(1/k^3)^k$.

If steps 1–4 are successful, then at the start of step 5, all of the messages will be in the correct row. Furthermore, there will be at most h_5 messages at any processor, where h_5 is h plus the number of time steps used in steps 1–4. Since the communication problem is an h-relation, each processor will be the destination of at most h messages. Therefore the p-OCPC algorithm described in section 2 can be used to deliver the messages on each row. The probability that this algorithm fails is at most p (the number of rows) multiplied by the probability that the p-OCPC algorithm fails, which is at most $p^{-\alpha}$ for any positive constant α.

In the introduction to this paper, we pointed out that the MOB-PC is easier to build than an OCPC because it restricts the number of processors that a given processor can send to directly. Nevertheless, we have provided an algorithm for realizing h-relations on a MOB-PC which is asymptotically as fast as the fastest known algorithm for realizing h-relations on an OCPC. Similarly, we could define a new machine by replacing each row and each column of a $p \times p$ MOB-PC with a $p^{1/2} \times p^{1/2}$ MOB-PC. Our algorithm could be used recursively to realize h-relations in $O(h + \log\log p)$ steps on the new machine. Clearly, this recursion could be carried out to any constant depth.

REFERENCES

[AM88] R. J. Anderson and G. L. Miller, *Optical communication for pointer based algorithms*, Technical Report CRI 88-14, Computer Science Department, University of Southern California, Los Angeles, 1988.

[Bol88] B. Bollobás, *Martingales, isoperimetric inequalities and random graphs*, in Combinatorics, A. Hajnal, L. Lovász, and V. T. Sós, eds., Colloq. Math. Soc. János Bolyai 52, North–Holland, Amsterdam, 1988, pp. 113–139.

[CDHR89] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik, *New simulations between CRCW PRAMs*, in Proc. Foundations of Computation Theory 7, Lecture Notes in Comput. Sci. 380, Springer-Verlag, Berlin, 1989, pp. 95–104.

[Col88] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[Dow91] P. W. Dowd, *High performance interprocessor communication through optical wavelength division multiple access channels*, in Proc. 18th ACM International Symposium on Computer Architecture, ACM, New York, 1991, pp. 96–105.

[Esh88] M. M. Eshaghian, *Parallel computing with optical interconnects*, Ph.D. thesis, University of Southern California, Los Angeles, 1988.

[Esh91] M. M. Eshaghian, *Parallel algorithms for image processing on OMC*, IEEE Trans. Comput., 40 (1991), pp. 827–833.

[EK88] M. M. Eshaghian and V. K. P. Kumar, *Optical arrays for parallel processing*, in Proc. 2nd International Symposium on Parallel Processing, IEEE Press, Piscataway, NJ, 1988, pp. 58–71.

[GT92] M. Geréb-Graus and T. Tsantilas, *Efficient optical communication in parallel computers*, in Proc. 4th ACM Symposium on Parallel Algorithms and Architectures,

ACM, New York, 1992, pp. 41–48.

[GV92] A. V. Gerbessiotis and L. G. Valiant, *Direct bulk-synchronous parallel algorithms*, in Proc. 3rd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comput. Sci. 621, Springer-Verlag, Berlin, 1992, pp. 1–18.

[GM91] J. Gil and Y. Matias, *Fast hashing on a PRAM*, in Proc. 2nd ACM–SIAM Symposium on discrete algorithms, SIAM, Philadelphia, 1991, pp. 271–280.

[MV91] Y. Matias and U. Vishkin, *Converting high probability into nearly-constant time: With applications to parallel hashing*, in Proc. 23rd ACM Symposium on Theory of Computing, ACM, New York, 1991, pp. 307–316.

[McC93] W. F. McColl, *General purpose parallel computing*, in Lectures on Parallel Computation, in Proc. 1991 ALCOM Spring School on Parallel Computation, A. M. Gibbons and P. Spirikas, eds., Cambridge University Press, Cambridge, UK, 1993, pp. 337–391.

[McD89] C. McDiarmid, *On the method of bounded differences*, in Surveys in Combinatorics, London Math. Soc. Lecture Notes Ser. 141, Cambridge University Press, Cambridge, UK, 1989, pp. 148–188.

[MP75] D. E. Muller and F. P. Preparata, *Bounds to complexities of networks for sorting and for switching*, J. Assoc. Comput. Mach., 22 (1975), pp. 195–201.

[Rao92] S. B. Rao, *Properties of an interconnection architecture based on wavelength division multiplexing*, Technical Report TR-92-009-3–0054-2, NEC Research Institute, Princeton, NJ, 1992.

[Val90] L. G. Valiant, *General purpose parallel architectures*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier, New York, 1990, Chapter 18 (see especially p. 967).

[Wit81] L. D. Wittie, *Communication structures for large networks of microcomputers*, IEEE Trans. Comput., C-30 (1981), pp. 264–273.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1120–1138, August 1997

010

THE ROBOT LOCALIZATION PROBLEM*

LEONIDAS J. GUIBAS†, RAJEEV MOTWANI‡, AND PRABHAKAR RAGHAVAN§

Abstract. We consider the following problem: given a simple polygon $\mathcal{P}$ and a star-shaped polygon $\mathcal{V}$, find a point (or the set of points) in $\mathcal{P}$ from which the portion of $\mathcal{P}$ that is visible is translation-congruent to $\mathcal{V}$. The problem arises in the localization of robots equipped with a range finder and a compass—$\mathcal{P}$ is a map of a known environment, $\mathcal{V}$ is the portion visible from the robot's position, and the robot must use this information to determine its position in the map. We give a scheme that preprocesses $\mathcal{P}$ so that any subsequent query $\mathcal{V}$ is answered in optimal time $O(m+\log n+A)$, where m and n are the number of vertices in $\mathcal{V}$ and $\mathcal{P}$ and A is the number of points in $\mathcal{P}$ that are valid answers (the output size). Our technique uses $O(n^5)$ space and preprocessing in the worst case; within certain limits, we can trade off smoothly between the query time and the preprocessing time and space. In the process of solving this problem, we also devise a data structure for output-sensitive determination of the visibility polygon of a query point inside a polygon $\mathcal{P}$. We then consider a variant of the localization problem in which there is a maximum distance to which the robot can "see"—this is motivated by practical considerations, and we outline a similar solution for this case. We finally show that a single localization query $\mathcal{V}$ can be answered in time $O(mn)$ with no preprocessing.

Key words. robotics, localization, computational geometry, geometric algorithms

AMS subject classifications. 65Y25, 68U05

PII. S0097539792233257

1. Introduction.

We consider the following problem: A robot is at an unknown position in an environment for which it has a map. It "looks" about its position, and based on these observations, it must infer the place (or set of places) in the map where it could be located. This is known as the *localization problem* in robotics [8, 22].

Aside from being an interesting and fundamental geometric problem, this task has several practical applications. As described in [8], localization eliminates the need for complex position-guidance equipment to be built into factories and buildings. Unmanned spacecraft require localization for the following reason [18, 20]: A rover lands on Mars, a map of whose terrain is available to it. It looks about its position and then infers its exact position on the Martian surface. Another application comes from robots that follow a planned path through a scene: the control systems that guide such a robot along the planned path gradually accumulate errors due to mechanical drift. Thus it is desirable to use localization from time to time to verify the actual position of the robot in the map, and apply corrections as necessary to return it to the planned path [22].

We assume that the robot is in an environment such as an office or factory, with flat vertical walls and a flat floor—thus the problem we address is for polygonal workspaces in two dimensions. The subject of this paper is localization using a *range*

* Received by the editors June 24, 1992; accepted for publication (in revised form) August 23, 1995.

http://www.siam.org/journals/sicomp/26-4/23325.html

† Department of Computer Science, Stanford University, Stanford, CA 94305-2140 (guibas@cs.stanford.edu). The research of this author was supported by NSF grants CCR-9215219 and IRI − 9306544, the Stanford OTL fund, the SIMA Stanford Consortium, and the Mitsubishi Corporation.

‡ Department of Computer Science, Stanford University, Stanford, CA 94305-2140. (ranjeev@cs.stanford.edu). The research of this author was supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Development Award, an OTL grant, NSF grant CCR-9010517, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, the Schlumberger Foundation, the Shell Foundation, and the Xerox Corporation.

§ IBM Almaden Research Center/K53, 650 Harry Road, San Jose, CA 95120 (pragh@almaden.ibm.com).

finder [19], a device commonly used in real robots [8, 9, 10, 19]. A range finder is a device that emanates a beam (laser or sonic) and determines the distance to the first point of contact with any object in that direction. This is similar to the *finger probe* model [7, 21] studied in computational geometry. In practice, a robot sends out a series of beams spaced at small angular intervals about its position, measuring the distance to points at each of these angles. The discrete "points of contact" are then fitted together to obtain a *visibility polygon* $\mathcal{V}$ with m vertices (in general, the number of beam probes will be much larger than m).

The robot has a map of its environment: a polygon $\mathcal{P}$ (possibly containing holes) having n vertices. We assume that the robot has a compass: its representations of $\mathcal{P}$ and $\mathcal{V}$ have a common reference direction (say north). We wish to solve the following problem: given $\mathcal{P}$ and $\mathcal{V}$, determine all of the points $p \in \mathcal{P}$ such that the visibility polygon of p is translation-congruent to $\mathcal{V}$; see Figure 1.

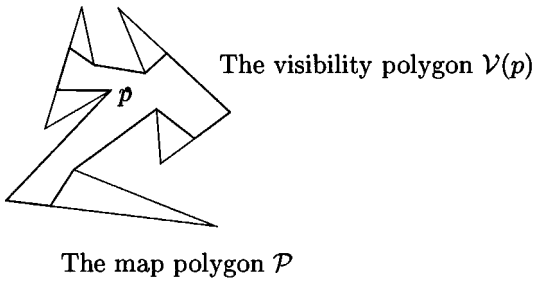


FIG. 1. *The setting for our robot localization problem.*

Because the map is likely to be fixed for a given environment, our main interest is in preprocessing the map so that subsequent queries can be answered quickly. Our main contribution is a scheme for preprocessing a simple polygon $\mathcal{P}$ so that any query $\mathcal{V}$ can be answered in time $O(m + \log n + A)$, where A is the size of the output (the number of places in $\mathcal{P}$ at which the visibility polygon would match $\mathcal{V}$); this query time is the best possible. Our preprocessing takes $O(n^5)$ time and space (section 5). We also exhibit a smooth tradeoff between the query time and the preprocessing cost, within certain limits. In these bounds, n^5 is really $n^4 r$, where r is the number of reflex vertices in $\mathcal{P}$. Better bounds hold for polygons where the number of mutually visible vertices is smaller than $O(n^2)$ and/or which do not have collinear sides.

The development of our scheme involves the study of a fundamental property of simple polygons—the *visibility cell decomposition*—that has several other applications. Sections 2, 3, and 4 study some properties of this decomposition. An interesting application of these ideas is to the construction of a data structure for the output-sensitive determination of the visibility polygon of a query point (see section 6.4). This requires a query time of $O(\log n + m)$ using $O(n^2 r)$ preprocessing and space, where m is the size of the visibility polygon. Once again, we can trade off between the query time and the space requirement.

In section 6, we consider variants of the basic problem. We first describe the effect of holes in the map polygon. Next, we consider a variant motivated by a property of some range finders—that a distance measurement is obtained only if there is a wall within a certain maximum distance D, and otherwise no reading is obtained (indicating only that the distance to the nearest wall is greater than D in that direction). We show how our approach can be modified to deal with this feature without increasing the query time, but with additional preprocessing. We

then address the following question: Given no preprocessing, what is the complexity of answering a single query? We provide an algorithm running in time $O(mn)$ by applying results on ray shooting.

Independently of our work, Bose, Lubiw, and Munro [2] obtained some of the results presented here. In particular, they provided a scheme for preprocessing a simple polygon so as to compute the vertices of the polygon that are visible from a query point in time $O(\log n + m)$ time. The machinery developed for this purpose includes some of the visibility cell decomposition structure theorems described below.

1.1. Overview of our scheme. We now give a brief overview of our scheme to motivate the study of the visibility cell decomposition in sections 2, 3, and 4. We approach the problem by partitioning the map polygon into regions such that within a region, the visibility polygon of any point is roughly the same; in section 2, we call this rough view a *skeleton*. An intuitive definition of the skeleton of $\mathcal{V}$ is that it is a contraction of $\mathcal{V}$ so that the skeleton boundary contains exactly those vertices from $\mathcal{V}$ that can be certified to be vertices of $\mathcal{P}$. We provide a data structure which quickly identifies all of the regions that have the same skeleton as the query $\mathcal{V}$. We then check the candidate regions to see if they contain any points that have exactly the same view as $\mathcal{V}$. Some difficulties that arise are the following:

(a) Due to occlusions by reflex vertices, an edge of the map polygon may have neither or only one of its endpoints visible from a point inside the polygon. Our characterization of a skeleton must cater to these incomplete edges.

(b) If the line segments forming several edges of the polygon are collinear, it is possible that a "window" in the map allows the robot to see only an interior portion of one of these edges. Further, it cannot easily identify which of these collinear edges it sees. The problem is compounded when there are several such windows and collections of collinear edges. In fact, this is one source of complexity in our preprocessing. In the case where the map $\mathcal{P}$ has no collinear edges, the preprocessing space can be improved to $O(n^3 r)$; see section 5.

(c) There can be regions that match the skeleton but contain no point whose visibility polygon is congruent to V. Thus we must still pinpoint those visibility regions (from all of the ones that share this skeleton) that contain a point whose visibility polygon exactly matches $\mathcal{V}$. We must do so in time proportional to A, so we cannot check each candidate region individually. We reduce this problem to a form of point location in a planar subdivision.

2. Visibility polygons and skeletons. Let $\mathcal{P}$ denote a polygon with n sides. We will refer to $\mathcal{P}$ as the *map polygon*. Let P denote the boundary of $\mathcal{P}$. We first assume that $\mathcal{P}$ has no holes, deferring this general case to section 6.1. Henceforth, all polygons will be assumed to be oriented with respect to a common reference direction.

Two points in P are *visible* to each other if the straight line joining them meets P only at these endpoints. The *visibility polygon* $\mathcal{V}(p)$ for any point $p \in \mathcal{P}$ is the polygon consisting of all points in $\mathcal{P}$ that are visible from p. Assume that the number of vertices in $\mathcal{V}(p)$ is m.

Let $V(p)$ denote the boundary of $\mathcal{V}(p)$. Assume that p does not lie on P, and hence it does not lie on $V(p)$. In general, there will be edges and vertices in $V(p)$ which do not coincide with edges and vertices in P. To deal with such cases, we define the notion of spurious edges and vertices. Informally, an edge or a vertex is nonspurious if the view from p provides a guarantee that this edge or vertex is on the boundary of $\mathcal{P}$.

DEFINITION 1. *An edge of* $V(p)$ *is* spurious *if it is collinear with* p.

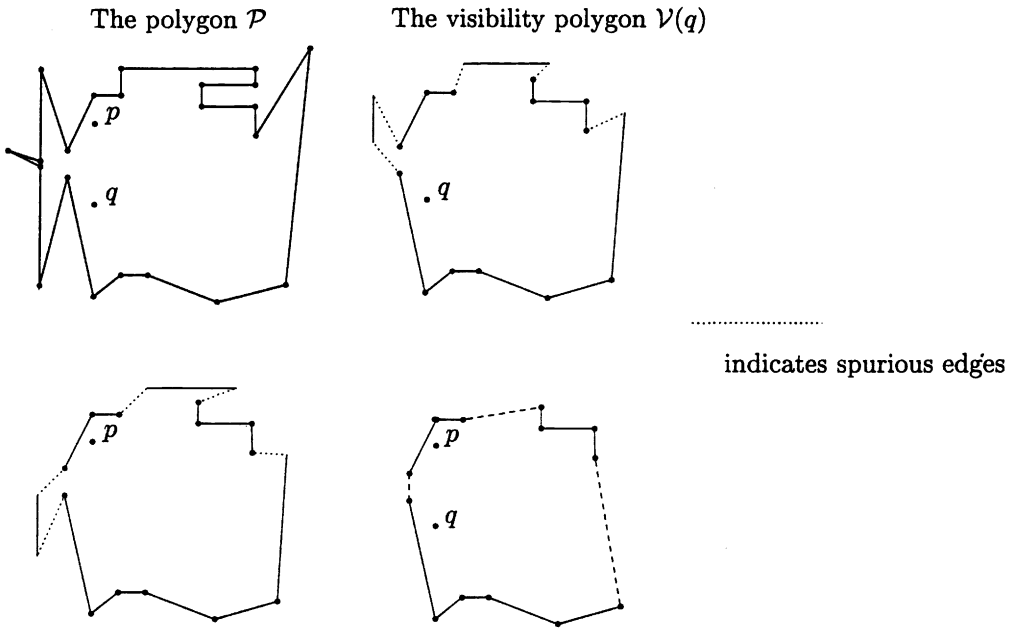


FIG. 2. *Visibility polygons and their skeletons.*

DEFINITION 2. *A vertex $v \in V(p)$ is* spurious *if it lies on a spurious edge $(u, v) \in V(p)$ and the other endpoint u is closer to p.*

Some of these notions are illustrated in Figure 2.

This definition may label as spurious an edge (or a vertex) which actually lies on P. This happens only if that edge is collinear with p. In that case, the closer of the two endpoints of the edge may be visible from p, but it will then block the view of any other point on the edge. Thus, although the edge (u, v) is in $V(p)$, the robot sitting at point p cannot infer this from its localized view. Similarly, the definition of a spurious vertex assumes that if a ray from p goes through vertices u and v of P, in that order, then u is an obstacle to the visibility of v from p. As the next lemma shows, the nonspurious components of $V(p)$ are invariant under any modifications to $\mathcal{P} \setminus \mathcal{V}(p)$. The proof is an easy consequence of the above definitions.

LEMMA 2.1. *An edge e or a vertex v in V is nonspurious if and only for each choice of $\mathcal{P}$ and $p \in \mathcal{P}$ such that $\mathcal{V}(p) = V$, e and v lie on P.*

A *reflex vertex* in P is a vertex which subtends an angle greater than 180^0 inside $\mathcal{P}$. It is the existence of reflex vertices which creates obstacles to viewing the points inside $\mathcal{P}$. Note that a spurious vertex can never be a reflex vertex in $V(p)$.

DEFINITION 3. *A reflex vertex v in $V(p)$ is a* blocking vertex *if at least one edge incident on v in P does not intersect $\mathcal{V}(p)$.*

It is now easy to establish the following lemma.

LEMMA 2.2. *If a vertex in $V(p)$ is a nonblocking reflex vertex, then both its incident edges in $V(p)$ must be nonspurious.*

The next lemma follows from the observation that each spurious edge can be extended to pass through p. Thus no two spurious edges of $V(p)$ can meet each other except at p, and by assumption p does not lie on the boundary of $\mathcal{V}(p)$.

LEMMA 2.3. *No two spurious edges can be adjacent in $V(p)$.*

For each spurious edge, the endpoint closer to p is a blocking reflex vertex and the other endpoint is a spurious vertex.

LEMMA 2.4. *Let $e \in V(p)$ be a nonspurious edge and e' be the edge of P on which it lies. Then e is the only portion of e' visible from p and the edge e is of one of the following three types:*

- full edge: *the endpoints of e are the same as those of e'*;
- half-edge: *one endpoint of e is spurious and the other is an endpoint of e'*;
- partial edge: *both endpoints of e are spurious vertices.*

We now conclude that the spurious vertices and edges in $V(p)$ can occur only in certain specific patterns. Consider a clockwise traversal of $V(p)$ starting with an arbitrary blocking (and therefore reflex) vertex. (If no such vertex exists, then $\mathcal{V}(p) = \mathcal{P}$, trivializing the whole problem, so from now on we assume the existence of such a vertex.) The sequence of vertices seen in this traversal can be decomposed into chains of consecutive nonspurious vertices alternating with chains of consecutive spurious vertices; call this the *vertex chain decomposition* of $V(p)$.

LEMMA 2.5. *The* vertex chain decomposition *of $V(p)$ has the following properties:*

1. *A nonspurious chain can contain blocking vertices only as its endpoints.*
2. *A spurious chain is of length at most* 2.
3. *Consider a spurious chain with only one vertex v. Let x be the last vertex of the preceding chain and y be the first vertex of the succeeding chain. Then one of x and y is a blocking vertex joined to v by a spurious edge, while the other is nonblocking and is joined to v by a half-edge.*
4. *Consider a spurious chain with two vertices u and v, in that order. Let x be the last vertex of the preceding chain and y be the first vertex of the succeeding chain. Then both x and y are blocking vertices with spurious edges going to u and v, respectively, and the edge (u, v) is a partial edge.*

Fix a canonical blocking reflex vertex v_o of $V(p)$ as the *origin* with reference to which we specify all other points. For example, v_o can be the leftmost blocking reflex vertex of $V(p)$. Let $V^*(p)$ be the polygon induced by the nonspurious vertices of $V(p)$ ordered by a clockwise traversal of starting at v_o. In $V^*(p)$, if two adjacent vertices are from the same chain in $V(p)$, then their edge is the same as in $V(p)$, and this must be a nonspurious edge. Otherwise, the two vertices are endpoints of neighboring chains in $V(p)$ and their edge in $V^*(p)$ is a newly introduced *artificial* edge.

Every artificial edge e' of $V^*(p)$ corresponds to some half or partial edge e of $V(p)$. The edge e is one of the edges of $V(p)$ which connect the two chains whose endpoints are the vertices of e'. We will label each artificial edge e' with a characterization of the line on which the corresponding edge e lies. This line characterization will be the coefficients of the linear equation that defines the line containing e, with the origin at v_o.

DEFINITION 4. *The* skeleton *$V^*(p)$ of a visibility polygon $\mathcal{V}(p)$ is the polygon induced by the nonspurious vertices of $V(p)$. Each artificial edge of $V^*(p)$ is labeled with the line equation and the type of the corresponding half- or partial edge.*

The skeleton of a visibility polygon can also be looked upon as a polygon induced by all of the full edges in $V(p)$ such that the chains of edges are tied together by artificial edges. It is important to keep in mind that a skeleton is a *labeled* polygon as described above.

DEFINITION 5. *The* embedding *of a skeleton $V^*(p)$ is a* 1–1 *mapping h from the vertices in the skeleton into the vertices of P such that the following hold:*

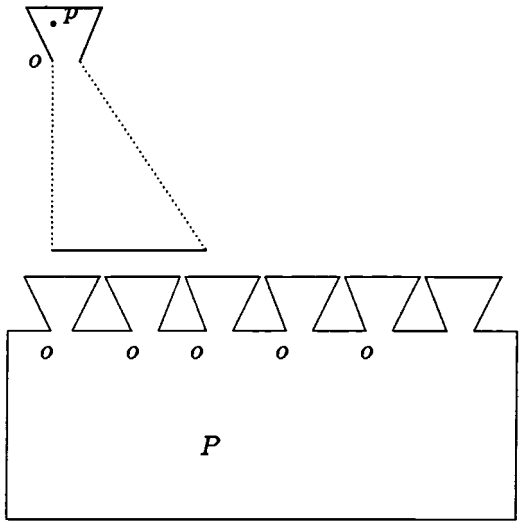


FIG. 3. *A bad case for localization.*

1. *For each vertex v in $V^*(p)$, the location of $h(v)$ relative to $h(v_o)$ is identical to the location of v relative to v_o in $V(p)$.*
2. *There is a full edge between vertices u and v in $V^*(p)$ if and only if there is an edge of P with $h(u)$ and $h(v)$ as endpoints.*
3. *Let l' be the line labeling an artificial edge between vertices u and v in $V^*(p)$. Then there is an edge e of P lying on a line l whose equation (with $h(v_o)$ as origin) is that of l', and there is a point of e visible from both u and v.*

Does $V^*(p)$ have enough information to uniquely determine the point p? Unfortunately not: the information about the endpoints of a half- or partial edge e in $V(p)$ is absent from the labels of the corresponding edge e' in $V^*(p)$. (The reason for this imprecise labeling will become clear later when we describe our search mechanism.) Thus a single embedding may have several candidate edges in P for the edge e. These candidate edges must all be collinear. For instance, in Figure 2, the partial edge on the left visible from p and that visible from q are collinear. Note that the skeletons of p and q come out to be the same.

Let r denote the number of reflex vertices in $\mathcal{P}$.

THEOREM 2.6. *Given a visibility polygon $\mathcal{V}(p)$, its skeleton $V^*(p)$ has at most r embeddings in $\mathcal{P}$ and this bound is best possible.*

Proof. Let us label any embedding h of $V^*(p)$ in $\mathcal{P}$ by the vertex $h(v_o)$, where v_o is the origin vertex in $V^*(p)$. We claim that there can be at most one embedding with the label v for any vertex v in P. This follows from the observation that the location of every vertex in $V^*(p)$ is fixed with reference to the origin vertex. Having specified the location of the origin as being at v immediately fixes the location of every other vertex and thus uniquely determines the embedding h itself.

Since v_o is a reflex vertex, there are at most r distinct labels for the embeddings of $V^*(p)$. It follows that number of distinct embeddings cannot exceed r.

To see that this bound is tight to within constant factors, consider the polygon $\mathcal{P}$ and the visibility polygon $\mathcal{V}(p)$ in Figure 3. It is clear that the origin of $V^*(p)$ can

be mapped to any of the vertices marked with o in P. Clearly, there are $\Theta(r)$ such embeddings. □

The location of the point p is fixed with reference to the origin vertex v_o of $V^*(p)$. Thus the only possible locations of the point p are the (at most r) locations corresponding to the different embeddings of the skeleton. Notice that an embedding of $V^*(p)$ may not correspond to an embedding of $V(p)$, although the converse is always true.

COROLLARY 2.7. *The number of solutions in $\mathcal{P}$ to a localization query $\mathcal{V}$ is at most r.*

3. Visibility cell decomposition. We now describe the subdivision of the map polygon into *visibility cells* such that the points in each cell have essentially the same visibility polygon. The subdivision is created by introducing lines into the interior of the map polygon. Each line partitions $\mathcal{P}$ into two regions, one where a vertex v is not visible due to the obstruction created by vertex u and another region where u cannot block the view of v. Each such line starts at a reflex vertex u and ends at the boundary of $\mathcal{P}$. It is collinear with a vertex v which is either visible from u or adjacent to it in P. This line is said to be *emanating* from v and *anchored* at u.

It is convenient to give each of the interior lines a direction and consider the interior of the map polygon to be dissected by a collection of such rays. A ray determined by vertex v as emanating vertex and vertex u as anchor vertex proceeds *from* u into P *away* from v. It forms the boundary between regions that can locally see v and others that cannot. We will classify this ray as a *left* or *right* ray for v according to whether the obstruction defining the anchor u is to the left or the right of the ray from the point of view of an observer sitting on v and looking along the ray. Note again that a ray starts at the anchor vertex and proceeds away from the emanating vertex.

THEOREM 3.1. *In the visibility cell decomposition of the map polygon $\mathcal{P}$,*

1. *the number of lines introduced in the interior of $\mathcal{P}$ is $O(nr)$, and this bound is the best possible*;
2. *each cell in the decomposition is a convex polygonal region inside $\mathcal{P}$.*

Proof. It easy to see that there are at most $O(nr)$ interior lines in the decomposition since each interior line is generated by a pair of anchor and emanating vertices, and each pair of vertices generates at most two interior lines. It is also quite easy to construct examples where the upper bound is achieved.

For the second part, let $\mathcal{C}$ be a visibility cell which is nonconvex and let w be a reflex vertex on the boundary of $\mathcal{C}$. We first claim that w cannot be a vertex from the boundary of $\mathcal{P}$. Otherwise, the edges incident on w would be extended into interior lines, and these interior lines would subdivide $\mathcal{C}$. On the other hand, no interior vertex can be a reflex vertex for its bordering visibility cells since it is formed by the intersection of two interior lines which start and end at the polygon boundary. This gives a contradiction. □

Since we can have $\Theta(nr)$ lines forming the cell decomposition, an obvious bound on the total complexity of the cells in the decomposition is $O(n^2r^2)$. However, the structure of our problem can be exploited to obtain the following tighter bound.

THEOREM 3.2. *The number of visibility cells in a given map polygon, as well as their total complexity, is $O(n^2r)$, and this bound is best possible.*

Proof. We will show that the number of subdivision vertices is $O(n^2r)$. There are a total of $O(nr)$ rays, and each ray gives rise to one boundary vertex only. Therefore, it suffices to count nonboundary vertices for the asymptotic bound in the theorem.

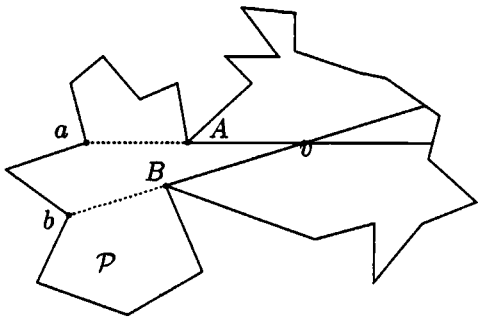


FIG. 4. *A type* 4 *vertex of the visibility decomposition.*

The simplest proof of this theorem consists of considering each of these $O(nr)$ rays separately. Let R be such a ray in the subdivision, anchored at A and emanating from a. Suppose we walk along this ray, starting at A and going away from a. During this walk, there can be at most $2n$ changes in the set of visible vertices. This is because in a simple polygon, once a vertex disappears from view when we walk in the polygon along a straight line, it will never become visible again (otherwise, the occluder in between represents a hole in the polygon). This implies that the ray R has $O(n)$ vertices on it. Since the number of rays is $O(nr)$, we get the desired bound.

However, we now present a more involved proof whose structural aspects will be useful in arguments later on.

Consider a vertex of the subdivision which does not lie on the boundary P. There must be two rays whose intersection gives rise to this vertex v. Let the first ray emanate from the vertex a with the anchor vertex A and the second ray emanate from b with the anchor B. Consider the two lines containing the given rays; they divide the plane into four "quadrants." Notice that if both rays are of the same orientation (i.e., both left or both right), then the corresponding anchors must lie in adjacent quadrants. In the case where the two rays have different orientations, the corresponding anchors lie in either the same quadrant or opposite quadrants.

Label each vertex of the subdivision that does not lie on the boundary of P by the pair of vertices that the two rays determining the vertex emanate from, as well as by the two bits specifying the relative placement of the two anchors in the quadrants defined above. The two bits classify the subdivision vertices into four types. Type 1 vertices are determined by two left rays and have the anchors in adjacent quadrants; type 2 vertices are analogous except that the rays are both of the right orientation. Type 3 vertices have the anchors in the same quadrant, while type 4 vertices have the anchors in opposite quadrants. Figure 4 illustrates a type 4 vertex.

The hardest case to deal with is that of the type 4 vertices—when the two anchors lie in opposite quadrants. Consider a type 4 vertex v determined by two rays, say a ray from a with right anchor A and a ray from b with left anchor B. Suppose we take the portion of the boundary of P from A to B not containing a or b and replace it with the interior segments Av and vB. We have not eliminated any vertices of our subdivision with the same label as v because in the eliminated part of P, at least one of a of b is not visible. In the remaining part of $\mathcal{P}$, there cannot be any other type 4 vertex labeled by a and b which uses either A or B as an anchor for the two rays. Thus there can be at most r vertices with this label. Since the total number of labels

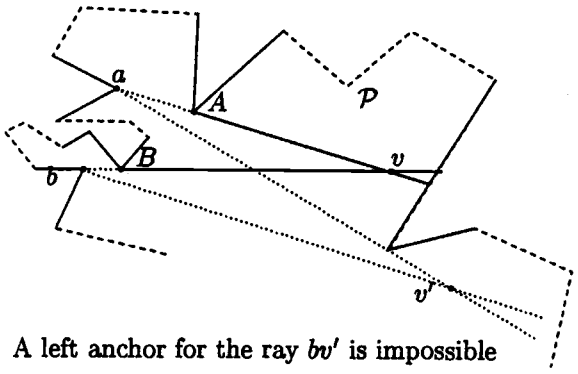


FIG. 5. *Labels with two left anchors are unique.*

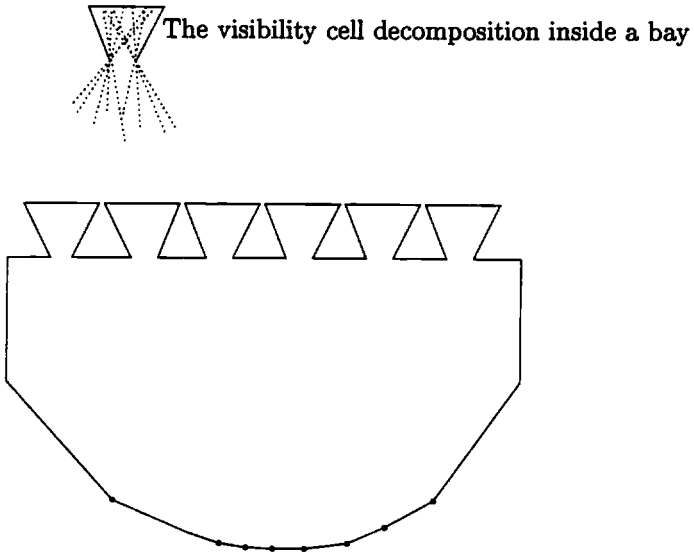


FIG. 6. *A polygon with a large visibility cell decomposition.*

is $O(n^2)$, we have the desired bound.

There are three other cases to consider. In each case, a simple topological argument shows that only a *unique* vertex can possess that label. See, for example, Figure 5, which shows the situation for two left rays. Thus there can be a total of $O(n^2r)$ interior vertices of our subdivision, and so the number of cells is similarly bounded.

For a lower bound, consider a polygon P that has $\Omega(r)$ small bays (each say of three sides) lying roughly along a straight line and facing a convex chain of $\Omega(n)$ sides that is visible from all of them; see Figure 6. Then within each bay, we can get $\Theta(n^2)$ regions corresponding to the visible subchain of the convex chain. This gives a total of $\Theta(n^2r)$ subdivision vertices in total. □

4. Visibility from a cell. We now examine the visibility polygons for all of the points in a particular cell and extract some common features from these views. We must be careful about the assignment of the points on the interior lines to the cells in the decomposition. Consider an interior ray emanating from vertex u and anchored at vertex v. The cells bordering this ray are divided into classes: those which can see u and those which cannot. The boundary edges determined by this ray are assumed to be a part of the latter kind of cells only. Using this rule, each interior vertex gets assigned to a unique cell as well.

The following theorem ties together the notions of a skeleton and a visibility cell.

THEOREM 4.1. *For any visibility cell $\mathcal{C}$ and points $p, q \in \mathcal{C}$, $V^*(p) = V^*(q)$.*

Proof. Consider the straight line joining p and q; call this line l. Clearly, l is totally contained in $\mathcal{C}$, even when p and q are boundary points. Further, no interior line of the decomposition intersects l. Suppose s is the point on l which is the closest to q such that it has a different visibility skeleton than $V^*(q)$.

First, consider the case where $V^*(s)$ and $V^*(q)$ do not have the same underlying polygon, i.e., they have a different set of vertices. Assume without loss of generality that the difference between the two skeletons is that a vertex of P, say x, is visible from s but is not visible from q. Then there must exist a reflex vertex $y \in V^*(q)$ such that it is an obstruction for q viewing x. Then the ray emanating from x and anchored at y must intersect l between s and q, giving a contradiction. Therefore, the underlying polygon for the skeleton is invariant over the entire cell.

Any difference in the skeletons $V^*(q)$ and $V^*(s)$ must then be in the labeling of the edges. It is fairly easy to verify that the locations of full edges and artificial edges must be the same in both cases, as must be the labelings of the artificial edges as corresponding to either half- or partial edges. The difference, if any, must be in the line characterizations that label some artificial edge. In that case, there must exist two edges of P, say e_s and e_q, such that s can see some portion of e_s but cannot see e_q at all, and q can see some portion of e_q but cannot see e_s at all. We then conclude that there is some point between s and q on l which sees a vertex not visible from q. This contradicts the assumption that s was the closest point of l to q which has a different skeleton.

Thus there can be no point s on l which has different visibility skeleton from q. □

This theorem allows us to make the following definition.

DEFINITION 6. *For any visibility cell $\mathcal{C}$, the visibility skeleton of the cell $V^*(\mathcal{C})$ is the common visibility skeleton for all points contained in $\mathcal{C}$.*

The exact choice of an edge e of P for any half- or partial edge is also invariant over the cell, although the portion of e that can be seen varies from point to point in the cell. However, Theorem 4.1 does not guarantee that if two points have the same visibility skeleton, then they are in the same visibility cell. In fact, a visibility skeleton can have r embeddings and could have several distinct visibility cells in its kernel, all with the same skeleton (as was already discussed; see Figure 2). This is because there could be several collinear edges of P that are all candidates for being the half- or partial edge corresponding to a particular artificial edge of the skeleton. If we assume that P contains no collinear edges, then there can be only one relevant visibility cell in the kernel of each embedding of the skeleton.

DEFINITION 7. *The binary relation "$\equiv$" over the visibility cells in the decomposition is such that for any two cells, $\mathcal{C}_1 \equiv \mathcal{C}_2$ if and only if $V^*(\mathcal{C}_1) = V^*(\mathcal{C}_2)$.*

It is easy to verify that this is an equivalence relation over the cells such that each equivalence class of cells is associated with a unique visibility skeleton. Let $\mathcal{E}_i$

for $1 \leq i \leq t$ be the equivalence classes for $\mathcal{P}$, and denote by V_i^* the visibility skeleton of the cells in the ith equivalence class. Also, let e_i denote the total size of all visibility cells in $\mathcal{E}_i$.

Before we embark on explaining our searching data structures, we would like to characterize the complexity of a visibility cell in terms of the complexity of its visibility skeleton. Recall that each side of a cell arises from a ray anchored at an endpoint of an artificial edge.

THEOREM 4.2. *Let $\mathcal{C}$ be a visibility cell whose visibility skeleton has s artificial edges. Then the complexity of $\mathcal{C}$ is $O(s)$.*

Proof. Each side of the polygon $\mathcal{C}$ is determined by a ray anchored at one of the blocking reflex vertices. Each such blocking vertex can have at most two rays which bound $\mathcal{C}$. Since the number of blocking reflex vertices is $O(s)$, the result follows. □

5. Data structures and search algorithms. We now describe the construction of the data structures and algorithms for query processing. In the first step of preprocessing, we compute the visibility cell decomposition of the map polygon and the visibility skeleton for each cell. Each skeleton polygon of size s is represented as an $O(s)$-dimensional real vector. These vectors are stored in a multidimensional search tree, each of whose leaves indexes an equivalence class of cells with the same skeleton. Given $\mathcal{V}(p)$, we extract the visibility skeleton $V^*(p)$ and query this data structure to identify the equivalence class of cells where p must be located. The last and most nontrivial stage of the search is concerned with identifying the possible locations of p within the equivalence class. This reduces to a search problem in a planar subdivision.

5.1. Computing cells and their skeletons. The visibility cells and their skeletons are computed by the following steps: (i) for each reflex vertex, identify the vertices of $\mathcal{P}$ that are visible from it—each such vertex can give rise to a line in the arrangement with one endpoint at that reflex vertex; (ii) compute the arrangement of all of these lines inside $\mathcal{P}$; (iii) compute for each cell in the arrangement the visibility polygon (and hence the skeleton). This last computation can be done in an incremental fashion as we walk along a line of the arrangement; the visibility polygon incurs only one change from one vertex on a line to the next vertex. Let C denote the number of visibility cells and N denote the number of visibility cell vertices; note that C and N are always $O(n^2 r)$.

THEOREM 5.1. *The preprocessing time and space for computing the visibility cell decomposition are $O(nr \log n + nN)$.*

Proof. By standard results in ray shooting in simple polygons [15] and the construction of arrangements via sweepline methods [11], steps (i) and (ii) above can together be completed in $O(nr \log n + N \log n)$ time and $O(nr + N)$ space. To bound the complexity of step (iii), we note that the visibility skeletons of adjacent regions differ in at most a constant number of contiguous edges. Thus we may generate the C visibility skeletons by the following procedure that walks along the lines forming the arrangement. We assume for simplicity of description that there are no degeneracies in the arrangement, i.e., at most two lines intersect at any point of the arrangement except at vertices of $\mathcal{P}$. Assume that every vertex of the arrangement that is not a vertex of $\mathcal{P}$ is labeled by the emanating and anchor vertices of the two lines that intersect to form that vertex.

We first compute the visibility polygons of the r reflex vertices in time $O(nr)$ [14]. Call this set ρ.

We begin at the (reflex) anchor vertex of the first line (call it ℓ) in the arrangement; given ρ, we can in time $O(n)$ obtain the visibility skeleton of the cell of the

arrangement that contains ℓ and its anchor and lies to the right of ℓ. We then walk on the arrangement along ℓ; at each new vertex w of the arrangement, a vertex v of $\mathcal{P}$ either becomes visible or invisible. Indeed, v is the emanating vertex of the other line that intersects ℓ at w. From this we can in $O(n)$ time and space read off and write down the visibility skeleton of the region bounded by line ℓ and vertex w and lying to the right of ℓ. We repeat this for all the lines forming the arrangement, for a total of $O(Nn)$ time and space.

By combining the complexities of all three steps, the result follows. □

Let $\mathcal{R}_i$, $1 \leq i \leq C$, be the regions/cells in the subdivision. Define r_i to be the complexity of $\mathcal{R}_i$, m_i to be the complexity of the $V^*(\mathcal{R}_i)$, and s_i to be the number of blocking vertices in these skeletons. Note that r_i is $O(s_i)$ and that $s_i \leq m_i$.

5.2. Locating the equivalence class. Assume that we have obtained a subdivision of $\mathcal{P}$ into C visibility cells and have computed a visibility polygon for one point in each of the cells. We start by showing how to compute the equivalence classes.

Given a visibility polygon of complexity m, the corresponding skeleton can be computed in $O(m)$ time. We will encode each of the C skeletons as an M-dimensional real vector, where $M = O(m)$. The encoding fixes an origin of the skeleton at a canonical reflex vertex (as described earlier) and specifies the position of every other vertex relative to the origin using $2m$ real numbers. Further, we store the edge labels using another $2m$ components. The ordering of the vertices and edges in the skeleton is specified implicitly by the ordering of the components of the encoding. This entire process takes $O(m)$ time.

A crucial property of the representation is that two cells have the same skeleton if and only if their representations are identical. This motivated the definition of a skeleton and is vital to constructing and searching the equivalence classes.

We first partition the cells according to the number of vertices in their visibility skeletons. Now consider the T_s cells whose skeletons have size s. We construct a multidimensional search tree [17] for the vectors corresponding to the skeleton representations of these cells. These search trees can be constructed in $O(sT_s + T_s \log T_s)$ time and space and support exact match queries in time $O(s + \log T_s)$. The various skeletons whose vector representations are identical will reach the same leaf of this search tree. Thus the leaves are in 1–1 correspondence with the equivalence classes of cells in this collection. (In practice, it would be more efficient to recompute the equivalence search trees using one representative from each equivalence class, after having computed the equivalence classes as above.)

We have at most n different search trees corresponding to the different values of s. Given a query $\mathcal{V}$, we can easily determine which tree to search and thence the correct equivalence class.

THEOREM 5.2. *The t equivalence classes, $\mathcal{E}_i$, can be computed in $O(nN)$ time using the equivalence search trees. The n search trees can all be constructed in $O(nN)$ time and space, and answer queries in time $O(s+\log n)$, where s is the query skeleton size.*

5.3. Searching within an equivalence class. It remains to specify how we search within an equivalence class of visibility cells for all the possible locations of the query point p. We will have one data structure for each of the at most n equivalence classes, associated with the corresponding leaf of the equivalence tree.

We now fix our attention on any one equivalence class $\mathcal{E}_i$. Let V^* be the skeleton corresponding to $\mathcal{E}_i$. Each cell in $\mathcal{E}_i$ can be identified with a distinct embedding of V^*. The cell must lie in the kernel of that embedding. For each embedding, there could be

several cells in the kernel, but these must be disjoint convex polygonal regions. (Recall that there will be only one cell per embedding if P contains no collinear sides.) Let the class $\mathcal{E}_i$ consist of cells from k different embeddings. The complexity of any one cell in $\mathcal{E}_i$ is at most m, the size of $\mathcal{V}(p)$. The following theorem bounds the overall complexity e_i of all the cells in $\mathcal{E}_i$.

THEOREM 5.3. *The total number of cells in any equivalence class $\mathcal{E}_i$, as well as their total complexity e_i, is $O(n^2)$. This bound is best possible.*

It turns out to be much easier to prove a stronger version of Theorem 5.3.

DEFINITION 8. *The class $\mathcal{C}_m$ of cells in the decomposition consists of all cells whose skeletons contain m nonspurious vertices.*

We will show that the total complexity of the cells in the class $\mathcal{C}_m$ is $O(n^2)$. This implies that the total complexity of the cells in an equivalence class is also $O(n^2)$ since these cells must all belong to $\mathcal{C}_m$ for some m.

THEOREM 5.4. *The overall complexity of the cells in $\mathcal{C}_m$ is $O(n^2)$.*

Proof. The longer proof of Theorem 3.2 will prove useful here. Recall that in that proof each vertex of the subdivision was given a label determined by the two rays on which it lies. The label consists of the names of the two emanating map vertices, as well as two bits specifying the layout of the anchor vertices with respect to these rays. There are four types of vertices corresponding to the four possible layouts. There could be as many as n vertices of type 4 which carry the label of a particular pair of map vertices. It is the latter kind of vertices which can be large in number. Overall, there are only $O(n^2)$ vertices which are of types 1–3 or lie on the boundary of the map polygon.

We must now do a careful assignment of the edges and vertices of the subdivision to the visibility cell. The rays determining a vertex create four quadrants, one of which is the region where the two emanating vertices are blocked from view by the anchors. The vertex is assigned to the incident cell which lies in this quadrant. Similarly, each edge lies on a ray on one side of which the emanating vertex is hidden from view by the anchor vertex. The edge is assigned to the adjacent cell which lies on that side of the ray. Thus each edge/vertex is assigned to only one cell, but it still bounds all its neighboring cells.

This assignment of edges and vertices is motivated by our definition of visibility. Consider a ray emanating from v and anchored at w. From the view of any point p on this ray, the vertex v is not visible. Thus in the skeleton $V^*(p)$, there will be a spurious edge from w to v, and v will be considered a spurious vertex.

Now consider the edges which bound the cells in $\mathcal{C}_m$. Some of these edges could be portions of the boundary of the map polygon. There are at most nr rays in the subdivision, and each ray will create one additional subdivision vertex on the boundary P. The number of subdivision edges lying on P cannot exceed the number of subdivision vertices on P, and these are at most $nr + n$ in number. We will ignore all such edges from now on since our goal is to prove a bound of $O(n^2)$.

Consider any particular ray R emanating at the vertex v and anchored at the vertex w. This ray starts at w and proceeds away from v until it hits the boundary P. We will assume that the ray is directed away from v. There may be several subdivision vertices on the ray, besides its endpoints, and this will divide R into a sequence of edges which bound some cells. We will think of these edges as open intervals separated by the vertices on R.

Assume that from the point of view of an observer sitting at v and looking in the direction of the ray, the anchor w lies to the left. We will argue only for the case of such "leftist" rays since the case of the "rightist" rays is similar. Each edge on the

ray will be assigned to the cell lying to its left. However, we will have to count each such edge as contributing to the complexity of both the cell to its right and the cell to its left.

As we trace along the ray from w to the boundary of P, we will traverse the edges on the ray in order. If the cell bounded by an edge (i.e., the cell to its right) is in $\mathcal{C}_m$, then the edge will contribute to the complexity of $\mathcal{C}_m$. Consider the vertices on this ray. Think of the ray as a vertical line which is directed towards the north. Each vertex x on this ray is caused by another ray, call it R', whose anchor may lie to the left or right of R—and this does not really matter. The anchor for R' may lie below it; in this case the vertex is called an *incrementing* vertex. This means that as we cross v, a new vertex comes into view, and this vertex is the emanating vertex for R'. Similarly, a vertex is called *decrementing* if its anchor lies above R'. Upon crossing such a vertex, the emanating vertex falls out of view. The key observation is that a type 4 vertex must be a decrementing vertex, and moreover its anchor must lie to the right of R.

Consider the sequence of edges we see that bound cells (to the right) which are in $\mathcal{C}_m$. Note that since all of these edges are not assigned to the cells to the right, the cells will have v in their skeleton. Thus any two consecutive $\mathcal{C}_m$ edges will be separated by a collection of vertices (including their endpoints) which contains an equal number of incrementing and decrementing vertices. We will label each $\mathcal{C}_m$ edge with the nearest incrementing vertex which lies below it. Since no two $\mathcal{C}_m$ edges are adjacent on R, each such edge gets assigned a unique label except possibly the first such edge on R. The crucial point here is that none of the labels can be a vertex of type 4.

Each vertex can label at most four $\mathcal{C}_m$ edges since it lies on two rays and can be used as a label for at most two $\mathcal{C}_m$ edges on each of these two rays—one each when we are considering the cells to left or to the right of R. Since none of the labels is of type 4, the number of distinct labels is $O(n^2)$. Moreover, the number of unlabeled $\mathcal{C}_m$ edges is at most nr, i.e., one per ray. We conclude that the number of edges contributing to the complexity of cells in $\mathcal{C}_m$ is $O(n^2)$.

It is easy to see that this bound can be reached in the example which proves the tightness of the $O(n^3)$ bound on the complexity of the subdivision. □

Consider any one embedding h of V^* and all of the cells in the kernel of the embedded skeleton which belong to $\mathcal{E}_i$. Let $h(p)$ denote the point in P which has the same location relative to $h(v_o)$ as p has to v_o. The following theorem states that if $h(p)$ lies in one of the cells with skeleton V^*, then $h(p)$ is a valid answer to the query $\mathcal{V}(p)$.

THEOREM 5.5. *Let h be any embedding of $V^*(p)$ and $h(p)$ be the corresponding location of p. Then $\mathcal{V}(h(p)) = \mathcal{V}(p)$ if and only if $V^*(h(p)) = V^*(p)$.*

Proof. Clearly, if $h(p)$ and p have the same visibility polygon, then they have the same skeleton. The nontrivial part is to show that if they have the same skeleton then they have the same visibility polygon.

Since the two points have the same skeleton, all of the nonspurious vertices are identically laid out in the two visibility polygons, as are all the full edges. Now consider any fixed artificial edge e' of their skeleton. Suppose that e' is labeled as being in correspondence with a half-edge. Then in the visibility polygon, there is a spurious edge se which is adjacent to this half-edge. The spurious edge starts at the same reflex vertex in both visibility polygons and is collinear with p. Thus the only difference between the two visibility polygons could be in the location of the spurious vertex where se meets the half edge. However, one endpoint of the half edge

is nonspurious and has the same location in both visibility polygons. Moreover, the line characterization of the half edge is the same in both cases. Therefore, the location of the spurious vertex must also be identical in the two cases.

The other case to be considered is where the artificial edge e' is in correspondence with a partial edge. In this case, there are two spurious edges in the visibility polygon which meet the partial edge. By an argument similar to the one above, it is easy to see that the location of the two spurious vertices must also be identical in the two visibility polygons.

We conclude that the relative location of all the spurious vertices must be identical in $\mathcal{V}(h(p))$ and $\mathcal{V}(p)$, and hence the two polygons must be identical. □

Thus to verify if p could have been in any particular embedding, it suffices to check if $h(p)$ lies in a cell with the same visibility skeleton. Let k denote the total number of valid embeddings. Note that in general each embedding may contain multiple cells with the same visibility skeleton V^*, as we are seeking. Our problem reduces to the following: given a collection of k families of disjoint convex polygons (one for each embedding) of total complexity e, we wish to identify the polygon(s) (if any) where a point q is located. When k is large, searching independently in each embedding's cells would require time at least k, which may be much larger than A (the output size). However, observe that all of the embeddings are congruent and have the same location of p with respect to the origin. The only difference between the embeddings is in the visibility cells therein which have V^* as their skeleton. The cells in different embeddings are possibly totally unrelated to each other.

Our solution is to consider all embeddings at once by overlaying all of their cells into one embedding of the skeleton. Thus with reference to the origin of V^*, we have k collections of convex polygons of total complexity e_i that are overlaid to create a planar subdivision. The problem is now that of performing a point location in this subdivision; each region is labeled by the set of visibility cells that intersect to create it. The overall complexity and time to compute this subdivision is at most $O(e_i^2)$.

Using data structures for point location in planar subdivisions due to Kirkpatrick [16] or Edelsbrunner, Guibas, and Stolfi [12], we can perform point location in time $O(\log n)$ with preprocessing and space $O(e_i^2)$. When we sum over all equivalence classes, our total cost will be

$$O\left(\sum_{i=1}^{t} e_i^2\right) = O\left(n^2 \sum_{i=1}^{t} e_i\right) = O(n^2 N)$$

by Theorem 5.4. In most cases, this will be the dominant preprocessing cost of our final algorithm.

There is actually one more problem to be overcome in implementing this approach: with each region in these subdivisions that we compute, we need to associate a list of the visibility cells whose intersection creates that region. This could blow up the space requirement by as much as another factor of n. To avoid this extra cost, we proceed as follows. Consider a particular subdivision $\mathcal{S}$ obtained by the overlay of congruent visibility skeletons in class $\mathcal{E}_i$ and let $\mathcal{G}$ be the dual graph of that subdivision whose vertices are a set of canonical points, chosen one per region of $\mathcal{S}$. Now double each edge in $\mathcal{G}$ and consider an Eulerian tour τ of this new $\mathcal{G}$ (which must exist since each vertex has even degree and can be computed in linear time in the size of $\mathcal{G}$). Break the tour at some region covered by no cell (e.g., any infinite region of $\mathcal{S}$). The resulting path, τ', can enter and exit each original cell multiple times, but its overall complexity will still be only e_i^2. Each passage of τ' through a cell can be thought of

an interval. Let us construct an interval tree on this collection of e_i^2 known intervals. We can determine the cells covering any point of the path τ' by standard interval-tree searching methods, improved by fractional cascading [6], in total time $O(\log n + A)$, where A is the size of the answer. We can do the same for any point in $\mathcal{S}$ since its coverage will be the same as for the canonical point selected in the region of $\mathcal{S}$ where it falls.

So we finally have our theorem.

THEOREM 5.6. *The localization problem can be solved with preprocessing time and space of* $O(n^2N)$ *and a query time of* $O(m + \log n + A)$.

Note that if we want any one solution, the query time drops to $O(m + \log n)$.

Also, if we assume that P contains no collinear sides, then we can get a better preprocessing bound. Let equivalence class $\mathcal{E}_i$ contain s_i cells of total complexity e_i; each cell must come from a different embedding in this case. The cells are convex, and thus their overlay can have complexity at most $O(s_i e_i)$ and can be computed within the same time bound [1]. Since each s_i is at most r, in this case, the overall space and preprocessing time needed drops to $O(rN)$.

We can also reduce the space in general at the expense of increased query time. Moreover, we can smoothly trade off the query time with the preprocessing time and space.

THEOREM 5.7. *Let* $1 \leq f(n) \leq n$. *The localization problem can solved in* $O(n^2N/f(n))$ *space and preprocessing and query time* $O(m + f(n)\log n + A)$.

Proof. Consider any particular equivalence class $\mathcal{E}_i$. Pick $\gamma = n^2/f(n)$ and consider the embeddings whose complexity is at most γ each. These can be partitioned into groups of embeddings such that each group has complexity roughly γ. The idea is to overlay the cells from the embeddings in a particular group, using a total space of $O(\gamma^2)$ for each such group. The number of such groups cannot exceed $O(n^2/\gamma)$ since the overall complexity of all the embeddings is bounded by $O(n^2)$. The total space required by these groups is $O(n^2\gamma)$. Searching independently in each group's planar subdivision requires time $O(n^2 \log n/\gamma)$.

The embeddings of complexity at least γ each are also searched independently. Their total space requirement is $O(n^2)$. Moreover, since they cannot be more than $O(n^2/\gamma)$ in number, it requires $O(n^2 \log n/\gamma)$ time to search them. Thus our total space requirement for this equivalence class is $O(n^2\gamma)$ and the query time is $O(n^2 \log n/\gamma)$. This implies the desired result. □

The next result is obtained when we perform an independent point location in each distinct embedding.

THEOREM 5.8. *The localization problem can be solved with* $O(nN)$ *space and preprocessing and a query time of* $O(m + r \log n + A)$.

6. Extensions and variants. We briefly outline below how the methods above can be extended to work in a number of additional cases.

6.1. Map polygons with holes. When the map polygon has holes, the size and complexity of the visibility cell decomposition can be higher. In this case, we have a tight bound of $\Theta(n^2r^2)$ on the number of visibility cells. Although we do not have the space to develop this here, we merely mention that when all of the holes are convex, the increase in the number of visibility cells can be bounded in terms of the *number of holes*. This also applies to the increase in the preprocessing and space bounds.

6.2. The limited-range version. We now consider a feature of range finders that arises in practice: they can reliably obtain range readings only up to some

distance D [19]. Beyond this distance, the noise levels are too high to measure the distance, and we learn only that the distance is greater than D. Our approach to preprocessing $\mathcal{P}$ can be modified to work even in this case. The set of points within distance D of an edge of $\mathcal{P}$ is an oval region. Now consider the arrangement of the oval regions defined by all the edges of $\mathcal{P}$; since any two ovals intersect at most at four places, this arrangement partitions the plane into $O(n^2)$ subdivisions. Within each cell of this subdivision, the set of edges of $\mathcal{P}$ that are within distance D is invariant. Intersecting this arrangement with our visibility cell decomposition, we obtain a modified decomposition with the property that in each subdivision, the (redefined) skeletons are the same. Our search process is now applied to this modified decomposition to obtain the same query time.

6.3. The single-shot query problem. Now consider the problem of answering a single query $\mathcal{V}$. Here the cost of any preprocessing must be included in the cost of answering the query. We present an algorithm running in time $O(nm)$ based on some results in ray shooting. Suppose we wish to determine at each vertex of $\mathcal{P}$ where the ray going in a fixed direction first hits $\mathcal{P}$ again. This problem is equivalent to trapezoidalizing $\mathcal{P}$ using lines parallel to the shooting direction. This can be done in linear time [5, 13] given a triangulation of the polygon $\mathcal{P}$. The recent result of Chazelle [4] shows that the triangulation itself can be computed in linear time.

THEOREM 6.1. *Given a map polygon $\mathcal{P}$ and a visibility polygon $\mathcal{V}$, the set of valid locations of p can be determined in time $O(nm)$.*

Proof. First, for each of the $O(m)$ spurious edges in $\mathcal{V}$, we determine the trapezoidalization of $\mathcal{P}$ in the direction of that spurious edge. This requires a total of $O(nm)$ time. Now we try each possible embedding of $\mathcal{V}$ in $\mathcal{P}$, of which there are at most n. In each potential embedding case, we can determine its validity by using the information from the trapezoidalizations queries. Effectively, we are trying to trace a fixed placement of $\mathcal{V}$ in $\mathcal{P}$; we do this by following edges of P and using the trapezoidalizations to implement in constant time ray-shooting queries when we need to walk along a spurious edge of $\mathcal{V}$. Thus the time to verify an embedding is $O(m)$ and the total time is $O(nm)$. □

When the range finder has a limited range D, we can answer a single query in $O(n^2)$ time.

6.4. Visibility query processing. Consider the problem of preprocessing a polygon $\mathcal{P}$ so that a visibility query can be efficiently answered. A visibility query is a point $p \in \mathcal{P}$, and we are required to compute the edges and vertices of P visible from p. An interesting side effect of our results is the construction of an efficient data structure for this problem. The obvious approach is to compute the visibility cell decomposition and store in each cell a modified skeleton for the cell. This new skeleton records the (circular) list of the visible edges and vertices for that cell, which is constant over the cell. A query can now be answered by performing a point location in this cell decomposition. However, this is inefficient in its use of space since we must store the skeletons for each cell separately.

We now sketch an idea for making the space requirement linear in the complexity of the cell decomposition. The idea is to avoid storing the full visibility list at each cell. Instead, at each interior line we store the change in the visibility as we cross it. Borrowing from an idea of Chazelle [3], we actually store the visibility skeleton only at those cells where the visibility is a "local" minimum (i.e., those cells that see less of P than any of their neighbors). These are the cells where crossing any boundary edge leads to additional vertices/edges becoming visible. To compute the visibility in any

cell, we perform a walk to a minimal visibility cell, keeping track of the changes in visibility as we cross interior lines. If we are not at a minimal cell, there is always a cell boundary such that crossing it will cause the number of visible vertices to decrease. The length of the walk cannot exceed the size of the output.

Furthermore, the number of these minimal visibility cells is $O(n^2)$. This is because there cannot be two adjacent type 4 vertices on the boundary of such a minimal cell. One way to see this is to use the analysis in the proof of Theorem 5.4, except that we apply it to edges that bound a minimal visibility cell rather than to edges that bound cells in $\mathcal{C}_m$, as was the goal in that theorem. In particular, consider traversing a ray away from the anchor which (say) lies to the left of the ray; then a minimal cell bounded by an edge e lying on this ray must also lie to the left of the ray. Further, the first endpoint of e that is encountered must be a decrementing vertex (and possibly of type 4), but the second endpoint of e must be incrementing and hence cannot be of type 4. It follows that each minimal cell has at least one vertex not of type 4, and we charge the minimal cell to this vertex. In this fashion, each vertex can be charged at most four times. By recalling that the number of vertices that are not of type 4 is $O(n^2)$, we get the following result.

THEOREM 6.2. *Using $O(n^3)$ space and preprocessing, a visibility query can be answered in time $O(\log n + m)$, where m is the size of the output visibility polygon.*

7. Further work.

- The single-shot problem resembles a classic string-matching problem, and it is likely that an algorithm running in time $o(mn)$ can be devised using techniques from that field.
- We assumed that the robot has a compass and thus $\mathcal{V}$ is oriented with respect to $\mathcal{P}$. What if this assumption were removed?
- In any real-life situation, the data obtained from the sensor is likely to be noisy. How can se solve this problem when we seek only approximate congruence between $\mathcal{V}$ and the visibility polygon of a point in $\mathcal{P}$?
- A hard but natural extension of our problem is to the case of three-dimensional polyhedral terrains. Here the robot's viewing mechanism would be a camera which would deliver two-dimensional images.

Acknowledgments. We gratefully acknowledge useful discussions with Bernard Chazelle, John Hershberger, Simeon Ntafos, and Emo Welzl.

REFERENCES

[1] B. ARONOV, M. BERN, AND D. EPPSTEIN, *Arrangements of polytopes*, manuscript, 1991.
[2] P. BOSE, A. LUBIW, AND J. I. MUNRO, *Efficient visibility queries in simple polygons*, in Proc. 4th Canadian Conference on Computational Geometry, ACM, New York, 1992, pp. 23–28.
[3] B. CHAZELLE, *An improved algorithm for the fixed-radius neighbor problem*, Inform. Process. Lett., 16 (1983), pp. 193–198.
[4] B. CHAZELLE, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.
[5] B. CHAZELLE AND J. INCERPI, *Triangulation and shape-complexity*, ACM Trans. Graph., 3 (1984), pp. 135–152.
[6] B. CHAZELLE AND L. GUIBAS, *Fractional cascading* I: *A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.
[7] R. COLE AND C-K. YAP, *Shape from probing*, J. Algorithms, 8 (1987), pp. 19–38.
[8] I. J. COX, *Blanche: An experiment in guidance and navigation of an autonomous robot vehicle*, IEEE Trans. Robotics Automat., 7 (1991), pp. 193–204.

[9] I. J. Cox and J. B. Kruskal, *On the congruence of noisy images to line segment models*, in Proc. 2nd International Conference on Computer Vision, 1988, IEEE, Piscataway, NJ, pp. 252–258.

[10] I. J. Cox and J. B. Kruskal, *Determining the* 2- *or* 3*-dimensional similarity transformation between a point set and a model made of lines and arcs*, in Proc. 28th IEEE Conference on Decision and Control, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 1167–1172.

[11] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, EATCS Monographs on Theoretical Computer Science, Vol. 10, Springer-Verlag, Heidelberg, Germany, 1987.

[12] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[13] A. Fournier and D. Y. Montuno, *Triangulating simple polygons and equivalent problems*, ACM Trans. Graph., 3 (1984), pp. 153–174.

[14] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, *Linear time algorithms for visibility and shortest path problems inside simple polygons*, in Proc. 2nd Annual ACM Symposium on Computational Geometry, ACM, New York, 1986, pp. 1–13.

[15] L. Guibas, M. Overmars, and M. Sharir, *Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques*, in Proc. 1st Scandinavian Workshop Algorithm Theory, Lecture Notes in Comput. Sci. 318, Springer-Verlag, Berlin, 1988, pp. 64–73.

[16] D. G. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[17] K. Mehlhorn, *Data Structures and Algorithms* 3*: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[18] D. P. Miller, D. J. Atkinson, B. H. Wilcox, and A. H. Mishkin, *Autonomous navigation and control of a Mars rover*, in Automatic Control in Aerospace: Selected Papers from the IFAC Symposium, Pergamon Press, Oxford, UK, 1989, pp. 111–114.

[19] G. L. Miller and E. R. Wagner, *An optical rangefinder for autonomous robot cart navigation*, in Autonomous Robot Vehicles, I. J. Cox and G. T. Wilfong, eds., Springer-Verlag, Berlin, 1990.

[20] C. N. Shen and G. Nagy, *Autonomous navigation to provide long distance surface traverses for Mars rover sample return mission*, in Proc. IEEE International Symposium on Intelligent Control, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 362–367.

[21] S. S. Skiena, *Geometric probing*, Ph.D. thesis, Report UIUCDCS-R-88-1425, Computer Science Department, University of Illinois at Urbana–Champaign, Champaign, IL, 1988.

[22] C. Ming Wang, *Location estimation and uncertainty analysis for mobile robots*, in Autonomous Robot Vehicles, I. J. Cox and G. T. Wilfong, eds., Springer-Verlag, Berlin, 1990.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1139–1165, August 1997

011

A FAST ALGORITHM FOR OPTIMALLY INCREASING THE EDGE CONNECTIVITY*

DALIT NAOR†, DAN GUSFIELD†, AND CHARLES MARTEL†

Abstract. Let $G = (V, E)$ be an undirected, unweighted graph with n nodes, m edges and edge connectivity λ. Given an input parameter δ, the *edge augmentation problem* is to find the smallest set of edges to add to G so that its edge connectivity is increased by δ. In this paper, we present a solution to this problem which runs in $O(\delta^2 nm + \delta^3 n^2 + nF(G))$, where $F(G)$ is the time to perform one maximum flow on G. In fact, our solution gives the optimal augmentation for *every* δ', $1 \leq \delta' \leq \delta$, in the same time bound. By introducing minor modifications to the solution, we can solve the problem without knowing δ in advance, and we can also solve the node-weighted version and the degree-constrained version of the problem. If $\delta = 1$, then our solution is particularly simple; it runs in $O(nm)$ time, and it is a natural generalization of the algorithm in [K. Eswaran and R. E. Tarjan, *SIAM J. Comput.*, 5 (1976), pp. 653–665] for the case where $\lambda + \delta = 2$. We also solve the converse problem in the same time bound: given an input number k, increase the connectivity of G as much as possible by adding at most k edges. Our solution makes extensive use of the structure of particular sets of cuts.

Key words. graph algorithms, graph connectivity, network flow, minimum cuts

AMS subject classifications. OSC40, OSC75, OSC85, 68R10, 68Q25

PII. S0097539792234226

1. Introduction.

Consider an undirected, unweighted graph $G = (V, E)$ with n nodes, m edges, and edge connectivity λ. Given a parameter δ, the *edge augmentation problem* is to find a smallest set of edges to add to G so that its edge connectivity is increased to $(\lambda + \delta) = C$.

The edge augmentation problem is a fundamental problem in graph theory. It has direct applications in the design of reliable networks [14, 15, 16] and indirect applications in problems of statistical data security [19]. More abstractly, the problem is of interest as a demonstration of the algorithmic power that comes from understanding and exploiting subtle graph-theoretic structure. Any solution to this problem must add at least $C - \lambda'$ new edges across any cut of size λ', $\lambda' < C$. Hence to obtain an efficient solution, one must make extensive use (explicitly or implicitly) of the underlying *structure* of the set of cuts of size up to C in G, and one must derive and use this structure efficiently.

In this paper, we give a solution to the edge augmentation problem which runs in $O(\delta^2 nm + \delta^3 n^2) + nF(G)$ time, where $F(G)$ is the time to perform one maximum flow in G. For the case of $\delta = 1$, our solution is particularly simple and runs in $O(nm)$ time. The edge augmentation problem was first shown to be solvable in polynomial time by Watanabe and Nakamura [36], who gave an algorithm that works in $O((\lambda + \delta)^2 n^4((\lambda + \delta)n + m))$ time, and this was improved by Watanabe [35] to

* Received by the editors July 15, 1992; accepted for publication (in revised form) August 23, 1995. This research was supported by National Science Foundation grants CCR-8803704 and CCR-9023727. A preliminary version of this paper appears in *Proc.* 31*st Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 698–707.

http://www.siam.org/journals/sicomp/26-4/23422.html

† Department of Computer Science, University of California at Davis, Davis, CA 95616 (dnaor@math.tau.ac.il, gusfield@cs.ucdavis.edu, martel@cs.ucdavis.edu). First author's current address, IBM Haifa Research Lab, Tel-Aviv Annex, IBM Building, 2 Weizmann Street, Tel-Aviv 61336, Israel (dalit@haifa.vnet.ibm.com).

$O((\lambda+\delta)^2 n^3((\lambda+\delta)n+m))$. Cai and Sun [5] also considered the problem and described a solution but did not provide any time analysis. An initial version of our algorithm appeared in [30]. Independently, using a quite different approach, Frank [9] gave an $O(n^5)$ algorithm for the problem which extends to a more general augmentation problem where local connectivities may be prescribed. A new implementation of our algorithm, which uses a poset representation for minimum cuts, was given by [11].

The edge augmentation problem is a particular instance of the general *graph augmentation problem*. This general area was first investigated by Eswaran and Tarjan [7], who gave a polynomial-time algorithm for the problem of augmenting an unweighted graph to make it 2-edge-connected (bridge-connected). In our terminology, this is the special case of $\lambda + \delta = 2$. They also showed that the corresponding weighted case is NP-complete. Later, Fredrickson and JáJá [10] showed that the 2-edge-connected weighted problem remains NP-complete even if the initial graph is a tree and gave an approximation algorithm for the problem. The problem of optimal strong augmentation of a mixed graph (a graph with both undirected and directed edges) was given an efficient solution by Gusfield [17]. Kajitani and Ueno [22] and Ueno, Kajitani, and Wada [34] solved the unweighted problem for all graphs which are trees, directed and undirected (a special case of $\lambda = 1$). Their method is polynomial, but no time bounds are stated explicitly in their papers. Both [36] and [5] deal with the general edge augmentation problem and show its polynomiality. These papers are very long, and the methods described therein are quite involved. The method of Frank [9] solves the edge augmentation problem for both directed and undirected graphs and is based on a different approach. It extends to a more general problem when local connectivities may be prescribed, and it generalizes to a strongly polynomial algorithm. Its extension by [3] applies to mixed graphs. In [11], the directed and undirected versions are considered. All methods allow, and often require, the addition of parallel edges. For the vertex analogue of the problem (i.e. adding edges to increase the vertex connectivity), much less in known; see [7, 32, 37, 20, 21] for results on the special case where $\lambda + \delta = 2$ or 3.

The main result in this paper is a solution to the edge augmentation problem for undirected, unweighted graphs for *any* initial edge connectivity λ and *any* increment δ. Our method is faster than the previous methods: its running time is $O(\delta^2 nm + \delta^3 n^2)$, with an initial cost of $nF(G)$ (when $\delta > 1$), where $F(G)$ is the best time to compute a maximum flow in an unweighted graph. (The best $F(G)$ known is $O(\min\{n^{2/3}m, m^{3/2}\})$ for simple graphs and $O(m^{3/2})$ for multigraphs [8, 2].) This is a considerable improvement over the $O((\lambda + \delta)^2 n^3((\lambda + \delta)n + m))$ method of [35], and when δ is not very large ($\delta < n$), it improves the $O(n^5)$ method of [9]. Our method is also simple and intuitive and allows a self-contained exposition. It can be easily adapted to solve the augmentation problem even without the knowledge of a target increment δ in advance, and yet it produces a solution that is optimal for any intermediate increment. It can therefore be used to solve the converse problem: given a number k, what is the highest connectivity that can be achieved by adding at most k new edges to the graph? It can also be adapted to solve two other variations on the problem which were suggested and first solved by Frank [9]: the node-weighted version and the degree-constrained version. In the node-weighted version, each node x has a weight $cost(x)$; the goal is to find a minimum-weight set of edges that solve the augmentation problem, where the weight of adding an edge (x, y) is $cost(x) + cost(y)$. In the degree-constrained version, each vertex v has an upper bound $g(v)$ on its degree; the task is to solve the augmentation problem subject to the constraint that for every

vertex v, its degree in the final graph does not exceed $g(v)$.

For the case where the increase in connectivity is one ($\delta = 1$), our method is particularly simple, and is a natural generalization of the algorithm of [7] for the 2-edge connectivity case ($\lambda + \delta = 2$). Its running time is $O(nm)$, compared with the $O(\lambda^2 n^3(\lambda + m))$ of [35] or $O(n^5)$ of [9]. Note that $O(nm)$ is also the best time bound, in terms of n and m, for computing the connectivity value alone [31] (also described in [1]), [26]. (Nagamochi and Ibaraki [27] achieved a better bound of $O(m + \min\{\lambda n^2, pn + n^2 \log n\})$ for computing the connectivity, p being the number of pairs of vertices that are connected by an edge; when expressed only in terms of n and m, it reduces to $O(nm)$. Gabow [11] achieved $O(m + \lambda m \log(n^2/m))$.) Hence within the same best time bound for computing just the connectivity value, we can optimally increase the edge connectivity by one. (The implementation of our algorithm in [11], which uses a poset representation of cuts, runs in $O(m + \lambda^2 n \log(n/\lambda))$ for $\delta = 1$ and in $O(m + (\lambda + \delta)^2 n \log n)$ for $\delta > 1$.)

Given the efficiency and simplicity of the solution for $\delta = 1$, a natural approach to the problem of general δ is to successively (and optimally) increase the connectivity by steps of one until connectivity $\lambda + \delta$ is reached. Unfortunately, it is not true that *every* optimal set of edges which increases the connectivity by one can be extended to an optimal solution to increase the connectivity by $\delta > 1$. This approach even fails for $\delta = 2$ and $\lambda = 1$, as demonstrated in Figure 1. The graph G in Figure 1 becomes 3-edge-connected if the edges $\{(1, 11), (2, 12)\}$ are added. G becomes 2-edge-connected if the edge $(6, 8)$ is added (which is an optimal augmentation by one), but then *two* more edges must be added to make G 3-edge-connected (say, $\{(1, 11), (2, 12)\}$). So for the general problem, a less myopic method is needed. Though not every optimal augmentation by one can be extended to an optimal augmentation by $\delta > 1$, it is true that *there exists* an optimal solution for the case where $\delta > 1$ consisting of a sequence of δ augmentations where each augmentation successively increases the connectivity by one using the minimum number of edges. Such a sequence is given in [35] as an improvement over the algorithm in [36]. Our solution also finds such sequence. In fact, the sequence of δ *optimal* augmentations by one found by our algorithm also has the property that it is optimal for *any* intermediate $\delta' \leq \delta$, and hence the target δ need not be specified in advance. In order to construct this sequence, the algorithm needs to select each optimal augmentation by one with the future augmentations "in mind."

The approach employed by our algorithm is similar at its high level to the one used in [36]. It also uses similar results to prove the correctness of the algorithm. See section 4.6 for further discussion. However, we obtain substantially better running times by using a better algorithm for augmenting by one and by improving the time to do successive augmentations. The successive augmentations are sped up by using improved algorithms, data structures, and structural properties of the graph.

A crucial component of our solution is the use of compact structures which implicitly represent various information about sets of cuts. We use three such structures. The first is the Gomory–Hu equivalent flow tree [13], the second is one that we call the extreme-sets tree, and the third is a compact representation of the set of *all* connectivity cuts in a graph, where a connectivity cut is an edge cut of size λ. This representation is described by [6]. The use of this relatively unknown structure (at least in the West) is one of the keys to our faster solution.

This paper is organized as follows. In section 2, we describe the compact representation of the connectivity cuts. Section 3 contains the $O(nm)$-time solution for the

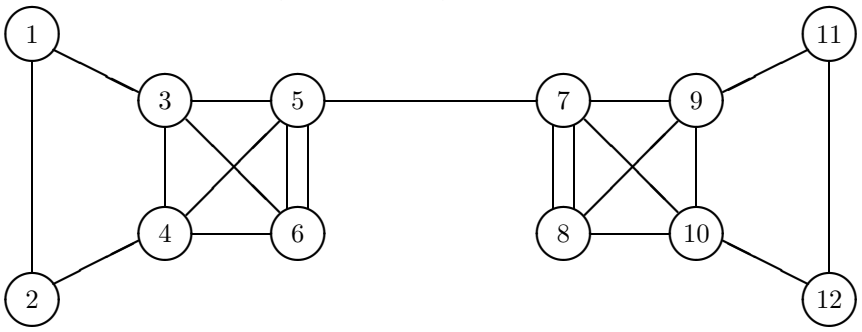


FIG. 1. *An example where an optimal augmentations by one cannot be extended to yield an optimal augmentation by two.*

edge augmentation problem where $\delta = 1$. In section 4, the solution for any δ is given. Section 5 refers to other variations on the problem: the no-target augmentation problem, the node-weighted version, and the degree-constrained problem. Finally, section 6 suggests as open questions a few possible extensions, in particular, the augmentation problem with no parallel edges.

2. Representing all connectivity cuts. We start by describing an elegant, compact representation for the sets of *all* connectivity cuts. In a graph $G = (V, E)$ with n nodes, m edges, and connectivity λ, a *connectivity cut* is a set of λ edges whose removal from the graph disconnects it into two sets of nodes, A and $\overline{A}$. We denote such a cut by $(A, \overline{A})$. The connectivity cut might not be unique; however, there can be only $O(n)$ of them if λ is odd and $O(n^2)$ of them if λ is even (we refer to [6, 23] and, indirectly, [4]).

The set of all connectivity cuts can be compactly represented by a simple edge-weighted graph $\mathcal{H}(G)$ of $O(n)$ nodes and edges [6], and this representation can be constructed in $O(nm)$ time [24] (see also [28, 29]). Each vertex in G maps to exactly one node in $\mathcal{H}(G)$ so that any node in $\mathcal{H}(G)$ corresponds to a subset (possibly empty) of vertices from G. A cut $(S, \overline{S})$ in $\mathcal{H}(G)$ defines a cut $(A, \overline{A})$ in G, where A consists of all the vertices from G that are mapped into nodes of S. Each minimal cut (a cut such that no proper subset of its edges is also a cut) in $\mathcal{H}(G)$ is also a λ-cut (a cut of total edge weight λ) in $\mathcal{H}(G)$, and it corresponds to a connectivity cut in G; each connectivity cut in G corresponds to one or more λ-cuts (or minimal cuts) in $\mathcal{H}(G)$. Hence the minimal cuts in $\mathcal{H}(G)$ compactly represent the connectivity cuts in G.

If λ is odd, then $\mathcal{H}(G)$ is particularly simple—it is a tree, all its edges are of weight λ, and any minimal cut in $\mathcal{H}(G)$ is obtained by removing one of its edges. If λ is even, then $\mathcal{H}(G)$ is called a *cactus*—it can contain cycles, but any two cycles have *at most* a single vertex in common, so no edge is in more than one cycle. Every edge in a cycle is called a *cycle edge* and is given weight $\lambda/2$; every other edge is called a *tree edge* and is given weight λ. Hence minimal cuts in $\mathcal{H}(G)$ are of exactly two types: a cut of the first type is obtained by removing a tree edge, and a cut of the second type is obtained by removing any pair of cycle edges that lie on the same cycle. Every tree is a cactus with no cycles; hence to unify our discussion, we always refer to $\mathcal{H}(G)$ as a cactus. Throughout the paper, we use the term "vertex" to denote a vertex in G and the term "node" to denote a node in $\mathcal{H}(G)$. In Figure 2 we give an example of a graph G with connectivity 4, and its connectivity cuts are represented by $\mathcal{H}(G)$.

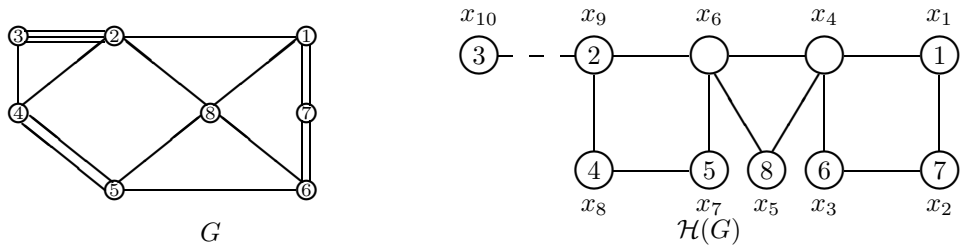


FIG. 2. *A graph G and its cactus representation $\mathcal{H}(G)$ ($\lambda = 4$). The weight of a solid edge in $\mathcal{H}(G)$ is 2, and the weight of a dashed edge is 4.*

3. Increasing connectivity by one. Our solution to the augmentation problem for $\delta = 1$ uses the representation $\mathcal{H}(G)$ and is a natural generalization of the method in [7] for the 2-edge-connectivity augmentation problem ($\lambda + \delta = 2$). To see that our solution generalizes the solution in [7], note that if the graph is originally 1-edge-connected, i.e., $\lambda = 1$, then the representation of all 1-cuts in the graph is well known: it is the tree obtained by contracting each of the 2-edge-connected components into a single node. The augmentation algorithm in [7] finds this representation for the 1-cuts and optimally augments it by one using a depth-first numbering of the tree leaves. Our method finds the cactus representation $\mathcal{H}(G)$ for *any* λ and augments it by one using a depth-first traversal of the cactus that generalizes the one of [7].

DEFINITION. *We say that a node u is a leaf in a cactus if either u is connected by a single tree edge or u is in a cycle and has degree exactly 2. Note that if there are no cycles in the graph, then this is the usual definition of a leaf (in Figure 2, the nodes $x_1, x_2, x_3, x_5, x_7, x_8, x_{10}$ are leaves).*

Let $u_1, \ldots, u_k$ be the leaves of $\mathcal{H}(G)$ and U_i be vertices in G which map to u_i, and let $(U_1, \overline{U_1}), \ldots, (U_k, \overline{U_k})$ be the λ-cuts of G defined by these leaves, i.e., either by the removal of the single incident tree-edge or by the removal of the two incident cycle-edges.

LEMMA 3.1. *If $\mathcal{H}(G)$ has k leaves, then at least $\lceil k/2 \rceil$ edges must be added to increase the connectivity of G by one.*

Proof. First, note that none of the leaves of $\mathcal{H}(G)$ can be empty subsets of vertices from G since otherwise they would not correspond to a λ-cut in G. Hence every u_i defines a λ-cut $(U_i, \overline{U_i})$ in G, so at least one new edge must go across each U_i, $1 \le i \le k$. Since the U_i's are disjoint, a new edge can satisfy at most two of these requirements. Hence at least $\lceil k/2 \rceil$ edges are needed. □

We now show that this bound is achievable. We first need to define a special type of depth-first-search (DFS) traversal in a cactus. This traversal consists of two stages. The first stage assigns different colors to the different simple cycles in the cactus so that all cycle edges of the same cycle are assigned the same color. Since every cycle edge is in exactly one cycle, such a coloring procedure is possible. It can be done efficiently, in $O(n + m)$ time, by adapting any procedure that finds the articulation points in a graph (see, for example, [25, 33]). The second stage is a DFS traversal that starts at an arbitrary node and obeys the following rule: if a node u is visited for the first time via a cycle edge that is colored red (say), then traverse all other edges incident at u *before* traversing the other red edge incident to u. Note that if the graph

has no cycles, then this is the usual DFS tree traversal.

An alternative presentation of the above DFS traversal is as follows: if every tree edge in the cactus is replaced by a pair of parallel edges, then the cactus becomes an Eulerian graph. The DFS traversal essentially finds some Eulerian tour in the graph.

We can now fully describe our solution to the problem of augmenting the connectivity of a graph by one.

ALGORITHM **Aug-1**

(1) Construct $\mathcal{H}(G)$, the representation of all connectivity cuts.

(2) Traverse $\mathcal{H}(G)$ in the modified DFS manner described above and enumerate its leaves $u_1, \ldots, u_k$ in the order in which they are first encountered in the DFS traversal.

(3.a) Form the pairs $\{(U_i, U_{i+\lceil k/2 \rceil}) : 1 \leq i \leq \lfloor k/2 \rfloor\}$, where U_i is the set of vertices from G that map to the leaf u_i of $\mathcal{H}(G)$.

(3.b) For each pair $(U_i, U_{i+\lceil k/2 \rceil})$, $1 \leq i \leq \lfloor k/2 \rfloor$, pick an *arbitrary* vertex from G in U_i and an *arbitrary* vertex from G in $U_{i+\lceil k/2 \rceil}$ and connect them by an edge. If k is odd, then connect some vertex in $U_{\lceil k/2 \rceil}$ to a vertex in an arbitrary different leaf U_j.

Note that the nonuniqueness of the solution comes from step (3.b) since it does not specify *which* vertex from G in U_i and $U_{i+\lceil k/2 \rceil}$ should be connected by the edge. This will be important in the solution for the case where $\delta > 1$. To prove the correctness of the algorithm, we need the following lemma.

LEMMA 3.2. *If the leaves of $\mathcal{H}(G)$ are numbered in the order they are first encountered in the above DFS traversal then, for every λ-cut $(S, \overline{S})$ of $\mathcal{H}(G)$, all the leaves in S have consecutive numbers, and all the leaves in $\overline{S}$ have consecutive numbers (under the definition that 1 succeeds k).*

Proof. Recall the two types of λ-cuts in $\mathcal{H}(G)$. First, suppose that $(S, \overline{S})$ is obtained by the removal of a tree edge of weight λ from $\mathcal{H}(G)$. In this case, the claim follows directly since it is the fundamental property of a DFS leaves enumeration in a tree. Now suppose that $(S, \overline{S})$ is obtained by the removal of two cycle edges (from the same cycle), each of weight $\lambda/2$. Let $c_1, \ldots, c_p$ be the vertices on this cycle, and without loss of generality, suppose that c_1 is the first vertex of this cycle to be visited in the DFS traversal. Then $c_1, \ldots, c_p$ must be visited in either this order or the counterclockwise order since any c_i can be reached from c_1 only via the cycle. Also, our DFS-traversal rule implies that for every c_i, the subgraph that is attached to c_i by edges not from this cycle is traversed *before* c_{i+1}. Hence since $(S, \overline{S})$ splits the cycle into two consecutive parts, the leaves on both sides must have consecutive numbers. ∎

THEOREM 3.3. *Algorithm* **Aug-1** *optimally increases the connectivity of G by one and can be implemented in $O(nm)$ time.*

Proof. We first show that at least one new edge crosses every λ-cut in $\mathcal{H}(G)$ (and thus crosses every λ-cut in G). Let $(S, \overline{S})$ be a λ-cut in $\mathcal{H}(G)$. By Lemma 3.2, the leaves in each side of the cut have consecutive DFS numbers (where 1 succeeds k). Without loss of generality, assume that S contains less than or equal to $k/2$ leaves, and let $u_i \in S$ (S must contain at least one leaf). If $i < \lceil k/2 \rceil$, $u_{i+\lceil k/2 \rceil}$ must lie in $\overline{S}$, so the edge $(u_i, u_{i+\lceil k/2 \rceil})$ crosses it. If $i > \lceil k/2 \rceil$, $u_{i-\lceil k/2 \rceil}$ must lie in $\overline{S}$, so the edge $(u_i, u_{i-\lceil k/2 \rceil})$ crosses it. Finally, if $i = \lceil k/2 \rceil$, the new edge attached to u_i crosses S since it is connected to a leaf different than u_i. Hence the connectivity of the new graph has increased by at least one. This is also an optimal augmentation since by Lemma 3.1, $\lceil k/2 \rceil$ is a lower bound on the number of edges that must be added, and

this method achieves this bound. Since $\mathcal{H}(G)$ can be constructed in $O(nm)$ time and can be traversed in $O(n+m)$ time, the total running time of algorithm **Aug-1** is $O(nm)$. □

Example. Consider the graph G in Figure 2, and apply algorithm **Aug-1** to G. The DFS traversal visits the leaves of $\mathcal{H}(G)$ in the following order: x_1, x_2, x_3, x_5, x_7, x_8, x_{10}. The pairs formed are (x_1, x_7), (x_2, x_8), and (x_3, x_{10}). Since in this example every leaf contains exactly one vertex from G, these pairs immediately translate to the edges $(1,5)$, $(7,4)$, and $(6,3)$. Finally, leaf x_5 is connected to x_1 (this is an arbitrary decision), so the last edge added is $(8,1)$.

4. Increasing connectivity by δ. The method for $\delta > 1$ is considerably more complicated than the solution for the case where $\delta = 1$. Given that a simple solution for the $\delta = 1$ case exists, one approach to the $\delta > 1$ case is to successively augment by one for δ steps by applying algorithm **Aug-1** δ times. We show that this approach *can* yield an optimal solution if some caution is taken at each step of the application. Recall that the method of algorithm **Aug-1**, which augments optimally by one, does not necessarily produce a unique solution: it specifies pairs of leaves in $\mathcal{H}(G)$ that are to be connected by an edge, without suggesting which vertex from G in each leaf should become the endpoint of the edge; hence if the leaf is not a singleton, various choices exist. We take advantage of this "freedom." We show that by looking deeper "inside" these leaves to find cuts of higher values that will have to be augmented later, we can find an optimal solution to the augmentation by one that can be extended to an optimal solution for δ. That is, we make a more "intelligent" choice in step (3.b) of algorithm **Aug-1** when choosing the current solution. By showing that this is possible, we in fact provide another proof to the result of [35] which states that there is an optimal solution to the case where $\delta > 1$ which consists of a sequence of δ optimal solutions to the problem of augmenting by one.

We start with section 4.1, which establishes an obvious lower bound on the number of edges that must be added to augment a graph. Next, in section 4.2, we define a partition, called the *extreme-sets partition*, used to guide the search for an "intelligent" choice at each stage of augmenting by one. Section 4.3 describes the complete algorithm that finds an optimal solution by using the extreme-sets partition together with successive applications of algorithm **Aug-1**. Finally, in sections 4.4 and 4.5, we prove the optimality of the algorithm and its running time.

4.1. The lower bound. We now state a lower bound from [36, 5, 9] on the number of edges that must be added for any increment δ. It is used to prove the optimality of our algorithm. For $U \subset V$, denote by $d(U)$ the number of edges connecting vertices in U and vertices in $V \setminus U$. We call $d(U)$ the *degree* of U.

Let P be any partition of V into disjoint subsets $P_1, \ldots, P_k$. Define $\Phi(P)$, the *edge demand* of P, as $\Phi(P) = \sum_{i=1}^{k} \max\{0\ ,\ (\lambda+\delta) - d(P_i)\}$.

OBSERVATION 4.1. (See [36, 5, 9].) *For any partition P, at least $\lceil \Phi(P)/2 \rceil$ edges must be added to increase the connectivity of G by δ.*

Proof. If for some subset P_i, $d(P_i) < \lambda + \delta$, then at least $(\lambda+\delta) - d(P_i)$ new edges that have one endpoint in P_i and cross $(P_i, \overline{P_i})$ must be added. Since the subsets of P are all disjoint, each edge can satisfy at most two of the requirements. Hence at least $\lceil \Phi(P)/2 \rceil$ edges are needed. □

COROLLARY 4.1 (The lower bound). *The minimum number of edges that must be added to a graph to make it $(\lambda+\delta)$-edge-connected is greater than or equal to $\max_P\{\lceil \Phi(P)/2 \rceil\}$ over all partitions P. Also, any augmentation which for some par-*

tition P makes the graph $\lambda + \delta$ edge-connected by adding $\lceil \Phi(P)/2 \rceil$ edges is optimal.

As shown in [36, 5, 9], this lower bound is achievable. Moreover, the minimum number of edges that are needed equals the ceiling of half of the demand of some partition and hence equals the ceiling of half of the maximum edge demand over all partitions (a min–max relation).

4.2. The extreme-sets partition. We define a partition of the vertices of G, called the *extreme-sets partition*, denoted ES, and later show that the number of edges added by our algorithm always equals $\lceil \Phi(ES)/2 \rceil$, where $\Phi(ES)$ is the edge demand of the partition ES, thus proving that ES has the maximum edge demand over all partitions.

DEFINITION. *A set $U \subset V$ is* k-extreme *if and only if its degree k is* strictly smaller *than the degree of* each *of its proper subsets, i.e., if and only if $d(U) = k$ and for any $W \subset U$, $d(W) > k$.*

LEMMA 4.2. *If X is k-extreme, $Y \neq X$ is l-extreme, and $k \leq l$, then either $Y \subset X$ or X and Y are disjoint.*

Proof. Clearly, $X \not\subset Y$ since otherwise Y is not an extreme set. Suppose that X and Y properly intersect. There can be two cases: either $X \cup Y = V$ or $X \cup Y \subset V$. If $X \cup Y = V$, then $\overline{X} \subset Y$; but $d(\overline{X}) = d(X) = k$ (since the graph is undirected), and $k \leq l$, contradicting the fact that Y is l-extreme. If X and Y properly intersect and $X \cup Y \subset V$, then V is divided into four nonempty quadrants $X \cap Y$, $X \cap \overline{Y}$, $\overline{X} \cap \overline{Y}$, and $\overline{X} \cap Y$. We number them as quadrants 1, 2, 3, and 4, respectively, and denote by d_{ij} the number of edges between quadrant i and quadrant j, $1 < i, j < 4$ ($d_{ij} = d_{ji}$). See Figure 3.

Since X and Y are both extremes, $d(X \cap \overline{Y}) > k$ and $d(\overline{X} \cap Y) > l$, so

$$d(X \cap \overline{Y}) + d(\overline{X} \cap Y) > k + l. \tag{1}$$

Also, $d(X) = k = d_{14} + d_{13} + d_{24} + d_{23}$ and $d(Y) = l = d_{12} + d_{13} + d_{24} + d_{43}$; hence $d_{14} + d_{24} + d_{23} \leq k$ and $d_{12} + d_{24} + d_{43} \leq l$. Note that $d(X \cap \overline{Y}) = d_{21} + d_{24} + d_{23}$ and $d(\overline{X} \cap Y) = d_{14} + d_{24} + d_{34}$, so

$$d(X \cap \overline{Y}) + d(\overline{X} \cap Y) = d_{12} + d_{23} + d_{34} + d_{14} + 2d_{24} \leq k + l, \tag{2}$$

but (2) contradicts (1). In summary, X cannot properly intersect with Y, and $X \not\subset Y$; hence the lemma follows. □

Lemma 4.2 is the key to defining the extreme-sets partition. Define the set of vertices V to be a 0-extreme set, and note that any single vertex u is $d(u)$-extreme, where $d(u)$ is the degree of u. By Lemma 4.2, the collection of extreme sets in G has the property that either any two sets are disjoint or one is contained in the other (a hierarchical structure). Such a collection can be represented by a tree in a natural way. In this tree, every leaf corresponds to a vertex in V, and the root corresponds to the set V; every other node in the tree corresponds to an extreme set, and a node y is an ancestor of another node x if and only if the extreme set associated with node y contains the extreme set associated with node x. We call this tree the *extreme-sets tree*, denoted EST.

Since we have a hierarchical structure of sets, we can now use a definition that was proposed in [36]: for any node x in the extreme-sets tree, let X be the set of vertices in G that correspond to x (the leaves in the subtree rooted at x). Starting at the leaves, for each node x other than the root, recursively define $\Phi(x)$ as the edge

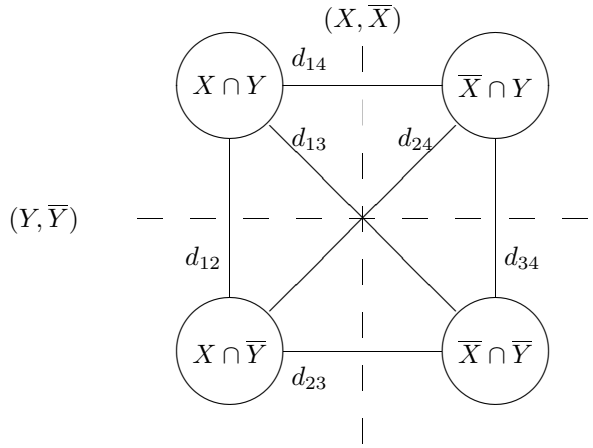


FIG. 3. *The four nonempty quadrants generated by two crossing cuts* $(X, \overline{X})$ *and* $(Y, \overline{Y})$.

demand of x,

$$\Phi(x) = \max \left\{ 0, (\lambda + \delta) - d(X), \sum_{\text{y child of x}} \Phi(y) \right\},$$

and the edge-demand of the root r as $\Phi(r) = \sum_{\text{y child of r}} \Phi(y)$. Intuitively, $\Phi(x)$ is a lower bound on the number of edges that need to be attached to vertices in X in order to increase the connectivity of G by δ; if $d(X) < (\lambda + \delta)$, then at least $(\lambda + \delta) - d(X)$ edges need to be attached to vertices in X, and this argument holds recursively for any subset of X. (The children of x correspond to disjoint subsets of X.)

If r is the root of the extreme-sets tree EST, then the recursive procedure **Find_Part(r)** finds a partition of V into subsets $X_1, \ldots, X_k$ such that the edge demand of this partition equals the edge demand of r. This partition is called the *extreme-sets partition* and is denoted by $ES = X_1, \ldots, X_k$. **Find_Part(r)** finds a set of nodes in the tree as close to the root as possible such that the edge-demand of each node is either zero or (if greater than zero) strictly greater than the sum of the edge demands of its children. Initially, ES is empty.

Find_Part(x)
If x is a leaf, then $ES \leftarrow ES \cup \{X\}$.
Else, for every child y of x,
  If $\Phi(y) = 0$ or $\Phi(y) > \sum_{\text{z child of y}} \Phi(z)$, then $ES \leftarrow ES \cup \{Y\}$
  Else **Find_Part(y)**

Clearly, ES is a partition of V and is uniquely defined by the tree and the edge demands of its nodes. Also, if $ES = \{X_1, \ldots, X_k\}$, then $\Phi(r) = \sum_{i=1}^{k} \max\{0, (\lambda + \delta) - d(X_i)\}$ since if a child y of the root did not contribute its corresponding set Y to the partition ES, then $\Phi(y) = \sum_{\text{z child of y}} \Phi(z)$, and this property holds recursively. Hence $\Phi(ES) = \Phi(r)$.

Figure 4 shows the extreme-sets tree of the graph G depicted in Figure 1. If $\delta = 2$, then $\Phi(x_5) = \Phi(x_6) = \Phi(x_7) = \Phi(x_8) = 0$, $\Phi(x_4) = 0$, $\Phi(x_2) = \Phi(x_3) = 1$, and $\Phi(x_1) = 2$. The left and the parts of the tree are symmetric; hence $\Phi(r) = 4$. The extreme-sets partition is therefore $ES = \{1\}, \{2\}, \{3, 4, 5, 6\}, \{7, 8, 9, 10\}, \{11\}, \{12\}$ and $\Phi(ES) = 4$.

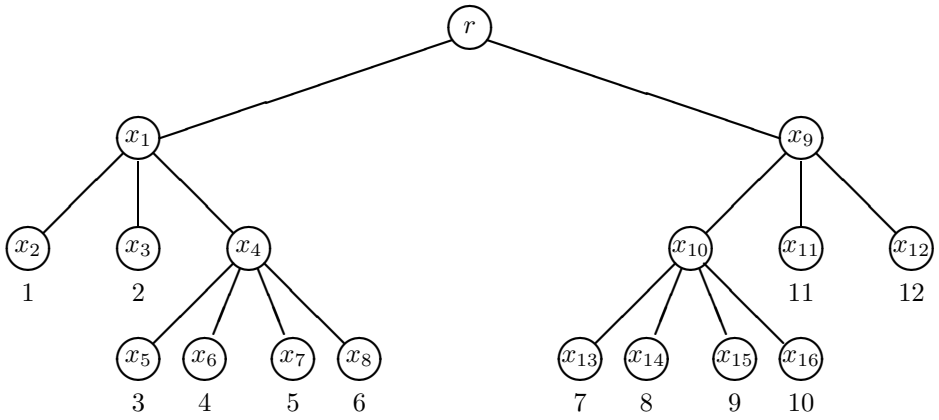


FIG. 4. *The extreme-sets tree of the graph G from Figure* 1.

Extreme sets play an important role in the augmentation problem. First, note that λ-extreme sets are the immediate children of the root of EST and that a set U of vertices in G is λ-extreme if and only if it is a leaf in the representation $\mathcal{H}(G)$. Let the set $U \subset V$ be a leaf in the representation $\mathcal{H}(G)$, and let $(A, \overline{A})$ be any λ-cut such that $A \cap U$ is nonempty. Then $U \subseteq A$ since by the definition of $\mathcal{H}(G)$, any λ-cut that puts some vertex i from U in one of its sides must contain the entire set U in the same side. Hence the degree of a leaf in $\mathcal{H}(G)$ is λ (since $(U, \overline{U})$ is a λ-cut), and the degree of any subset of it is greater than λ since otherwise there will be another λ-cut that is completely contained in it, a contradiction to the above. Recall that the new edges that are added to increase the connectivity of G by one in algorithm **Aug-1** connect only vertices from G that are in some leaf in $\mathcal{H}(G)$. Hence for the case where $\delta = 1$, if the λ-extreme sets are augmented appropriately, then no other cuts have to be augmented. In general, suppose that U is k-extreme and $k < \lambda + \delta$. We can "expect" U to be a leaf in the representation of the augmented graph when the augmented connectivity is k since no subset of it is of degree k or less. This intuitive reasoning will be stated more precisely later.

4.3. The algorithm. The algorithm that optimally augments G by δ works in δ phases. At phase i, $1 \leq i \leq \delta$, a graph G^i is constructed, and its connectivity is $\lambda + i$. As we shall prove, every intermediate graph G^i is an optimal solution for the problem of increasing the connectivity of G by i. At each phase, we basically apply algorithm **Aug-1** (the algorithm that optimally augments by 1) to the graph G^{i-1} and obtain G^i. However, whenever there is more than one optimal way to augment G^i by one, we use the extreme-sets tree EST^i of the graph G^i to make the right choice among the various possibilities. Once EST^i is known, the rule for choosing the right solution is easy to implement.

The following is a high-level description of phases $1, 2, \ldots, \delta$ of the algorithm. Initially, we assume that $G^0 \equiv G$, and that EST^0, the extreme-sets tree for G, is known.

ALGORITHM **Aug-δ**

(1) Repeat steps (1)–(3.a) of algorithm **Aug-1** on the graph G^i, i.e., find its connectivity cuts representation $\mathcal{H}(G^i)$, traverse and enumerate its leaves in the DFS

manner described earlier, and form the pairs of leaves of $\mathcal{H}(G^i)$ that are to be connected by new edges.

(2) The following steps replace step (3.b) in the algorithm **Aug-1**:

(2.a) For each leaf U_a in $\mathcal{H}(G^i)$, find the node u_a in the tree EST^i that corresponds to the set U_a. (U_a is a $(\lambda+i)$-extreme set in G^i, so the node u_a must exist.) Find a node w_a of EST^i in the subtree of u_a, possibly u_a itself, with the following properties: (i) $\Phi(w_a) > 0$; (ii) $\Phi(z) = 0$ for every child z of w_a. (If u_a is a leaf in EST^i, then $w_a \equiv u_a$.)

(2.b) Construct the graph G^{i+1} by adding the following edges to G^i: for any pair (U_a, U_b) of leaves in $\mathcal{H}(G^i)$ that is formed in step (1), pick an arbitrary vertex from G which is a leaf in the subtree of w_a in EST^i and an arbitrary vertex from G which is a leaf in the subtree of w_b in EST^i and connect them by an edge.

(3) Compute the extreme-sets tree EST^{i+1} for the graph G^{i+1} by *updating* EST^i.

Handling the case of an odd number of leaves in $\mathcal{H}(G^i)$. If the number of leaves in $\mathcal{H}(G^i)$ is odd at some phase other than the last one (that is, $i < \delta$), then at the end of step (1), there will be some leaf U_a that participates in two different pairs (i.e., *two* new edges have to be attached to U_a). Note that $\Phi(u_a) \geq 2$ since this is not the very last stage of the algorithm. For this leaf, in step (2.a) we select *two* nodes, w_{a_1} and w_{a_2}, from the subtree of u_a and associate each one with a different pair. Then step (2.b) can be performed as before. w_{a_1} and w_{a_2} are selected as follows: First, find a node w_a in the subtree of u_a such that (i) $\Phi(w_a) \geq 2$ and (ii) $\Phi(z) < 2$ for every child z of w_a. (Since $i < \delta - 1$, $\Phi(u_a) \geq 2$, so this is always possible.) If w_a is a leaf, then define $w_{a_1} \equiv w_a$ and $w_{a_2} \equiv w_a$. Otherwise, let z_1 and z_2 be two children of w_a with the *largest* edge demand among w_a's children: for $j = 1, 2$, find a node w_{a_j} in the subtree of z_j, possibly z_j itself, such that (i) $\Phi(w_{a_j}) > 0$ and (ii) $\Phi(z) = 0$ for every child z of w_{a_j}. This is similar to the rule of (2.a). Note that only three cases are possible: (a) $\Phi(z_1) = 0$, $\Phi(z_2) = 0$; (b) $\Phi(z_1) = 1$, $\Phi(z_2) = 0$; (c) $\Phi(z_1) = 1$, $\Phi(z_2) = 1$. This fact will be important later.

4.4. Optimality of the algorithm.

THEOREM 4.3. *Starting with any graph G and edge connectivity λ, the connectivity of G^δ at the end of algorithm* **Aug-δ** *is $\lambda + \delta$.*

Proof. At each phase i of the algorithm, $1 \leq i \leq \delta$, the set of new edges that are added to G^{i-1} is a set of edges which could have been added by algorithm **Aug-1** when applied to G^{i-1}. Hence the connectivity of the resulting graph G^i has increased by one, so the connectivity of G^δ is $\lambda + \delta$. ∎

THEOREM 4.4. *Let $\Phi(ES^i)$ be the edge demand of the partition ES^i defined by the extreme-sets tree of G^i. The number of edges added by algorithm* **Aug-δ** *is exactly $\lceil \Phi(ES^0)/2 \rceil$.*

Proof. We will show in Theorem 4.9 that if l_i new edges are added at the ith phase of the algorithm, $i = 1, \ldots, \delta - 1$, then $\Phi(ES^i) = \Phi(ES^{i-1}) - 2l_i$. At the last phase, $l_\delta = \lceil \Phi(ES^{\delta-1})/2 \rceil$ since at most one endpoint (the last one) may be added without satisfying a demand. Applying the argument repeatedly, we get that the total number of edges that are added over all phases is $\lceil \Phi(ES^0)/2 \rceil$. ∎

The key to the optimality of algorithm **Aug-δ** is therefore Theorem 4.9, which states that the edge demands of successive extreme sets trees are decreased by the "right" amount, i.e., by twice the number of edges added at each phase. The correctness of this theorem follows from a sequence of lemmas that are also used for the derivation of the running time of the algorithm. We now state and prove these lemmas.

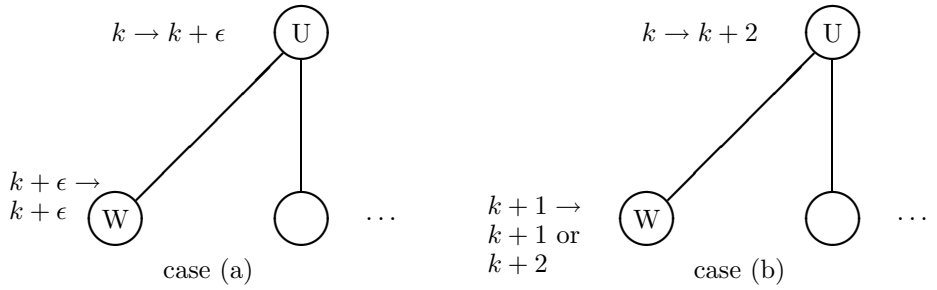


FIG. 5. *Lemma* 4.6; *W and U are represented as nodes in the tree.*

OBSERVATION 4.2. *Every nonextreme set $Z \subset V$ such that $d(Z) = k$ contains a k'-extreme set W, where $k' \leq k$.*

Proof. Z is nonextreme; therefore, it contains a subset $Z' \subset Z$ with $d(Z') \leq k$. If Z' is extreme, then the lemma follows ($W \equiv Z'$); otherwise, repeat this argument on Z'. The process terminates and eventually finds a subset of Z which is also an extreme set since a singleton is by definition an extreme set. □

OBSERVATION 4.3. *If W is an extreme set in G^{i-1}, then at the end of the ith phase of* **Aug-δ**, *the degree of W can increase by at most* 2.

Proof. First, suppose that the number of leaves in $\mathcal{H}(G^{i-1})$ is even. Every leaf in $\mathcal{H}(G^{i-1})$ gets one new endpoint attached to it, so the degree of every $(\lambda + i - 1)$-extreme set in G^{i-1} is increased by one by the end of the ith phase. If W is extreme, then by Lemma 4.2, it can be contained in at most one $(\lambda + i - 1)$-extreme set. If W is not contained in any $(\lambda + i - 1)$-extreme set, then its degree does not change; otherwise, its degree increases by one only if the endpoint of the new edge is also in W, and it remains unchanged otherwise. If the number of leaves in $\mathcal{H}(G^{i-1})$ is odd, then exactly one leaf gets two new endpoints attached to it; the same arguments follow in this case. □

The next two lemmas are cornerstones in the proof of optimality and the time analysis of the algorithm.

LEMMA 4.5. *If U is an extreme set in G^{i+1}, then it must have been an extreme set in G^i.*

LEMMA 4.6. *A set U is k-extreme in G^i but not extreme in G^{i+1} if and only if one of two cases* (*summarized in Figure* 5) *occurs:*

(a) *in G^i there is a $(k+\epsilon)$-extreme set W, $W \subset U$, $\epsilon = 1$ or 2, such that in G^{i+1} the degree of U is $k+\epsilon$, but the degree of W remains $k+\epsilon$.*

(b) *in G^i there is a $(k+1)$-extreme set W, $W \subset U$, such that in G^{i+1} the degree of U is $k+2$, but the degree of W either remains $k+1$ or changes to $k+2$.*

The proofs of these lemmas assume without loss of generality that $i = 0$, and we denote by G and G' the graphs G^0 and G^1, respectively. Also, for $X \subset V$, denote by $d(X)$ and $d'(X)$ the degrees of X in G and G', respectively.

Proof of Lemma 4.5. We show the contraposition of the lemma. That is, we show that if U is nonextreme in G, then U contains a subset $W \subset U$ such that in G', $d'(W) \leq d'(U)$; hence U must remain nonextreme in G'.

Clearly, only new edges that are added to G at the current step may change the degree of U relative to its subsets. Also, if all new edges that are attached to vertices in U are connected *across* U, i.e., their other endpoint is not in U, then the degree of any subset of U can increase by at most the degree increase of U; so U, which is nonextreme in G, remains nonextreme in G'. Therefore, we need consider only the

case where there is at least one new edge (x, y), $x, y \in U$.

Let S be the λ-extreme set that the edge (x, y) crosses such that $d'(S) = d(S)+1$. Recall that a new edge always connects two λ-extremes, and since the degrees of all but one λ-extreme are increased by one, such an S must exist. By the definition of S and U, $S \cap U \neq \emptyset$, and also $U \not\subseteq S$ (since (x, y) crosses S but not U). Hence either $S \subset U$ or S and U properly intersect.

1. $S \subset U$: this is an easy case since S is a subset of U with degree $d'(S) = \lambda + 1$, but $d'(U) \geq \lambda + 1$ (the connectivity of G' is $\lambda + 1$), so U is not extreme in G'.

2. S properly intersects U: again, there are two cases to consider.

2.1. If $S \cup U = V$, then $d'(U \cap \overline{S}) = d'(S) = \lambda + 1$; but $d'(U) \geq \lambda + 1$ and $(U \cap \overline{S}) \subset U$, so U is not extreme in G'.

2.2. If $S \cup U \subset V$, then V is divided into four nonempty quadrants $S \cap U$, $S \cap \overline{U}$, $\overline{S} \cap \overline{U}$, and $\overline{S} \cap U$. We number them as quadrants 1, 2, 3, and 4, respectively, and let d_{ij} be the number of edges between quadrant i and quadrant j, $1 < i, j < 4$ ($d_{ij} = d_{ji}$). See Figure 3, replacing X with S and Y with U.

Since S is an extreme set in G, $d(S \cap \overline{U}) > d(S)$, or

$$\begin{aligned} &d_{21} + d_{23} + d_{24} > d_{14} + d_{13} + d_{42} + d_{23} \\ &\quad \Longrightarrow d_{21} > d_{14} + d_{13}, \end{aligned}$$

which implies that

$$d_{21} > d_{14} - d_{13}$$

since all d_{ij}'s are nonnegative. Adding $(d_{24} + d_{34})$, we get

$$d_{21} + d_{24} + d_{34} > d_{14} - d_{13} + d_{24} + d_{34}$$

or

$$d_{21} + d_{24} + d_{13} + d_{34} > d_{14} + d_{24} + d_{34},$$

that is, $d(U) > d(\overline{S} \cap U)$. Hence $\overline{S} \cap U$ is a subset of U with degree (in G) strictly smaller than U. The only new edges that can make the degree of $\overline{S} \cap U$ greater or equal to the degree of U are the ones that cross $\overline{S} \cap U$ but not U, i.e., edges between $\overline{S} \cap U$ and $S \cap U$. But these edges also cross S, and we know that there is exactly one new edge that crosses S (since $d'(S) = d(S) + 1$). Therefore, there can be at most one new edge between $\overline{S} \cap U$ and $S \cap U$, so $d'(U) \geq d'(\overline{S} \cap U)$. This shows that U is not extreme in G'. □

Proof of Lemma 4.6. Let U be an extreme set in G which is no longer extreme in G' and $d(U) = k$; that is, there exists $Z \subset U$ with $d'(Z) \leq d'(U)$, but since U is extreme in G, $d(Z) > d(U)$. For that to happen, the degree of U must increase; but by Observation 4.3, it can increase by at most 2. Consider all possible cases.

$d'(U) = k + 1$: Since $d(Z) > k$ and $d'(Z) \leq k + 1$, $d(Z) = d'(Z) = k + 1$. ($d(Z)$ was not increased at that phase.) We now show that case (a) of the lemma with $\epsilon = 1$ must hold. If Z itself is a $(k + 1)$-extreme set in G, then we are done; otherwise, by Observation 4.2, Z contains an extreme set W with $d(W) \leq k + 1$. But W is also a subset of U, and U is k-extreme, so $d(W) = k + 1$, and the claim follows.

$d'(U) = k+2$: Since $d(Z) > k$ and $d'(Z) \leq k+2$, either $d(Z) = k+2$ and no new edges are attached to Z or $d(Z) = k + 1$ and zero or one new edges are attached to Z. We now show that either case (a) with $\epsilon = 2$ or case (b) of the lemma must hold.

If $d(Z) = k + 2$ and Z is extreme in G, then case (a) with $\epsilon = 2$ holds; if Z is not extreme, it contains an extreme set W with $d(W) \leq k + 2$. But U is k-extreme and W is also contained in U, so $d(W) = k + 1$ or $d(W) = k + 2$; since no new edges are attached to W, cases (a) or (b) hold. If $d(Z) = k + 1$, then case (b) holds since either Z is extreme or Z contains an extreme set W with $d(W) = k + 1$ in G; in both cases, the degree of the extreme set increases by 0 or 1. □

Constructing EST^{i+1} from EST^i. Lemmas 4.5 and 4.6 show how to obtain EST^{i+1} from EST^i. They show that no new nodes can be created (Lemma 4.5) and that the rule for removing nodes from EST^{i+1} is simple (Lemma 4.6). Given EST^i and the set of new edges that are added at step i, first compute the new degrees of all sets that are tree nodes; then traverse EST^i from the root downwards and look for any degree violation of the form described in Lemma 4.6, namely, either case (a) or (b) in Figure 5. If such a violation is detected at node u of the tree, then u is removed and its children are attached to its parent; this operation is called a *node removal*. The rest of the tree remains the same. Note that all we do is check the degrees of *nodes in the tree*, without considering other subsets of V, and this observation is what makes the (dynamic) computation of successive extreme-sets trees much more efficient than recomputing a new tree at each iteration.

Relating $\Phi(ES^i)$ to $\Phi(ES^{i-1})$. Let T and T' be the extreme-sets trees of G^{i-1} and G^i, respectively, where r and r' are their corresponding roots. Also, for every set $X \subset V$, $d(X)$ and $d'(X)$ are the degrees of X in G^{i-1} and G^i respectively. For a node x in a tree, $\Phi(x)$ is the edge demand of x in G^{i-1} and $\Phi'(x)$ is the edge demand of x in G^i. First, assume that $\Phi(ES^0)$ is even so that *every* new edge satisfies a demand of some extreme set. We will handle the odd case later.

LEMMA 4.7. *If x is a node in T other than the root and ϵ new edges are attached to vertices in X ($\epsilon = 0, 1, 2$), then*

$$\Phi'(x) = \begin{cases} 0 & \text{if } \Phi(x) = 0, \\ \Phi(x) - \epsilon & \text{if } \Phi(x) > 0. \end{cases}$$

Proof. For any node x other than the root, if a new edge is attached to some vertex in X, then its other endpoint is not in X. This is true since a new edge always connects two vertices that belong to two different leaves in $\mathcal{H}(G^{i-1})$, and all the vertices in X belong in the same leaf in $\mathcal{H}(G^{i-1})$. Hence the quantity $\lambda + \delta - d(X)$ decreases by ϵ. This fact is used below.

The lemma is proved by induction on the distance of x from the bottom of the tree. For a leaf x, if $\epsilon = 0, 1$, then the lemma follows immediately since $\Phi(x) = \max\{0, \lambda + \delta - d(X)\}$; if $\epsilon = 2$ for a leaf x, then $\Phi(x) \geq 2$, so the lemma also follows.

For an internal node x, let $y_1, y_2, \ldots$ be its children, so $\Phi(x) = \max\{0, \lambda + \delta - d(X), \sum_i \Phi(y_i)\}$. Clearly, if $\Phi(x) = 0$, then $\Phi'(x) = 0$; otherwise, $\Phi(x) = \max\{\lambda + \delta - d(X), \sum_i \Phi(y_i)\}$.

First, consider the case where only one edge is attached to X ($\epsilon = 1$). If the new edge is attached to a vertex in y_j for some j and $\Phi(y_j) = 0$, then $\sum_i \Phi(y_i) = 0$ (since by the rule of step (2.a), if $\Phi(y_j) = 0$ and $\epsilon = 1$, then the demand of all of y_j's siblings must also be zero). Hence $\Phi(x) = \lambda + \delta - d(X)$ and clearly $\Phi'(x) = \Phi(x) - \epsilon$. Otherwise, if $\Phi(y_j) > 0$, then by the induction hypothesis $\Phi'(y_j) = \Phi(y_j) - \epsilon$, so $\sum_i \Phi'(y_i) = \sum_i \Phi(y_i) - \epsilon$. Combining the two cases, we get that $\Phi'(x) = \max\{\lambda + \delta - d'(X), \sum_i \Phi'(y_i)\} = \Phi(x) - \epsilon$.

Now suppose that *two* edges are attached to X ($\epsilon = 2$). This will happen only if $\Phi(x) \geq 2$. If both edges were attached to vertices in y_j for some j, then it must be that

$\Phi(y_j) \geq 2$, so by induction $\Phi'(x) = \max\{\lambda + \delta - d'(X), \sum_i \Phi'(y_i)\}$. If the edges are attached one to a vertex in y_j and the other to a vertex in y_k for some $j \neq k$ so that $\Phi(y_j) = 1$ but $\Phi(y_k) = 0$ or $\Phi(y_j) = \Phi(y_k) = 0$, then $\sum_{i \neq j,k} \Phi(y_i) = 0$. (Refer to the paragraph following step (3) of algorithm **Aug-δ**.) Therefore, $\Phi(x) = \lambda + \delta - d(X)$, so $\sum_i \Phi'(y_i) = 0$ and $\Phi'(x) = \{\lambda + \delta - d'(X)\} = \Phi(x) - \epsilon$. If $\Phi(y_j) + \Phi(y_k) \geq 2$, then the lemma follows inductively. □

If $\Phi(ES^0)$ is even, then every node x which is an immediate child of the root that gets a new edge attached to it must be a leaf in the cactus. The demand of a cactus leaf is always at least one; moreover, if $\Phi(ES^0)$ is even and two edges are attached to some cactus leaf, then it must be that its demand is at least two. (In fact, the only case where two edges are attached to a cactus leaf but its demand is one is when $\Phi(ES^0)$ is odd and the very last edge is attached.) Hence $\Phi(x) \geq \epsilon$ for every node x which is an immediate child of the root. Therefore, a corollary of Lemma 4.7 is that $\Phi'(r) = \Phi(r) - 2l_i$, where l_i is the number of new edges added by the algorithm at the ith phase. The edge demand of the root is defined as the sum of the demands of its children, and every new endpoint reduces the demand of some child by one; the total decrease in the demand of T is therefore $2l_i$. We now show that the total decrease in the demand of T' is also $2l_i$.

LEMMA 4.8. $\Phi'(r') = \Phi'(r)$.

Proof. T' is obtained from T by a sequence of node-removal operations. Let $\alpha(x) = (\lambda + \delta) - d'(X)$. By Lemma 4.6, x is in T but not in T' if it has a child w (in T) such that $d'(W) \leq d'(X)$, so $\alpha(w) \geq \alpha(x)$. Hence

$$\sum_{y \text{ child of } x \text{ in } T} \Phi'(y) \geq \sum_{y \text{ child of } x \text{ in } T} \alpha(y) \geq \alpha(w) \geq \alpha(x).$$

Since $\Phi'(x) = \max\{\alpha(x), \sum_{y \text{ child of } x} \Phi'(y)\}$ and $\sum_{y \text{ child of } x} \Phi'(y) \geq \alpha(x)$, we get that $\Phi'(x) = \sum_{y \text{ child of } x} \Phi'(y)$. Hence the removal of x from T does not change the edge demand of its parent since x's children are attached to its parent. In general, node removals do not affect the edge demand of any of the remaining nodes, including the root, so $\Phi'(r) = \Phi'(r')$. □

THEOREM 4.9. *Assume that either $\Phi(ES^0)$ is even or we are at any phase $i < \delta$. If $\Phi(ES^i)$ is the edge demand of the partition ES^i defined by the extreme-sets tree of G^i and l_i new edges are added at the ith phase of algorithm* **Aug-δ**, *then $\Phi(ES^i) = \Phi(ES^{i-1}) - 2l_i$.*

Proof. Lemma 4.7 implies that $\Phi'(r) = \Phi(r) - 2l_i$, and by Lemma 4.8, $\Phi'(r) = \Phi'(r')$, so $\Phi'(r') = \Phi(r) - 2l_i$. Since the edge demand of the partition defined by a tree equals the edge demand of its root, the theorem follows. □

Now assume that $\Phi(ES^0)$ is odd. In this case, the very last edge that is added at phase $i = \delta$ does not satisfy a demand of a leaf in the cactus. Hence at that phase, $\Phi(ES^i) = 0$, and $\Phi(ES^{i-1}) + 1 = 2l_i$, so over all phases we have $\Phi(ES^0) + 1 = \sum_i 2l_i$.

4.5. Time analysis of algorithm Aug-δ. Algorithm **Aug-δ** repeatedly applies algorithm **Aug-1** to the graph and maintains the correct extreme-sets tree at each phase. The major issue is therefore the computation of the tree in each phase. In section 4.4, we showed that EST^{i+1} can be obtained from EST^i by a sequence of node-removal operations. A node that needs to be removed is identified by a simple criteria based on the degrees of its children in the tree, namely that the minimum degree of its children is greater than its own degree. Hence EST^{i+1} can be computed from EST^i by two traversals of the tree: the first, a bottom-up traversal, updates the new degrees of the tree nodes, and the minimum degree of the children is propagated

upwards; then in a top-down fashion, if a node is to be removed, its children are attached to its parent. Note that the new degrees need not be recomputed from the graph but only updated, i.e., the addition of an edge (u, v) increases the degree of u and v by one. This information is propagated towards the upper nodes during the tree traversal, so it takes time linear in the size of the tree. The top-down traversal may redirect every edge in the tree exactly once; hence it also takes time linear in the size of the tree. In summary, EST^{i+1} can be obtained from EST^i in linear time (the size of the tree is $O(n)$).

We now show how to compute EST^0. The definition of extreme sets does not lend itself to an efficient computation since it requires the inspection of the degrees of all of its subsets. Fortunately, it turns out that extreme sets are highly structured and closely related to another type of sets, maximal-edge-connected components, which can be found relatively easily.

Define $C(i, j)$ as the minimum cut value between vertices i and j in G. We say that a set U is a *maximal l-edge-connected component* if and only if (i) $l = \min_{i,j \in U} C(i, j)$ and (ii) for any $v \notin U$, $i \in U$, $C(i, v) < l$.

LEMMA 4.10. *Every k-extreme set U ($d(U) = k$) is a maximal l-edge-connected component for some l. (There is no relation between l and k.)*

Proof. Let U be a k-extreme set, let $v \notin U$ and $i \in U$, and define $l = \min_{i,j \in U} C(i, j)$. If $l > k$, then since $(U, \overline{U})$ is a cut that separates v from i for any $v \notin U$ and $i \in U$ and since $d(U) = k < l$, $C(i, v) < l$, so U is a maximal l-edge-connected component. The more difficult case is when $l \leq k$. Define $(X, \overline{X})$ to be some cut of size l that separates two vertices $i, j \in U$. (Such a cut exists since $l = \min_{i,j \in U} C(i, j)$.) Since $X \not\subset U$ (otherwise, U is not k-extreme) yet X splits U, U and X must properly intersect. Also, $X \cup U \neq V$ since otherwise $d(U \cap \overline{X}) = d(X) = l \leq k$, $(U \cap \overline{X}) \subset U$, which contradicts our assumption that U is not k-extreme.

In summary, X and U must properly intersect and their intersection divides V into four nonempty quadrants, $X \cap U$, $X \cap \overline{U}$, $\overline{X} \cap \overline{U}$, and $\overline{X} \cap U$, numbered 1, 2, 3, and 4 respectively. As before, let d_{ij}, $1 \leq i, j \leq 4$, be the number of edges between quadrants i and j. We will now show that $d(\overline{U} \cap X) < l$ and that $d(\overline{U} \cap \overline{X}) < l$. Hence for any $v \notin U$ (v is either in $(\overline{U} \cap X)$ or in $(\overline{U} \cap \overline{X})$), there exists a cut of size $< l$ that separates it from any $i \in U$, so the claim follows.

Note that since U is k-extreme, we have $d(U \cap X) > k$,

$$\begin{aligned} d_{12} + d_{13} + d_{14} > k = d_{12} + d_{13} + d_{42} + d_{43} \\ \Longrightarrow d_{14} > d_{42} + d_{43} \\ \Longrightarrow d_{14} - d_{42} > d_{43} \\ \Longrightarrow d_{14} + d_{42} \geq d_{14} - d_{42} > d_{43}, \end{aligned} \tag{3}$$

and also $d(U \cap \overline{X}) > k$,

$$\begin{aligned} d_{14} + d_{24} + d_{34} > k = d_{12} + d_{13} + d_{42} + d_{43} \\ \Longrightarrow d_{14} > d_{12} + d_{13} \\ \Longrightarrow d_{14} - d_{13} > d_{12} \\ \Longrightarrow d_{14} + d_{13} \geq d_{14} - d_{13} > d_{12}. \end{aligned} \tag{4}$$

Adding $d_{31} + d_{32}$ to both sides of equation (3), we get

$$d(\overline{U} \cap \overline{X}) = d_{34} + d_{31} + d_{32} < d_{14} + d_{42} + d_{31} + d_{32} = d(X) = l,$$

that is, $d(\overline{U} \cap \overline{X}) < l$. Adding $d_{23} + d_{24}$ to equation (4), we get

$$d(\overline{U} \cap X) = d_{12} + d_{23} + d_{24} < d_{14} + d_{13} + d_{23} + d_{24} = d(X) = l,$$

that is, $d(\overline{U} \cap X) < l$. □

A subset U is simply called a *maximal component* if it is a maximal l-edge-connected component for some l. Note that while we have shown that all extreme sets are maximal components, it is *not* the case that all maximal components are extreme sets. Lemma 4.10 is the key to efficiently computing the initial extreme-sets tree EST^0 since maximal components are relatively easy to find. Maximal components form a hierarchical structure that can be represented by a tree, the *MCC tree*, since any two maximal components are either disjoint or contained in one another. This hierarchical structure is implied, for example, by the famous result of Gomory and Hu [13], as explained in the next paragraph. *MCC* is a rooted tree whose leaves are the vertices of G. Each node u in the tree represents a subset $U \subseteq V$ which consists of the leaves in the subtree of the u. u is a node in *MCC* if and only if U is a maximal l-edge-connected component for some l.

One can efficiently compute the *MCC* tree from an equivalent-flow tree of G. An *equivalent-flow tree* is a weighted tree whose vertex set is V, and for any $i, j \in V$, the minimum weight edge on the path from i to j in the tree is the value of the minimum cut between i and j in G. An equivalent-flow tree can be constructed with $n - 1$ max-flow computations [13, 18]. The next procedure finds the *MCC* tree of G from an equivalent-flow tree of G. As it builds *MCC* it keeps at each node u of the tree built so far a *partial* equivalent flow tree T_u (the equivalent flow tree of the vertices in U).

- Initially, *MCC* contains a single node r (root) that corresponds to V, the vertex set of G. V is a λ-maximal edge-connected component. Associate with this node the equivalent-flow tree of the entire graph, T_r.
- Let u be a node in *MCC* that contains more than one vertex from V, $U \subseteq V$ be the set of vertices in u, and T_u be the equivalent-flow tree associated with U. Remove from T_u all edges with the smallest weight, and let $T_{u_1}, T_{u_2}, \ldots$ be the subtrees created as a result of this operation. $U_i \subset U$ is the set of vertices in T_{u_i}. For each subset U_i, create a new tree node u_i that contains vertices U_i, and make it a child of u in *MCC*. U_i is now a k-maximal edge-connected component, where k is the smallest edge weight in the subtree T_{u_i}, and the equivalent-flow tree associated with u_i is T_{u_i}.
- Repeat this process until all tree nodes contain singletons (which are the leaves of *MCC*).

By Lemma 4.10, if U is extreme, then it must be a maximal edge-connected component and hence must appear as a node u in *MCC*. By Observation 4.2, if U is a maximal k-edge-connected component but is *not* extreme, then it must contain an *extreme* set W such that $d(W) \leq d(U)$, so W is also a maximal k'-edge-connected component for some $k' > k$. Therefore, W must appear as a node in the subtree of u in the *MCC* tree and indicates that u and its corresponding set U is not extreme. Note that every leaf in *MCC* is a singleton and hence an extreme set.

In summary, EST^0 is a partial tree of *MCC*, i.e., some of *MCC*'s internal nodes are missing from it. To find and remove these internal nodes, compute the degrees $d(W)$ of all nodes w in *MCC*. Then in a bottom-up fashion, determine for each *internal* node u whether all nodes in its subtree are of degrees greater than $d(U)$; if not, then remove u from the tree and attach its children to u's parent. The resulting tree is EST^0. Since the computation of *MCC* is dominated by the $n - 1$ maximum-flow computations required to construct the equivalent-flow tree and since EST^0 can

be constructed from *MCC* in $O(nm)$ time (the degrees of the tree nodes must be computed), we get the following result.

COROLLARY 4.11. *Let G be a simple graph. The extreme-sets tree EST^0 of G can be computed in $O(nF(G))$ time, where $F(G)$ is the time to compute a single maximum flow in G. Since G is undirected and unweighted, the time to compute EST^0 is $O(m^{3/2}n)$* [8, 2].

The next lemma is an extension of the edge connectivity algorithm of [31] to multigraphs. It also implies that the edge connectivity of an undirected, *edge-weighted* graph is computable in $O(Pm)$ time, where $P = \sum_{v \in V} p(v)$ and $p(v)$ is the maximum capacity of all edges incident at v, provided that the weights are integers. Note that in a simple graph, $P = n$, yielding the original result of [31]. The proof of the lemma is included in Appendix A since it is not directly relevant to the paper.

LEMMA 4.12. *For vertices u and v, let $c(u, v)$ be the number of edges between u and v, and $p(v) = \max_{u \in V}\{c(u, v)\}$. Define $P = \sum_{v \in V} p(v)$. Let m be the number of pairs of vertices which are connected by some edge in G. The edge connectivity of G and its representation $\mathcal{H}(G)$ can be computed in time $O(Pm)$.*

THEOREM 4.13. *For any $\delta > 0$, algorithm* **Aug-δ** *optimally solves the edge augmentation problem in time $O(\delta^2 nm + \delta^3 n^2 + n^{5/3}m)$.*

Proof. Let P', p', and m' be the corresponding values of P, p, and m at the current phase of the algorithm. If the initial graph G is simple, then at the end of algorithm **Aug-δ**, $p'(v) \leq \delta + 1$ for all v. This is because at each iteration, algorithm **Aug-1** adds at most one new edge between a fixed pair of vertices (that is, for some pair u, v, **Aug-1** will never add two or more edges between them). Hence at the end of the algorithm, $P' = \sum_{v \in V} p'(v) \leq n(\delta+1) \leq 2\delta n$. Also, the algorithm adds at most $\delta n/2$ new edges throughout the phases (at most $n/2$ at each phase), so at the end of the algorithm, $m' \leq m + \delta n/2$. By Lemma 4.12, the construction of $\mathcal{H}(G^i)$ at each phase is bounded by $O(P'm') = O(\delta nm + \delta^2 n^2)$, yielding $O(\delta P'm') = O(\delta^2 nm + \delta^3 n^2)$ overall. Computing the initial extreme-sets tree EST^0 requires $O(nF(G))$ time, and EST^i can be computed from EST^{i-1} in $O(n)$ time. Hence we get $O(nF(G)) = O(n^{5/3}m)$ for the initial computation, plus $O(\delta^2 nm + \delta^3 n^2)$ for the computations throughout the phases. ∎

4.6. Relation between algorithm Aug-δ and Watanabe and Nakamura's algorithm. In the presentation of the algorithm in this paper, we have discussed some of its similarities to the algorithm of Watanabe and Nakamura (as presented in [36, 35]). In this section, we elaborate on the similarities and differences between the two approaches.

(i) Watanabe and Nakamura [36] provided the first polynomial algorithm for the general augmentation problem. This algorithm adds an edge at a time and tests (according to a well-defined criteria) that this edge can be extended to an optimal solution. Later, in his technical report, Watanabe [35] proved that the algorithm can be sped up by finding at once a *set* of edges which optimally increase the connectivity by one and which can be extended further.

The main idea employed by our algorithm **Aug-δ** is to extend the simple and efficient approach developed in algorithm **Aug-1** (section 3) in order to solve the general problem of augmentation by δ. Section 4 shows that the idea can be implemented correctly and efficiently.

Hence both approaches find successive optimal augmentations by one until they reach the desired connectivity. However, they differ in the method by which they select an optimal augmentation by one that can be extended further. Moreover, we were

specifically interested in those augmentations by one that can be found by algorithm **Aug-1** via the use of the cactus graph and the DFS enumeration.

(ii) The hierarchical structure of maximal connected components, which we call *MCC* in the paper, is used in both algorithms. However, the *MCC* is *a major component* in the algorithm of Watanabe and Nakamura, whereas in our algorithm it is used only as an auxiliary tool to efficiently construct the main data structure in our algorithm, the *EST* tree (in fact, only to construct EST^0).

Specifically, extreme sets, and the extreme-sets tree, are key objects in our algorithm. (Note that they are not defined or used in [36, 35].) They can be intuitively related to the notion of edge demand, and our algorithm uses them extensively to guide the selection at each phase of our algorithm. Unfortunately, the definition of extreme sets does not lend itself to a practical implementation since it requires testing the degrees of all subsets of a given set. By observing the relationship between extreme sets and maximal connected components (section 4.5), we were able to use the *MCC* tree in order to devise an efficient method to construct the first extreme sets tree EST^0. This relationship also implies that various properties of extreme sets (for example, the one proved in Lemma 4.2) can also be proved indirectly via maximal connected components.

(iii) Finally, at a high level, our proof method resembles the proof method of [36]. Both follow the following steps: (1) use the lower bound observation to derive a lower bound that is implied by some partition; in our case, we use the extreme-sets partition. This is also the first step in the proofs of [9, 5]. For that we use the definition of $\Phi(x)$ which was introduced in [36]. (2) show that the addition of x new edges to the graph decreases this lower bound by x. (3) show that at the end of the algorithm, the connectivity has gone up by the desired amount.

5. Variations on the edge augmentation problem.

5.1. The no-target problem. Suppose that the target connectivity is not known or given in advance; instead, the goal is to augment the graph, possibly several times, so that for any intermediate connectivity, the augmentation is optimal. Our solution can be adapted to solve the augmentation problem even if the target increment δ is not known in advance by introducing a simple modification to step (2.a) of algorithm **Aug-δ**. In both cases, the algorithm first finds the node u_a that corresponds to the set U_a which is a leaf in $\mathcal{H}(G^i)$ and then selects a vertex in U_a to be the endpoint of the new edge. The algorithms differ in the criteria by which this vertex is selected. If the target δ is known in advance, then the vertex in U_a is selected according to criteria $\mathcal{A}$ as follows:

> Start at node u_a. As long as the edge demand of the current node is positive and the current node is not a leaf, pick a child whose *edge demand* is positive and make it the current node. If no such child exists and the current node is not a leaf yet, pick an arbitrary leaf in its subtree.

If the target δ is *not* known, then the vertex in U_a is selected according to criteria $\mathcal{B}$ as follows:

> Start at node u_a. As long as the current node is not a leaf, pick the child with the smallest *degree* and make it the current node.

OBSERVATION 5.1. *Let G be a graph with connectivity λ. For any $\delta \geq 1$, if $K \equiv \lambda + \delta$ and u is a node in the extreme-sets tree, then $\Phi(u) > 0$ if and only if $d(U) < K$.*

Proof. The proof is by induction on the height of the subtree rooted at u. For a leaf u, the claim follows immediately since $\Phi(u) = K - d(U)$. For an internal node u, if $d(U) < K$, then clearly $\Phi(u) \geq K - d(U) > 0$. Now suppose that $\Phi(u) > 0$ but $d(U) \geq K$: for any child w of u, $d(W) \geq d(U) \geq K$ since W is a subset of U and U is an extreme set. Hence by the induction hypothesis, $\Phi(w) = 0$, so $\Phi(u) = \max\{0, K - d(U), \sum_w \Phi(w)\} = 0$, a contradiction. □

THEOREM 5.1. *Criteria $\mathcal{B}$ correctly modifies algorithm* **Aug-δ** *to solve the edge augmentation problem when δ is not known.*

Proof. Suppose that the algorithm is run using criteria $\mathcal{B}$, stops when a connectivity $K = \lambda + \delta$ is reached (for any δ), and adds a set of new edges E'. We show that the algorithm that uses criteria $\mathcal{A}$ and δ as a target could have picked exactly the same set of new edges E'. Since the latter was shown to be correct and optimal, the modified algorithm is also correct and optimal for *any* connectivity value it reaches.

Starting at u_a, follow the path chosen according to criteria $\mathcal{B}$ without knowing δ. Let u be the current node along this path and $w_1, w_2, \ldots$ be its children. Suppose that w_j is the child with the smallest degree among u's children and therefore is picked to be the next node on the path. If $d(w_j) \geq K$, then $d(w_i) \geq K$ for any w_i, so by Observation 5.1, $\Phi(w_i) = 0$ for all i. In that case, according to criteria $\mathcal{A}$, *any* leaf in the subtree of u can be picked; hence the (entire) remaining path chosen by criteria $\mathcal{B}$ could have been chosen by criteria $\mathcal{A}$. If $d(w_j) < K$, then $\Phi(w_j) > 0$; but according to criteria $\mathcal{A}$, any child whose edge demand is positive may be picked, so the choice of w_j is valid in both cases. Hence the entire path could have been picked according to criteria $\mathcal{A}$, so the endpoint of the new edge is properly chosen. This argument applies for any new edge in E'. □

The converse problem. The solution to the problem of increasing the connectivity of G as much as possible by adding at most k edges is now immediate. Apply the modified version of **Aug-δ** using criteria $\mathcal{B}$ (with no δ in mind) until k new edges are used. The connectivity at that point is the solution to the converse problem.

5.2. The node-weighted problem. Frank [9] first formulated and solved the following variant of the edge augmentation problem. Suppose that there is a nonnegative cost $c(v)$ associated with every vertex $v \in V$ and that the cost of adding a new edge (x, y) to G is $c(x) + c(y)$. The goal is to find the minimum-cost set of edges to add to G in order to make it $(\lambda + \delta)$-edge-connected. The solution in [9] was proved using polymatroid theory, and it extends to more general instances of the problem. We now show that algorithm **Aug-δ** can be adapted in a very natural way to optimally solve the node-weighted version of the edge augmentation problem for undirected graphs. Since the edge demand $\Phi(X)$ of any subset of vertices $X \subset V$ must be satisfied, $\lceil \Phi(ES)/2 \rceil$, the lower bound defined by the extreme-sets partition on the *number* of edges that need to be added to G, still holds. We establish a lower bound on the *cost* of the solution to the node-weighted problem and then show how to find a set of edges whose cost equals this lower bound. It will be shown that the minimum node-weighted solution is also optimal with respect to the *number* of edges it adds, i.e., it is the minimum node-weighted solution among all possible solutions that add exactly $\lceil \Phi(ES)/2 \rceil$ new edges.

Let x be a node in the extreme-sets tree, $X \subset V$ be the set of vertices in G that are leaves in the subtree of x, and $\Phi(x)$ be its edge demand. Recursively define $\Psi(x)$, the min-cost edge demand of x, as

$$\Psi(x) = \sum_{y \text{ child of } x} \Psi(y) + \max\left\{0, \left(\lambda + \delta - d(X) - \sum_{y \text{ child of } x} \Phi(y)\right)\right\} C_{\min}(X),$$

where $C_{\min}(X) = \min_{v \in X}\{c(v)\}$ (the minimum node cost among all vertices in X). The min-cost edge demand of the root r is defined as $\Psi(r) = \sum_{y \text{ child of } r} \Psi(y)$. Intuitively, $\Psi(x)$ is a lower bound on the cost of the edges that are "missing" from X: every subset Y of X that requires $\Phi(y)$ edges contributes at least $\Psi(y)$ to the total cost; the rest of the missing edges from X, $\max\{0, (\lambda+\delta-d(X)) - \sum_{y \text{ child of } x} \Phi(y)\}$, may be connected to *any* vertex in X, so they contribute at least $C_{\min}(X)$ per edge. Note that if $c(v) = \alpha$ for every v, then $\Psi(r) = \alpha\Phi(r)$.

Clearly $\Psi(ES)$ is a lower bound on the cost of the optimal solution, where ES is the extreme-sets partition of G. This lower bound can be achieved if the following criteria, criteria $\mathcal{C}$, is employed instead of the original criteria $\mathcal{A}$ in step (2.a) of Algorithm **Aug-δ**:

> Start at node u_a in the tree. As long as the edge demand of the current node is positive and the current node is not a leaf, pick a child whose *edge demand* is *positive*, and make it the current node; otherwise, pick the *minimum-cost* leaf in the subtree of the current node.

Since criteria $\mathcal{C}$ is a restricted version of criteria $\mathcal{A}$, the solution obtained by running **Aug-δ** with criteria $\mathcal{C}$ is also optimal with respect to the number of new edges it adds.

THEOREM 5.2. *Criteria $\mathcal{C}$ correctly modifies algorithm* **Aug-δ** *to solve the node-weighted version of the edge-augmentation problem.*

Proof. We have to show that at each phase $\Psi(ES^i)$ drops by the total cost of the new edges added in that stage. The proof is essentially similar to the one given in section 4.4. The analogue of Lemma 4.7 is that if $c(\epsilon_x)$ is the total cost of the new endpoints attached to X, then $\Psi'(x) = \Psi(x) - c(\epsilon_x)$; to prove the analogue lemma, one needs to observe that in the bottom of the induction, if x is a leaf in T, then $c(\epsilon_x) = \epsilon c(x)$, where ϵ is the *number* of new endpoints attaches to X. The analogue of Lemma 4.8 replaces Φ with Ψ, and the proof is similar. The corollary is that the total cost of the solution is $\Psi(ES)$, and hence it is the minimum-cost solution. ∎

5.3. The degree-constrained problem. We now show that our algorithm can be extended in a natural way to solve another variant of the edge augmentation problem which was first formulated and solved by Frank [9]. Suppose that there are constraints $g(v)$ and $f(v)$ associated with the degree of every vertex $v \in V$. The *degree-constrained augmentation problem* is to add the smallest number of edges to the graph and increase its connectivity by δ, subject to the constraints that the degree of v in the resulting graph does not exceed $g(v)$ and is not below $f(v)$. There is not always a solution to this problem. Frank gave a necessary and sufficient condition for this problem and an $O(n^5)$-time algorithm to solve it. We consider a restricted case of this problem, where only the $g(v)$ is specified, i.e., $f(v) = 0$ for all v, and give a necessary and sufficient condition for the problem to be solvable, expressed in terms of our algorithm. If the condition is met, then a minor addition to algorithm **Aug-δ** produces a solution to the degree-constrained version. The running time of the modified algorithm remains $O(\delta^2 nm + \delta^3 n^2)$. Recall the extreme-sets partition ES defined by the extreme-sets tree EST of G. Clearly, $\lceil \Phi(ES)/2 \rceil$ is a lower bound on the number of edges required to solve the problem.

THEOREM 5.3. *Let $\Phi(x)$ be the edge demand of x, and let $x_1, \ldots, x_r$ be the immediate children of the root of the EST tree. The degree-constrained augmentation problem is solvable if and only if*

1. *for every node x in EST,*

$$\sum_{v \in X} g(v) - d(v) \geq \Phi(x),$$

2. *if $\Phi(ES)$ is odd, then*

$$\sum_{v \in S} g(v) - d(v) \geq \Phi(ES) + 1,$$

where $S = \{v \in X_i, i = 1, \ldots, r | \Phi(x_i) > 0\}$.

Moreover, if the problem is solvable, then the optimal solution adds $\lceil \Phi(ES)/2 \rceil$ new edges.

Proof. If for some $X \subset V$, $\sum_{v \in X}(g(v) - d(v)) < \Phi(x)$, then no solution can satisfy the edge demand of X under the degree constraints, so the necessity of this condition is obvious. Also, if $\Phi(ES)$ is odd, then $\lceil \Phi(ES)/2 \rceil$ edges must be added, so the total degree increment (on all vertices) must be at least $\Phi(ES) + 1$. Since $\Phi(ES) = \sum_{i=1}^{r} \Phi(x_i)$, we get that the allowed amount of degree increment among vertices in $x_1, \ldots, x_r$ must be at least $\Phi(ES) + 1$.

Suppose that condition 1 above initially holds for every node x in EST^0, and consider the first phase of algorithm **Aug-δ**. First, assume that either $\Phi(ES)$ is even or $\Phi(ES)$ is odd and we are not at the last phase of the algorithm. When applying the algorithm, in step (2.a), start at u_a; then repeat following a child whose edge demand is positive until no such child exists. Let x be the current node. Note that $\Phi(x) > 0$ since the addition of every edge, other than the last edge when $\Phi(ES)$ is odd, always satisfies some demand. If x is a leaf, simply pick it to be the endpoint of the new edge; otherwise, instead of picking an arbitrary leaf from the subtree of x, choose a leaf v with $g(v) > d(v)$.

Observe that if x is a leaf, then $g(x) - d(x) \geq \Phi(x) > 0$, so $g(x) > d(x)$, and a new edge can be attached to x. If x is not a leaf, then $\sum_{v \in X}(g(v) - d(v)) \geq \Phi(x) > 0$, so there must be some $v \in X$ with $g(v) > d(v)$, and a new edge can be attached to v. Hence if condition 1 above holds, then the current phase of the (modified) algorithm can be applied without violating the degree constraints. It now needs to be shown that at the end of the current phase, conditions 1 and 2 still hold, i.e., for every node x in the new tree, if $d'(x) = d(x) + \epsilon$ is its new degree, then $\sum_{v \in X}(g(v) - d'(v)) \geq \Phi'(x)$, and also $\sum_{v \in S} g(v) - d'(v) \geq \Phi'(ES) + 1$ if $\Phi(ES)$ is odd. This is true since by Lemma 4.7, if $\Phi(x) > 0$, then

$$\Phi'(x) = \Phi(x) - \epsilon \leq \left(\sum_{v \in X} g(v) - d(v) \right) - \epsilon = \sum_{v \in X} g(v) - d'(v),$$

which also implies that

$$\Phi'(ES) + 1 = \sum_{i=1}^{r} \Phi'(x_i) + 1 \leq \sum_{v \in S} g(v) - d'(v).$$

Hence if the condition holds initially, then by applying algorithm **Aug-δ** (with the minor addition), the necessary condition is maintained throughout, so the algorithm terminates.

Now consider the last phase of the algorithm when $\Phi(ES)$ is odd. Without loss of generality, let x_i be the root's child for which $\sum_{v \in X_i} g(v) - d(v) \geq \Phi(x_i) + 1$ (condition

2 assures that such an x_i exists). We add the following detail to the last phase of the algorithm: In step (1) of algorithm **Aug-δ**, enumerate the leaves of the cactus by assigning the number 1 to x_i. (x_i is necessarily a leaf in the cactus since it is a child of the root at the last stage.) In addition, attach the last endpoint of the last edge to some vertex $v^* \in X_i$ for which $g(v^*) > d(v^*)$. Note that (i) there must be such $v^* \in X_i$ and (ii) the last endpoint of the last edge can be attached to a vertex in any leaf of the cactus *other than* to a vertex in the $\lceil k/2 \rceil$'s leaf, where $k \geq 2$ is the number of leaves in the cactus. Since v^* is in the first leaf, this requirement is satisfied. Hence condition 2 of the theorem guarantees that the last phase of the algorithm can be completed properly.

The modified algorithm adds $\lceil \Phi(ES)/2 \rceil$ new edges, since it produces a solution which could have been chosen for the unconstrained problem. But since this is also a lower bound on the number of edges needed, the solution is optimal. □

6. Open problems.

The augmentation problem with no parallel edges. No polynomial solution is known for the edge augmentation problem if parallel edges are *not* allowed. (Our algorithm, as well as all previous algorithms, may add parallel edges.) The state of this problem is an interesting question.

Clearly, any λ-edge-connected graph with $\lambda > n - 1$ must have parallel edges. At the other extreme, an unconnected graph can be made 1-edge-connected without adding parallel edges by linking its connected components to a tree of components. Hence if the target connectivity is either 1 or greater than $n - 1$, then the answer is known. For the intermediate values, the following preliminary observations can be made.

If the graph G is either 1- or 2-edge-connected, then many of the known algorithms, such as those of [7, 36, 37], can augment it to be 3-edge-connected by selecting nonparallel edges. This is also true for our solution: if $\lambda = 1, 2$, then any pair of leaves in the cactus which are singletons that are connected by an edge in the graph must also be adjacent in the DFS enumeration; but our algorithm adds edges between only nonconsecutive leaves.

However, this is not true in general, as shown by the following sequence of graphs. For any $3 \leq \lambda \leq n - 1$, define $G_\lambda = (V_\lambda, E_\lambda)$ as follows:

$$V_\lambda = \{a_1, \ldots, a_\lambda, b_1, \ldots, b_\lambda, b_{\lambda+1}, \ldots, b_p\}$$

for $p > \lambda + 1$ and

$$E_\lambda = \{(a_i, a_j) | 1 \leq i < j \leq \lambda\} \cup \{(b_i, b_j) | 1 \leq i < j \leq p\} \cup \{(a_i, b_i) | i = 1, 2, \ldots \lambda\}.$$

That is, G_λ consists of two cliques, one of size λ and the other of size $p > \lambda+1$, together with a set of λ edges *between* the cliques. The edge connectivity of G_λ is λ, and $\mathcal{H}(G_\lambda)$ is a star with an empty node in the middle and leaves $a_1, a_2, \ldots, a_\lambda, \{b_1, \ldots, b_p\}$. The graph G_3 with $p = 5$ is shown in Figure 6. If $\lambda \geq 3$, then any optimal solution that increases the connectivity of G_λ by one (i.e., adds $\lceil \frac{\lambda+1}{2} \rceil$ edges) must add a new edge of the type (a_i, a_j), which already exists in G_λ.

THEOREM 6.1. *Any 0-, 1- or 2-edge-connected graph with ≥ 4 vertices can be optimally augmented to a 3-edge-connected graph without adding parallel edges.*

For any $\lambda \geq 3$, there is a family of λ-edge-connected graphs on n vertices, $n \geq \lambda + 2$, which require the addition of parallel edges when optimally augmenting to a $(\lambda + 1)$-edge-connected graph.

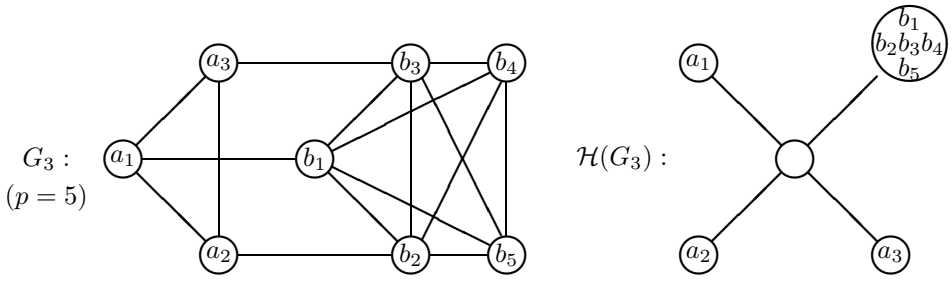


FIG. 6. *An example where an optimal augmentation by one requires parallel edges* (G_3 *with* $p = 5$).

The final lemma characterizes a broad class of graphs which can be optimally augmented by one without parallel edges. Its proof is included in Appendix B.

LEMMA 6.2. *If* λ*, the connectivity of* G*, is smaller than the minimum degree of* G*, then* G *can be augmented by one without parallel edges.*

Appendix A. Finding edge connectivity in a multigraph. The algorithm of [31] finds the edge connectivity of an uncapacitated simple graph in $O(nm)$ time. We now show how to extend the analysis of this algorithm to multigraphs, i.e., graphs with parallel edges. Let m be the number of pairs of vertices which are connected by some edge. Let $p(v)$ be the *maximum* number of parallel edges between v and any one of its neighbors, and define $P = \sum_{v \in V} p(v)$. Then the running time of the algorithm is $O(Pm)$. Note that for simple graphs, $P = n$; hence we obtain the original $O(nm)$. The algorithm of [31] for simple graphs is described and analyzed in [1] (see [12]).

The algorithm of [31]. Let $v_1, \ldots, v_n$ be an arbitrary ordering of the vertices of G. G_i is the graph obtained from G by contracting the set $\{v_1, \ldots, v_i\}$ into a single node called v_0. It is easy to see that if C_i is the minimum cut between v_{i+1} and v_0 in G_i, then λ, the edge connectivity of G, is the smallest of all C_i. The ith phase of the algorithm computes C_i; at the beginning of that phase, any edge of the form (v_0, v_j) in G_i ($v_j \neq v_0$) is assumed to be marked, as well as the vertex v_j.

C_i is computed in two stages. First, all paths of length 1 or 2 from v_{i+1} to v_0, which are called *short paths*, are found and each contributes one unit of flow to C_i. The short paths are found by scanning the adjacency list of v_{i+1}: every edge (v_{i+1}, v_0) is a path of length one; an edge (v_{i+1}, v_j), where v_j is a marked node, followed by a marked edge (v_j, v_0) is a path of length 2. These paths remain untouched. Then all paths of length three or more between v_{i+1} and v_0, which are called the *long paths*, are found. This is done using the original Ford–Fulkerson algorithm by successively building residual graphs while ignoring the edges on the short paths found earlier, finding augmentation paths, and augmenting the flow by one unit for each such path. Observe that since every edge on a short path is connected to either v_0 or v_{i+1}, there is never a need to undo the flow on this edge; it is therefore allowed to ignore these edges in building the residual graphs.

At the end of the computation of C_i, G_{i+1} is constructed: v_{i+1} is contracted with v_0, adjacency lists are updated, and all edges (v_{i+1}, v_j), $v_j \neq v_0$, that participated in some long path are marked (including v_j).

Time analysis for multigraphs. We now extend the analysis in [31] to multigraphs. The short paths in phase i are found by scanning v_{i+1}'s adjacency list. Hence the total time to find all short paths over all phases is $O(C)$, where C is the number of

edges in the graph, since every edge is scanned twice. Similarly, the time to build $G_1, \ldots, G_{n-1}$ and mark the edges and vertices during all phases is $O(C)$. Clearly, $C \leq Pm$. We now show that the time to search for the long paths is bounded by $O(Pm)$.

Suppose that w appears for the first time as the *second* vertex on a *long* path during phase i and that there are x_i such long paths. At the end of that phase, v_{i+1} is contracted with v_0, w becomes adjacent to v_0 from then on, and x_i of its edges are marked. Now suppose that w appears again as the second vertex on some path during phase j, $j > i$. If the number of these paths is not greater than x_i, then they all must be short paths (since they will use the edges that were marked during phase i). If the number of these paths is greater than x_i, then x_i of them must be short, and the remaining x_j paths will be long paths. However, $x_i + x_j$ is bounded by the number of edges between w and v_{j+1} and hence bounded by $p(w)$. This argument can be applied repeatedly, every time w appears as a second vertex on some path, so the total number of long paths that contain w as the second vertex is at most $p(w)$. Therefore, the number of long paths found during the entire algorithm is bounded by P. Since the time to search for a long path is linear in the size of the residual graph, i.e., $O(m)$, the long paths are found in $O(Pm)$.

Appendix B. Augmentation with no parallel edges. We now prove Lemma 6.2, which states that a certain class of graphs can be optimally augmented by one without parallel edges.

LEMMA B.1. *Let G be a graph with no parallel edges and connectivity $\lambda > 1$, and let A, $|A| > 1$, be a leaf in $\mathcal{H}(G)$; that is, A is λ-extreme. There exists a vertex $v \in A$ whose neighbors all belong to A.*

Proof. Since A is λ-extreme, $d(A) = \lambda$ and $d(v) > \lambda$ for any $v \in A$. If $\min(A) = \min_{v \in A}\{d(v)\}$ is the minimum over all degrees of the vertices in A, then $\min(A) > \lambda$. We have

$$\begin{aligned}\lambda|A| < \min(A)|A| &\leq \sum_{v \in A} d(v) \leq d(A) + 2(\#\text{ edges within A}) \\ &= \lambda + 2(\#\text{ edges within A}) \leq \lambda + |A|(|A| - 1)\end{aligned}$$

Hence $\lambda|A| < \lambda + |A|(|A| - 1)$ or $\lambda(|A| - 1) < |A|(|A| - 1)$. Therefore, $|A| > \lambda$ since we assumed that $|A| > 1$. If $|A| > \lambda$, then there must be a vertex $v \in A$ that is connected only to vertices in A since the number of edges going out of A is λ, and this proves the lemma. □

LEMMA B.2. *If λ, the connectivity of G, is smaller than the minimum degree of G, then G can be augmented by one without parallel edges.*

Proof. If λ is smaller than the minimum degree of G, then there are no λ-extreme sets that are singletons. Hence Lemma B.1 assures that any pair of leaves in the cactus $\mathcal{H}(G)$ can be connected by a new edge (i.e., not from the original set of edges). Therefore, algorithm **Aug-1** can always select a set of nonparallel edges that optimally augment G by one. □

Acknowledgments. We would like to thank one of the referees for pointing out the problem with an earlier version of Theorem 5.3. We would also like to thank Prof. Lou Hakimi for preliminary discussions on the augmentation problem.

REFERENCES

[1] G. M. Adelson-Velskii, E. A Dinits, and A. V. Karzanov, *Flow Algorithms*, Nauka, Moscow, 1976 (in Russian).

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[3] J. Bang-Jensen and B. Jackson, *Minimal augmentation and vertex splitting in mixed graphs*, Preprints 5, Institut for Matematik og Datalogi, Odense Universitet, Odense, Denmark, 1992.

[4] R. E. Bixby, *The minimum number of edges and vertices in a graph with edge connectivity n and m n-bonds*, Networks, 5 (1975), pp. 253–298.

[5] G. R. Cai and Y. G. Sun, *The minimum augmentation of any graph to a k-edge-connected graph*, Networks, 19 (1989), pp. 151–172.

[6] E. A. Dinits, A. V. Karzanov, and M. L. Lomosonov, *On the structure of a family of minimal weighted cuts in a graph,* in Studies in Discrete Optimization, A. A. Fridman, ed., Nauka, Moscow, 1976, pp. 290–306 (in Russian).

[7] K. Eswaran and R. E. Tarjan, *Augmentation problems*, SIAM J. Comput., 5 (1976), pp. 653–665.

[8] S. Even and R. E. Tarjan, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[9] A. Frank, *Augmenting graphs to meet edge connectivity requirements*, SIAM J. Discrete Math., 5 (1992), pp. 25–53.

[10] G. N. Fredrickson and J. JáJá, *Approximation algorithms for several graph augmentation problems*, SIAM J. Comput., 10 (1981), pp. 270–183.

[11] H. N. Gabow, *Applications of a poset representation to edge connectivity and graph rigidity*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 812–821.

[12] A. V. Goldberg and D. Gusfield, *Book review: Flow Algorithms, by G. M. Adelson-Velskii, E. A Dinitz, and A. V. Karzanov*, SIAM Rev., 33 (1991), pp. 306–314.

[13] R. E. Gomory and T. C. Hu, *Multi-terminal network flows*, SIAM J. Appl. Math., 9 (1961), pp. 551–560.

[14] M. Grotschel, C. Monma, and M. Stoer, *Polyhedral approaches to network survivability*, Report 189, Institut fur Mathematik, Universitat Augsburg, Augsburg, Germany, 1990.

[15] M. Grotschel, C. Monma, and M. Stoer,*Computational results with a cutting plane algorithm for designing communication networks with low-connectivity constraints*, Oper. Res., 40 (1992), pp. 309–330.

[16] M. Grotschel, C. Monma, and M. Stoer, *Facets of polyhedra arising in the design of communication networks with low-connectivity constraints*, Technical Report 90-40, DIMACS, Rutgers University, Piscataway, NJ, 1990.

[17] D. Gusfield, *Optimal mixed graph augmentation*, SIAM J. Comput., 16 (1987), pp. 599–612.

[18] D. Gusfield,*Very simple methods for all pairs network flow analysis*, SIAM J. Comput., 19 (1990), pp. 143–155.

[19] D. Gusfield, *A graph theoretic approach to statistical data security*, SIAM J. Comput., 17 (1988), pp. 552–571.

[20] T. S. Hsu and V. Ramachandran, *On finding a smallest augmentation to biconnect a graph*, Technical Report TR-91-12, University of Texas at Austin, Austin, TX, 1991.

[21] T. S. Hsu and V. Ramachandran, *A linear time algorithm for triconnectivity augmentation*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 548–559.

[22] Y. Kajitani and S. Ueno, *The minimum augmentation of a directed tree to a strongly k-edge-connected directed graph*, Networks, 16 (1986), pp. 181–197.

[23] A. Kanevsky, *Graphs with odd and even edge connectivity are inherently different*, Technical Report TAMU-89-10, Texas A&M University, College Station, TX, 1989.

[24] A. V. Karzanovand E. A. Timofeev, *Efficient algorithm for finding all minimal edge cuts of a nonoriented graph*, Kibernet., 2 (1986), pp. 8–12 (in Russian); Cybernetics, 2 (1986), pp. 156–162 (in English).

[25] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison–Wesley, Reading, MA, 1989.

[26] D. Matula, *Determining edge connectivity in $O(nm)$*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 249–251.

[27] H. Nagamochi and T. Ibaraki, *A linear time algorithm for finding a sparse k-connected*

spanning subgraph of a k-connected graph, Algorithmica, 7 (1992), pp. 583–596.

[28] D. Naor, *The structure of minimum cuts with applications to edge-augmentation*, Ph.D. thesis, Department of Computer Science, University of California at Davis, Davis, CA, 1991.

[29] D. Naor and V. V. Vazirani, *Representing and enumerating edge connectivity cuts in $\mathcal{RNC}$*, in Algorithms and Data Structures, Proc. 2nd Workshop WADS, F. Dehne, J. R. Sack, and N. Santaro, eds., Lecture Notes in Comput. Sci. 519, Springer-Verlag, Berlin, pp. 273–285.

[30] D. Naor, D. Gusfield, and C. Martel, *A fast algorithm for optimally increasing the edge-connectivity*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 698–707.

[31] V. D. Podderyugin, *An algorithm for finding the edge connectivity of graphs*, Vopr. Kibernet., 2 (1973), p. 136.

[32] A. Rosenthal, and A. Goldner, *Smallest augmentation to biconnect a graph,* SIAM J. Comput., 6 (1977), pp. 55–66.

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

[34] S. Ueno, Y. Kajitani, and H. Wada, *Minimum augmentation of a tree to a k-edge-connected graph,* Networks, 18 (1988), pp. 19–25.

[35] T. Watanabe, *An efficient augmentation to k-edge-connect a graph*, Technical Report C-23, Department of Applied Mathematics, Hiroshima University, Hiroshima, Japan, 1988.

[36] T. Watanabe and A. Nakamura, *Edge-connectivity augmentation problems,* J. Comput. System Sci., 35 (1987), pp. 96–144.

[37] T. Watanabe and A. Nakamura, 3-*connectivity augmentation problems,* Proc. 1988 IEEE International Symposium on Circuits and Systems, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 1847–1850.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1166–1187, August 1997

012

GRAPH DECOMPOSITION IS NP-COMPLETE: A COMPLETE PROOF OF HOLYER'S CONJECTURE*

DORIT DOR† AND MICHAEL TARSI†

Abstract. An H-decomposition of a graph $G = (V, E)$ is a partition of E into subgraphs isomorphic to H. Given a fixed graph H, the H-decomposition problem is to determine whether an input graph G admits an H-decomposition.

In 1980, Holyer conjectured that H-decomposition is NP-complete whenever H is connected and has three edges or more. Some partial results have been obtained since then. A complete proof of Holyer's conjecture is the content of this paper. The characterization problem of all graphs H for which H-decomposition is NP-complete is hence reduced to graphs where every connected component contains at most two edges.

Key words. graph, decomposition, NP-completeness

AMS subject classifications. 03D15, 68R10, 05C70

PII. S0097539792229507

1. Introduction.

Given a graph H, the H-decomposition problem is stated as follows: Can the edge set of an input graph G be partitioned into subgraphs isomorphic to H?

Holyer [10] conjectured the NP-completeness [8] of H-decomposition whenever H consists of at least three edges. In that wide form, the conjecture was known to be false (assuming P $\neq$ NP) even before it was stated. Brouwer and Wilson [4] presented a polynomial-time algorithm for the case where H is the union of t disjoint edges ($H = tK_2$). Independently, Alon [1] obtained the same result after the case of $H = 3K_2$ was studied in details by Bialostocki and Roditty [3]. Later, Preisler and Tarsi [15] presented a polynomial algorithm for the union of a single path of two edges and t disjoint edges, $H = P_3 \bigcup tK_2$. The case where $H = P_3 \bigcup K_2$ was solved previously by Favaron, Lonc, and Truszczynski [7]. Recently, Lonc [12] proved the existence of a polynomial-time algorithm for the case where $H = sP_3$ is the union of s vertex disjoint, two-edges-long paths.

In all of the above, polynomial algorithms were found for H-decomposition in which each connected component of H is either a single-edge- or two-edge-long path. This led to a relaxed version of Holyer's conjecture, restricted to graphs H which contain a connected component with three edges or more.

Several partial results in that direction have been obtained during the last decade. Holyer [10], [9] proved his conjecture for ($H =$) complete graphs, for simple paths, and for simple circuits. Leven (unpublished) presented a proof for the case where H is a star (a complete bipartite graph $K_{1,n}$). Cohen and Tarsi [17] generalized those results to a family of graphs which contains the set of all trees. Masuyama and Hakimi [18] proved NP-completeness for all graphs which include a vertex of degree 1.

A related topic is the factorization problem, which is the analogous problem for vertex partition: Determine for a fixed graph H whether an input graph $G = (V, E)$ contains vertex-disjoint subgraphs, isomorphic to H, such that the union of their

* Received by the editors April 7, 1992; accepted for publication (in revised form) August 31, 1995.

http://www.siam.org/journals/sicomp/26-4/22950.html

† School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel (ddorit@math.tau.ac.il, tarsi@math.tau.ac.il).

vertex sets is V. Kirkpatrick and Hell [11] proved that this problem is NP-complete (NPC) if and only if H contains at least three vertices in a connected component.

The content of this paper is a complete proof of Holyer's conjecture, that is, a proof of the following result.

THEOREM 1.1. *H-decomposition is* NPC *whenever H contains a connected component with three edges or more.*

We conclude this section with some notational remarks.

- Let $G = (V, E)$ be a graph G with vertex set $V = V(G)$ and edge set $E = E(G)$. For another graph $G' = (V, E')$, we use $G - G'$ or $G - E'$ to denote $(V, E - E')$, and for a set V' of vertices, we define $G - V' = (V - V', E - \{e \in E : e \text{ is incident to a vertex of } V'\})$.
- The graph $G = (V, H)$ is *k-connected* if $G - V'$ is connected for any $(k-1)$-element subset V' of V, and it is *k-separable* if it is not $(k+1)$-connected.
- An *H-subgraph* of G is a subgraph of G which is isomorphic to H.
- If G admits an H-decomposition D, then we refer to each graph in D as a *D-part*.
- Let D be an H-decomposition of G and let H' be a D-part. Since H' is an H-subgraph, there exists an isomorphism $f : V(H') \to V(H)$ (select a certain isomorphism if there are more than one). For $v \in V(H') \subseteq V(G)$ and $x \in V(H)$, the relation $x = f(v)$ is denoted by $v = x(H')$. We also say in this case that *v plays the role of x in H'*.

2. Methodological overview. Our proof of Theorem 1.1 is rather long and technical—not to say cumbersome. This section can be read as a stand-alone extended abstract which draws the overall picture without coping with all technical details. Yet it is also an integrated part of the proof, setting the general frame, whereas the following sections fill in the empty squares.

One major difficulty faced in previous attempts to prove Theorem 1.1 lies in its wide nature. Apparently, it is rather easy to custom tailor the necessary machinery for a given graph H. It is a much harder task to design a suit that fits all graphs. For example, if G is H-decomposable, then the degree of each vertex of G is the sum of certain vertex degrees of H. This leads to the following definition of the graph parameter $g(H)$.

DEFINITION 2.1. *Let $H = (V, E)$ be a graph. The greatest common divisor of the degrees $d(x)$ over all vertices $x \in V$ is denoted by $g(H)$.*

Clearly, the set of H-decomposable graphs is reached when $g(H) = 1$, and indeed the graphs H treated in [17] and in [18] all have $g(H) = 1$. For a uniform general proof, a family of simply structured graphs is required, where H-decomposable graphs can be found regardless of the specific graph H at hand. Such a family is merely the set of complete graphs K_n due to the following theorem by Wilson [14], which is considered to be the key theorem of graph-decomposition theory.

WILSON'S THEOREM. *For every graph $H = (V, E)$, the conditions $|E|$ divides $\binom{n}{2}$ and $g(H)$ divides $(n-1)$, which are obviously necessary for the complete graph K_n to be H-decomposable, are also sufficient if n is larger than some constant $n_0(H)$.*

With complete graphs as the basic building blocks, we still use three different main proof schemes, each designed for a certain family of graphs H. It is observed in [17] that if C-decomposition is NPC, where C is a connected component of H, then H-decomposition is also NPC. It then suffices to consider connected graphs H.

Graph decomposition is basically a partition of a set into disjoint subsets taken from a given collection. It seems natural to consider a similar problem, known to be

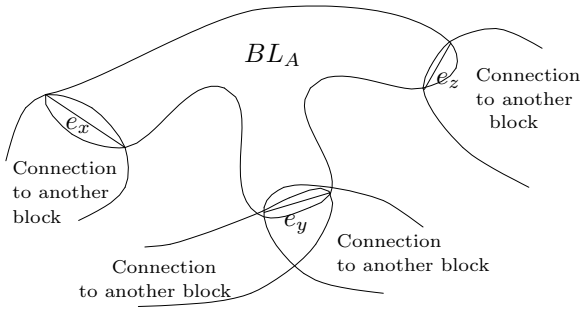


FIG. 1. *A part of* $G_H(I)$ *where* $A = \{x, y, z\}$.

NPC, as a starting point for a polynomial reduction. (H-decomposition is clearly in NP; our task here is to prove NP-hardness.) The problem selected for our first two schemes is the following.

DEFINITION 2.2. *An instance of* k*-XC* (k-eXact Cover) *is a pair* $I = (U, \mathcal{A})$*, where* U *is a finite set and* $\mathcal{A}$ *is a collection of* k*-element subsets of* U*. The* k*-XC problem on* I *is to decide whether there exists a partial collection* $X \subseteq \mathcal{A}$ *of pairwise-disjoint sets such that* $|U| = k \cdot |X|$ *and* $\bigcup_{A \in X} A = U$*.*

For every integer $k \geq 3$, k-XC is known to be NPC (k-dimensional matching is a restricted version of k-XC).

To prove that H-decomposition is NPC, for every instance $I = (U, \mathcal{A})$ of k-XC (for some constant $k \geq 3$), we construct a graph $G_H(I)$ in polynomial time such that H-decomposition of $G(I)$ is equivalent to k-XC on I.

2.1. First scheme: Blocks with isolated boundary edges. Let $I = (U, \mathcal{A})$ be an instance of k-XC. To represent each k-tuple $A \in \mathcal{A}$, we design an "all-or-none" building block. This should be a graph BL_A, which contains k *incidence edges*, $e_1, \ldots, e_k$, each representing the inclusion of an element $x \in U$ in A. Both graphs BL_A and $BL_A - \{e_1, \ldots, e_k\}$ should be H-decomposable.

The graph $G_H(I)$ is formed by merely grouping the blocks together such that every element $x \in U$ is represented by an incidence edge e_x, shared by all blocks BL_A, for which $x \in A$. Other than that, the blocks are vertex disjoint.

An exact cover X of I easily translates into an H-decomposition of $G_H(I)$ as follows: Decompose BL_A for every $A \in X$ and $BL_A - \{e_1, \ldots, e_k\}$ for the other k-tuples A, which are not in X. Since X is an exact cover, given any $x \in U$, the common incidence edge e_x is "used" by one of the blocks and hence an H-decomposition of $G_H(I)$ is indeed formed (see Figure 1).

To meet the "only if" requirement of the reduction, the block should be designed such that any H-decomposition of $G_H(I)$ would impose an H-decomposition on either BL_A or $BL_A - \{e_1, \ldots, e_k\}$ for every $A \in \mathcal{A}$. For clear discussion of that aspect, we introduce some further notation.

DEFINITION 2.3.

- *A* module $M = (V, E, B)$ *is a connected graph on vertex set* V *and edge set* E*, which contains a prespecified subgraph* B*, called its* boundary*. The vertices* (*respectively, edges*) *of* B *are the* boundary vertices (*respectively,* boundary edges) *of* M*. The edges of* $M - B(M)$ *are the* interior edges *of* M *and the* interior vertices *are those incident to no boundary edge.*

- *A* modular extension *of a module M is a graph G which contains M as an induced subgraph such that no edge in $G-M$ is adjacent to an interior vertex of M. We also say in that case that M is a* modular subgraph *of G.*
- *Let G be a modular extension of M. A subgraph T of G which includes an interior edge of M as well as an edge of $G-M$ is called* split *(with respect to M).*

We can use now the new terminology to summarize the scheme as a lemma.

LEMMA 2.1. *H-decomposition is* NP*-complete, if H allows the existence of a* block module *BL_H whose boundary B consists of k vertex-disjoint edges for some integer $k \geq 3$ and it satisfies the following conditions:*

1. *BL_H and $BL_H - B$ are both H-decomposable.*
2. *Any H-decomposition of a modular extension of BL_H contains an H-decomposition of either BL_H or $BL_H - B$.*

It is straightforward to verify that condition 2 indeed provides the "only if" direction to complete the reduction described above. The explicit construction of block modules is presented in section 3.

2.2. Second scheme: Boundary cliques. There exists an inherent flaw in the use of Lemma 2.1 related to the graphic parameter $g(H)$.

An obviously necessary condition for H-decomposability of a graph G is $g(H)|g(G)$. Consequently, if a graph G and a subgraph G' of G are both H-decomposable, then $g(H)|g(G-G')$. Consider the block module BL_H of Lemma 2.1 and its subgraph $BL_H - B$. Both graphs are H-decomposable and the difference between them is the boundary B consisting of vertex-disjoint edges. Clearly, $g(B) = 1$ and hence this scheme is applicable only for graphs H with $g(H) = 1$.

The isolated boundary edges of the first scheme are replaced in the second by m-cliques. The value of m depends on the graph H at hand and is specified in section 4. For reasons that will become clear later, we define $d = m - 1$.

Given an instance $I = (U, \mathcal{A})$ of 3^d-XC, a graph $G_H(I)$ is constructed as follows: For every $A = \{x_1, \ldots, x_{3^d}\} \in \mathcal{A}$, we construct a *grouping* module GR_A, which replaces the block module of the first scheme. The boundary of GR_A consists of 3^d *arms* denoted by $K_1^+, \ldots, K_{3^d}^+$ and 3^d *antiarms* denoted by $K_1^-, \ldots, K_{3^d}^-$. Each arm and each antiarm is an m-clique and they are all vertex disjoint. The module GR_A is built to make both subgraphs $GR_A - \{K_1^+, \ldots, K_{3^d}^+\}$ and $GR_A - \{K_1^-, \ldots, K_{3^d}^-\}$ H-decomposable. For every $x \in U$, let all grouping modules GR_A for which $x \in A$ share a common boundary arm E_x.

The antiarms are taken care of by means of another module, the *r-alternator*, r-A_H. Its boundary consists of r edge-disjoint m-cliques $K^1, \ldots, K^r$, such that the graph obtained by deleting any $r-1$ of them is H-decomposable. We refer to each of these cliques as an *arm* of the alternator.

For each element x included in $A_1, \ldots, A_t$ ($A_i \in \mathcal{A}$), we construct a *t-alternator* AE_x, each arm of which is an antiarm of one of $GR_{A_1}, \ldots, GR_{A_t}$. The graph obtained is $G_H(I)$ (see Figure 2).

Let D_A^- and D_A^+ denote an H-decomposition of $GR_A - \{K_1^+, \ldots, K_{3^d}^+\}$ and $GR_A - \{K_1^-, \ldots, K_{3^d}^-\}$, respectively. An exact cover $X \subset \mathcal{A}$ easily provides an H-decomposition of $G_H(I)$ as follows: Take the union of the decompositions D_A^+ for every $A \in X$ and D_A^- for every $A \notin X$. Consider an element $x \in U$ which belongs to sets $A_1, \ldots, A_t$. Since X is an exact cover, the common arm E_x is covered by exactly one of the $D_{A_i}^+$'s, for which $A_i \in X$ and $t-1$ arms of the t-alternator AE_x are covered, each by a decomposition $D_{A_j}^-$, $A_j \notin X$. The remaining "one-armed alternators" are H-decomposable

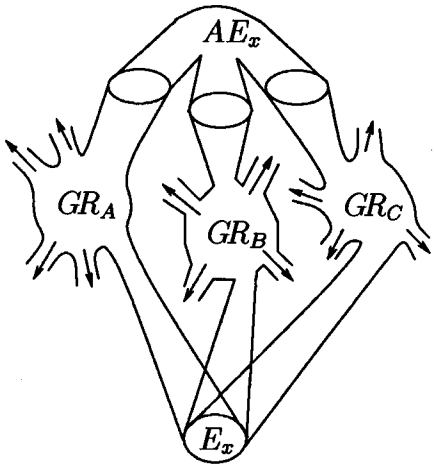


FIG. 2. *A part of* $G_H(I)$ *where* $x \in A, B, C \in \mathcal{A}$.

and thus complete an H-decomposition of $G_H(I)$.

Our construction of the grouping module is based on the existence of the *Holyer graph* $H_{m,3}$, which is defined in [10] (see also section 5), where the following is proven.

LEMMA 2.2. *There are exactly two distinct* K_m*-decompositions of* $H_{m,3}$*:* $K^+ = \{K_i^+\}_{i=1}^{3^{m-1}}$ *and* $K^- = \{K_i^-\}_{i=1}^{3^{m-1}}$*, each consisting of* 3^{m-1} m*-cliques.*

The grouping module GR_A consists of a copy of the Holyer graph $H_{d+1,3}$, where each $(d+1)$-clique in $K^+ = \{K_i^+\}_{i=1}^{3^d}$ and $K^- = \{K_i^-\}_{i=1}^{3^d}$ is an arm of one of $2 \cdot 3^d$ 2-alternator modules. The other $2 \cdot 3^d$ arms of these alternators, 3^d for K^+ and 3^d for K^-, are the *arms* and *antiarms* of GR_A, respectively.

Due to our definition of the alternator and Lemma 2.2, the subgraphs obtained from GR_A by deleting either all of its arms or all of its antiarms are indeed H-decomposable. It remains to assure that any H-decomposition of $G_H(I)$ would impose an exact cover on I. This mission is much more involved and relies on a careful design of the alternator modules, which is left to section 4.

2.3. Third scheme: Triangular graphs. Due to technical reasons that are explicitly stated in section 4.2, our second scheme fails when H is a graph of a certain type, which we call *triangular*. This is basically a triangle, where each of the three edges is replaced by any connected component.

DEFINITION 2.4. *A graph* $G = (V, H)$ *is called triangular by three of its vertices* $v_0, v_1, v_2 \in V$ *if it is the union of three connected components* C_0, C_1, C_2 *and* $V(C_i) \cap V(C_{i+1}) = \{v_i\}$ *for* $i \in \{0, 1, 2\}$*, addition modulo* 3 *(see Figure* 3*).*

Fortunately, Holyer's proof [10] for the case where H is a triangle can be modified to cover all triangular graphs. Holyer's method and its application to triangular graphs is described in section 5.

2.4. Case listing. We conclude this preliminary section with a "checklist" of the various families of graphs for which different versions of the proof are given.

- 3-connected graphs with $g(H) = 1$ are treated in section 3.1.
- 2-connected, 2-separable graphs with $g(H) = 1$ are treated in section 3.2.
- 5-connected graphs with $g(H) > 1$ are treated in section 4.1.

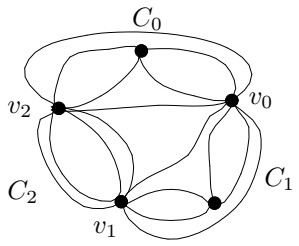


FIG. 3. *A triangular graph.*

- 2-connected, 4-separable, nontriangular graphs with $g(H) > 1$ are treated in section 4.2.
- 1-separable, nontriangular graphs are treated in section 4.3.
- Triangular graphs are treated in section 5.

One can easily verify that the list above indeed covers all connected graphs H. After considering the last two cases in the list, it remains to consider 2-connected, nontriangular graphs. Such graphs with $g(H) = 1$ are treated in the first two cases and graphs with $g(H) > 1$ are treated in the next two. Despite some overlapping, none of the cases is redundant.

3. Explicit construction of BL_H. Block modules which comply with Lemma 2.1 are made up of smaller modules, which we call chains.

DEFINITION 3.1. *A* chain module *related to a connected graph H is a module $C_H = (V, E, B)$ whose boundary contains two edges and which satisfies the following conditions:*

1. *A path in C_H which includes both boundary edges is at least $k+2$ edges long.*
2. *Once either one of the boundary edges is deleted from C_H, the remaining graph is H-decomposable.*
3. *If G is a modular extension of C_H which admits an H-decomposition D, where no D-part includes both boundary edges of C_H, then no D-part is split.*

LEMMA 3.1. *Let H be a connected graph with $k \geq 3$ edges. The H-decomposition problem is* NP*-complete if H allows the existence of a chain module $C_H = (V, E, B)$.*

Proof. A *block module* LB_H is formed of a copy H' of H, every edge of which is a boundary edge of one of k chain modules, which are otherwise vertex disjoint. The other boundary edges of the chains will be denoted by $e_1, \ldots, e_k$ and serve as boundary edges of the block.

We now show that BL_H meets the requirement set in Lemma 2.1.

The graph $LB_H - H'$ is the union of k chains with one boundary edge deleted from each. The same holds for $LB_H - \{e_1, \ldots, e_k\}$. Condition 2 of Definition 3.1 then implies that both LB_H and $LB_H - \{e_1, \ldots, e_k\}$ are H-decomposable, as condition 1 of Lemma 2.1 requires. Let G be a modular extension of LB_H and let D be an H-decomposition of G. Condition 1 of Definition 3.1 makes the boundary edges of a chain too far apart for one H-subgraph to include both of them, and no shortcut in G is possible due to the structure of the block. Then no H-subgraph of G exists which includes both boundary edges of a chain. By condition 3 of Definition 3.1, no D-part is split with respect to any of the chains. By condition 2, the number of edges in a chain is 1 (mod$k$). Consequently, D contains a decomposition of each one of the chain modules with either one of its boundary edges deleted. The only H-subgraph

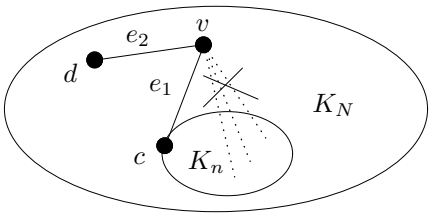


FIG. 4. *The link module.*

which includes an edge of H' with no edge interior to a chain is H' itself. Hence if $H' \in D$, then D contains a decomposition of BL_H. If $H' \notin D$, then a subset of D decomposes $BL_H - \{e_1, \ldots, e_k\}$. Hence condition 2 of Lemma 2.1 is met. □

We proceed by constructing chain modules which satisfy the conditions stated in Definition 3.1. The chain is formed by concatenation of smaller modules, which we call *links*. Each link module includes two boundary edges, the removal of either one of which provides an H-decomposable graph. Several links are chained, each sharing one boundary edge with each neighbor, to form a chain module, where the boundary edges are far apart from each other, as required by condition 1 of Definition 3.1. As pointed out in section 2, this technique is restricted to graphs H where $g(H) = 1$. We assume first that H is 3-connected, which makes condition 3 very easy to meet. The structure of the link module is later modified to fit to the case where H is 2-connected and 2-separable. The case where H is 1-separable is treated in section 4.3 using a different scheme.

3.1. Constructing the chain module where H is 3-connected.

The link module L_H. In accordance with Wilson's theorem, select an integer n such that K_n and K_{n+1} are both H-decomposable. (Notice that $g(H) = 1$ is necessary for the existence of such an n.) Apply Wilson's theorem one more time to choose an integer N such that K_N is K_{n+1}-decomposable. Select a vertex v of the complete graph K_N and delete $n-1$ of the edges incident to v. Also, select c and d, two of the vertices which remain adjacent to v. The graph obtained with boundary edges (v, d) and (v, c) is the link module L_H.

PROPOSITION 3.2. *The subgraph obtained from L_H by deleting either one of its boundary edges $e_1 = (v, c)$, or $e_2 = (v, d)$, is H-decomposable.*

To verify the proposition, start with a K_{n+1}-decomposition D of K_N, where the $(n+1)$-clique, on v, c, and the $n-1$ vertices which then become nonadjacent to v, is a D-part. Once all of the edges incident to v in this clique (including (v, c)) are deleted, the remaining edges form a copy of K_n. Since K_n and K_{n+1} are both H-decomposable, the partition obtained can be refined into an H-decomposition of $L_H - \{(v, c)\}$ (see Figure 4). The same argument holds for $L_H - \{(v, d)\}$.

The chain module C_H. A sequence $(L_1, L_2, \ldots, L_{2k})$ of $2k$ copies of L_H is concatenated into a chain module C_H by means of identifying the edge (v_i, d_i) (that is, the edge (v, d) of L_i) with the edge (c_{i+1}, v_{i+1}) and their corresponding end vertices (v_i with c_{i+1} and d_i with v_{i+1}) for every $1 \le i \le 2k-1$. The two edges (c_1, v_1) and (v_{2k}, d_{2k}) serve as the boundary edges of C_H (see Figure 5).

It remains to show that C_H indeed satisfies the conditions of Definition 3.1. Condition 1 is satisfied since the boundary edges of the chain are located $2k$ links apart

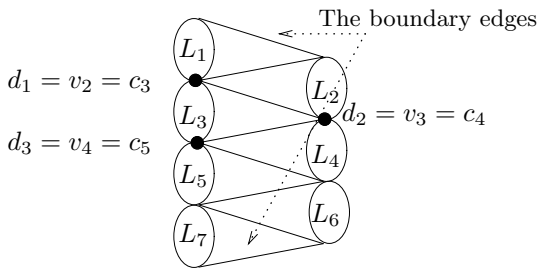


FIG. 5. *The chain module, C_H.*

from each other. Condition 2 is a straightforward corollary of Proposition 3.2. Any split subgraph with at most k edges disconnects at the two end vertices of a boundary edge. The 3-connectivity of H then implies condition 3.

3.2. Constructing the chain where H is 2-connected and 2-separable. 3-connectivity is essential for the last argument. The link module L_H is almost a complete graph, and many copies of any graph H of lower connectivity are split among chains, violating condition 3 of Definition 3.1. To avoid this, we focus on a D-part which includes a boundary edge of a link module. We construct a sparse modified link where the connection of this D-part with the rest of the graph is "minimal." When the modified link module L'_H replaces L_H as the basic brick of a chain, the conditions of Definition 3.1 are met for 2-connected, 2-separable graphs. This mission requires a deeper look into the structure of 2-separable graphs.

DEFINITION 3.2. *A* separating pair (SP) *of a 2-connected graph H is a pair of vertices $\{x, y\} \subset V$ such that the graph $H - \{x, y\} = (V - \{x, y\}, E - \{e | e$ is incident to either x or $y\})$ is not connected. An $\{x, y\}$-component is a subgraph of H induced by the union of $\{x, y\}$ with a connected component of $H - \{x, y\}$. All $\{x, y\}$-components then share vertices x and y (and an edge (x, y) if it exists), and they are otherwise disjoint. An SP-sequence of H is a sequence $(\{x_1, y_1\}, \ldots, \{x_t, y_t\})$ of distinct (not necessarily disjoint) SP's such that all $\{x_j, y_j\}$ for $j > i$ are included in the same $\{x_i, y_i\}$-component.*

The modified link L'_H. Let $H = (V, E)$ be a 2-connected, 2-separable, graph with $g(H) = 1$ and $|E| = k \geq 3$. Let $S = (\{x_1, y_1\}, \ldots, \{x_t, y_t\})$ be an SP-sequence of H of maximum length t. Also, let S be selected such that the degree $d(x_t)$ is minimum among all SP-sequences of length t. From an $\{x_t, y_t\}$-component which does not contain $\{x_{t-1}, y_{t-1}\}$, select a vertex $z \neq y_t$ adjacent to x_t (see Figure 6). In accordance with Proposition 3.2, we start with an H-decomposition D of $L_H - \{(v, c)\}$. Let H_v be the D-part which includes (v, d). Recall that D is formed by decomposing complete graphs and hence no constraint holds regarding the roles of c, d, and v in H_v. We take advantage of this freedom to select $v = x_t(H_v)$, $d = z(H_v)$, and $c \notin V(H_v)$. From H_v we construct $H_{v'}$, a new copy of H, as follows: For every $u \in V(H_v)$ such that $(u, z = d) \notin E(H_v)$, we replace u in H_v by a new vertex u'. That is, every edge (u, w) is deleted and (u', w) is inserted instead. Also, a new vertex v' is added to replace v in $H_{v'}$ in the role of x_t. We define $D' = (D - \{H_v\}) \bigcup \{H_{v'}\}$. The modified link L'_H is the union of all $T \in D'$ and the edge (v', c) with boundary edges (v', c) and (v', d). The H-subgraph $H_{v'}$ is called the *kernel* of L'_H (see Figure 6).

The modified link is meant to satisfy the following stronger version of Proposition 3.2.

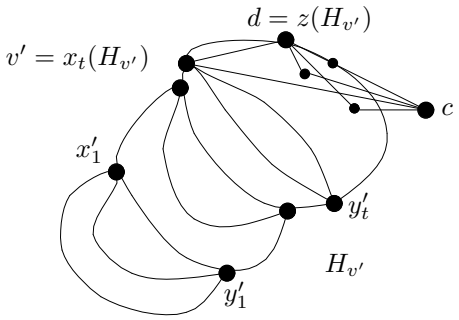


FIG. 6. *The modified link module* L'_H.

PROPOSITION 3.3.

1. *The subgraph obtained from* L'_H *by deleting either one of its boundary edges* (v', c) *or* (v', d) *is* H*-decomposable.*
2. *Let* L *be an* L'_H *modular subgraph of a graph* G*, where either* $\{v', c\}$ *or* $\{v', d\}$ *is an SP of any split* H*-subgraph. If* D *is an* H*-decomposition of* G*, then there exists a* D*-part* $H(L)$ *entirely contained in* L *such that the degree of* $v'(L)$ *in* $L - H(L)$ *is at most* 1.

Proof. Obviously, D' is an H-decomposition of $L'_H - \{(v', c)\}$. Furthermore, the neighborhood of (the set of vertices adjacent to) c in L_H is identical to that of d. The vertex d in H_v plays the role of $z \in V(H)$. The neighbors of this vertex, except for v, were not changed and $c \notin V(H_v)$. Consequently, the neighborhoods of c and d are also identical in L'_H. Thus, switching the roles of c and d in each $T \in D'$, we also obtain an H-decomposition for $L'_H - \{(v', d)\}$. Condition 1 is then satisfied.

Let G and L be as stated in part 2 of Proposition 3.3. Let $H_{v'}$ be the kernel of L and let $S' = (\{x'_1, y'_1\}, \ldots, \{x'_t = v', y'_t\})$ be the copy in $H_{v'}$ of the maximal SP-sequence S as defined above. The modified link is designed to make "most of" its kernel $H_{v'}$ be "separated" from the rest of the link, to which it is connected only through the farthest $\{x_t, y_t\}$-component. Accordingly, the sequence S' is also an SP-sequence of L and of G. Select an edge e from an $\{x'_1, y'_1\}$-component which does not contain the other members of S'. Let $H(L)$ be the D-part which contains e. Clearly, S' is also an SP-sequence of $H(L)$. If $H(L)$ contains an edge out of L, then $\{v'(L), d(L)\}$ or $\{v'(L), c(L)\}$ is an SP of $H(L)$, which can be appended to S' to form a longer SP-sequence—a contradiction. This implies that $H(L) \subset L$. Regarding the degree of $v'(L)$, this vertex belongs to the tth SP of S'. The sequence S was selected to make the degree of x_t minimal. Thus $d(v'(L))$ in $H(L)$ is at least $d(x_t)$ (in H). On the other hand, the degree of $v'(L)$ in L is $d(x_t) + 1$. (It was inserted to replace x_t and then an edge (v', c) was added.) This obviously implies $d(v'(L)) \leq 1$ in $L - H(L)$. □

The chain module C_H, where H is 2-connected and 2-separable, is constructed by the concatenation of $2k$ modified links L'_H exactly as described in the previous subsection for 3-connected graphs. The vertex v of each link is replaced by the vertex v' of the modified link. It remains to verify that the conditions of Lemma 3.1 are met by that construction.

Condition 1 is satisfied since the boundary edges of the chain are located $2k$ modified links apart from each other. Condition 2 is a consequence of part 1 of

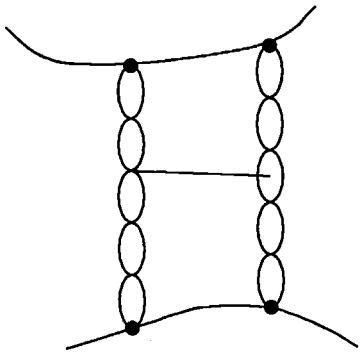


FIG. 7. *The chain module C' after deletion of all $H(L'_i)$'s.*

Proposition 3.3. Let C, G and D be as stated for Condition 3. Considering the structure of a chain, it turns out that an H-subgraph which is split with respect to a modified link, indeed has an SP consisting of the end-vertices of a boundary edge, as required for part 2 of Proposition 3.3. Let C' denote the subgraph obtained from the chain C after the edges of the D-part $H(L_i)$, defined in Proposition 3.3, are deleted from each of the modified links L_i. By Proposition 3.3 at most one of the edges adjacent to v' of each L_i still remains in C'. Two consecutive L_i's share one of these edges and thus at most one such edge remains in all of C'. Thus C' is structured of two 1-separable subgraphs, possibly connected to each other by a single edge. A path between C''s boundary edges passes through at least k articulation points of C' (see Figure 7).

Since H is 2-connected, only one D-part can be partially contained in C'. According to the construction of a chain, the number of edges in C is $1 (\mathrm{mod}\ k)$. This clearly still holds for C'. Hence, there must be a single D-part which contains a single edge of C' and this edge is clearly a boundary edge of C as required.

4. Constructing alternator modules. The link module used in the first scheme has two boundary edges, and an H-decomposable subgraph is obtained when either one of them is deleted. This is impossible, as indicated in section 2.2, when $g(H) > 1$. Instead, we use another module, called a *square*, as the elementary block from which r-alternators are built. The boundary of the square is a 4-cycle such that if either pair of opposite edges is removed, we obtain an H-decomposable graph.

DEFINITION 4.1. *A* square module *related to a graph H is a module $S_H(v_1, v_2, v_3, v_4)$, (abbreviated S_H where no ambiguity is caused) whose boundary is a 4-cycle $B(S_H) = \{e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_3, v_4), e_4 = (v_4, v_1)\}$, and it satisfies the following two conditions:*

1. *Each of $S_H - \{e_1, e_3\}$ and $S_H - \{e_2, e_4\}$ is H-decomposable.*
2. *Every H-decomposition of a modular extension of S_H contains an H-decomposition of either one of the above two subgraphs.*

Square modules are combined to form an r-alternator module $r\text{-}A_H$ as follows.

Let u be a vertex of H for which the degree $d = d(u)$ is maximum. The vertex set of the alternator contains subsets $V^q = \{v_i^q : 0 \le i \le d\}$ for $1 \le q \le r$ and $W = \{w_{i,j} : i \ne j,\ 0 \le i, j \le d\}$. Now construct $r\binom{d+1}{2}$ square modules $S_{i,j}^q = S_H(v_i^q, v_j^q, w_{j,i}, w_{i,j})$ for every $1 \le q \le r$ and $0 \le i < j \le d$. Let the $S_{i,j}^q$'s be vertex disjoint except for the common vertices of the V^q's and W. Also, for each $0 \le i \le d$, construct a copy H_i^-

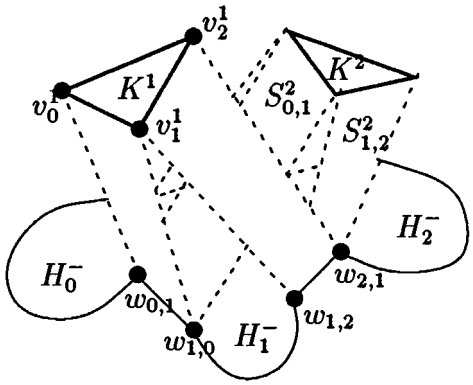


FIG. 8. *The* 2*-alternator module for* $d = 2$.

of $H - \{u\}$, where the d vertices $w_{i,j}, j \neq i$, serve as the d neighbors of the deleted vertex u. We refer to these $(H - \{u\})$-subgraphs as *wings*. Let the vertex sets of the wings otherwise be disjoint from each other and from the V^q's and W. The boundary of r-A_H is induced by $\bigcup_{q=1}^{r} V^q$. The induced $(d+1)$-cliques on each vertex set V^q are denoted by K^q and serve as the arms of the module (see Figure 8).

We first verify that the alternator indeed complies with the scheme described in section 2.2.

PROPOSITION 4.1. *The subgraph obtained from an* r*-alternator by deleting* $r-1$ *of its arms is* H*-decomposable.*

Proof. Select $1 \leq q \leq r$ and delete each of the H-subgraphs induced by $V(H_i^-) \bigcup \{v_i^q\}$ for $0 \leq i \leq d$. Once all of the cliques $K^{q'}$, $q' \neq q$, are also removed, there are altogether two edges deleted from each square $S_{i,j}^q$, which form a complete matching on its boundary. The remaining subgraph is thus H-decomposable by condition 1 of Definition 4.1. □

We now show that the process by which an exact cover on I is translated into an H-decomposition of $G_H(I)$, described in section 2.2, is indeed invertible. Once this is done, the entire scheme is summarized into the existence of square modules.

LEMMA 4.2. *If* H *allows the existence of a square module* (*cf. Definition* 4.1)*, then* H*-decomposition is* NPC.

Proof. Let $G_H(I)$ be the graph whose construction is detailed in section 2.2, and let D be an H-decomposition of $G_H(I)$. Let G' denote the subgraph obtained from $G_H(I)$ after the deletion of the D-parts which cover the interior of all of the square modules $S_{i,j}^q$ in all of the alternator modules which form $G_H(I)$. Applying condition 2 of Definition 4.1, G' is the union of all wings of the alternators involved, with some additional edges incident with the common boundary vertices. It turns out that the degree sequence of the vertices of G' is that of t disjoint copies of H, where t is the total number of wings in $G_H(I)$, which is also the number of parts in an H-decomposition of G', except for some excessive repetition of the maximal degree d. Now focus on the vertices of a wing. Each of them is of degree at most d, which implies that no such vertex is shared between two distinct D-parts; otherwise, there would be too many vertices of certain small degree for t edge-disjoint D-parts. Consequently, for any r-alternator module in $G_H(I)$, each wing is entirely contained in a single D-part, which must be induced by $V(H_i^-) \bigcup \{v_i^q\}$ for some $1 \leq q \leq r$. Again applying condition 2 of Definition 4.1 to the square modules $S_{i,j}^q$ for that specific value of q, it turns out that

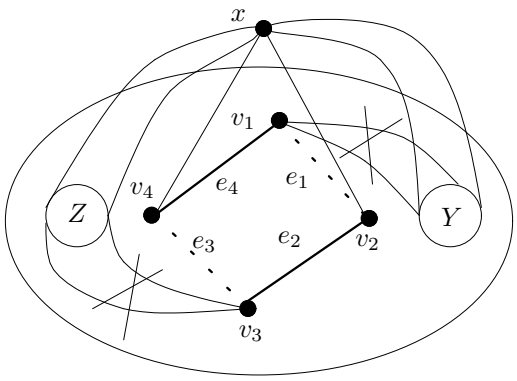


FIG. 9. *The presquare module.*

the same q is used for each $0 \le i \le d$. In particular, if we focus on the 2-alternators of the grouping modules, D provides a K_{d+1}-decomposition of the $H_{d+1,3}$-subgraph of each GR_A. By Lemma 2.2, it is either K^+ or K^-. Therefore, $X = \{A | D_A^+ \subset D\}$ forms a solution for I. □

It now remains to argue that square modules indeed exist. Notice that condition 1 of Definition 4.1 might be met regardless of $g(H)$ because the degrees of vertices in the subgraph obtained do not depend on which one of the two pairs of edges is deleted. However, $g(H) > 1$ appears to be helpful for meeting condition 2, as indicated by the following lemma. (In contrast with condition 2, if $g(H) = 1$, then a D-part may contain two adjacent boundary edges.) This is why our first scheme did not become redundant once the second was developed.

LEMMA 4.3. *If $g(H) > 1$, then condition 2 of Definition 4.1 is a consequence of the apparently weaker requirement:*

$2'$. No D-part of an H-decomposition D of a modular extension of S_H is split.

Proof. Let D be an H-decomposition of a modular extension of $S = S_H(v_1, v_2, v_3, v_4)$. Let D' denote the set of all D-parts which include an interior edge and are hence entirely contained in S if condition $2'$ holds. Clearly, $R = S - \bigcup_{T \in D'} T$ is a subset of $B(S) = \{e_1, e_2, e_3, e_4\}$. Since $g(H) > 1$, the difference between the degrees of a vertex in two unions of disjoint H-subgraphs is either 0 or at least 2. Thus if condition 1 is satisfied, then R is a complete matching on $\{v_1, v_2, v_3, v_4\}$ that is either $R = \{e_1, e_3\}$ or $R = \{e_2, e_4\}$, which is the assertion of condition 2. □

The first step towards establishing the existence of S_H is the following construction of a *presquare* module for any given graph H.

The presquare module PS_H. Let H be any graph. In accordance with Wilson's theorem, select integers n and N such that K_n is H-decomposable and K_N is K_n-decomposable. Make N big enough to allow two vertex-disjoint copies of K_n in a K_n-decomposition of K_N. Select four vertices v_1, v_2, v_3, v_4 of the graph $K_N = (V, E)$ and two sets $Y, Z \subset V$ of $n - 2$ vertices each, disjoint from each other and from $\{v_1, v_2, v_3, v_4\}$. Insert a new vertex $x \notin V$ and define $PS_H(v_1, v_2, v_3, v_4)$ to be

$$(V \cup \{x\}, E - \{(v_1, y), (v_3, z) : y \in Y, z \in Z\} \cup \{(x, u) : u \in Y \cup Z \cup \{v_2, v_4\}\}).$$

Let the boundary $B(PS_H)$ of the module be the 4-cycle $\{e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_3, v_4), e_4 = (v_4, v_1)\}$ (see Figure 9).

The presquare satisfies the first condition of Definition 4.1.

PROPOSITION 4.4. *Each of* $PS_H - \{e_1, e_3\}$ *and* $PS_H - \{e_2, e_4\}$ *is* H*-decomposable.*

To verify the proposition, delete the edges of the two n-cliques induced by $Y \cup \{x, v_2\}$ and by $Z \cup \{x, v_4\}$. If the two edges $e_1 = (v_1, v_2), e_3 = (v_3, v_4)$ are also deleted, then the remaining edges are those of the original K_N from which two n-cliques induced by $Y \cup \{v_1, v_2\}$ and by $Z \cup \{v_3, v_4\}$ are deleted. By the definition of n and N, This remaining graph can be partitioned to complete K_n-decomposition, and hence H-decomposition, of $S_H - \{e_1, e_3\}$. The same argument holds for $S_H - \{e_2, e_4\}$.

The means by which the presquare is used to construct the square module vary as various families of graphs H are considered.

4.1. The square module for 5-connected graphs. If H is 5-connected and $g(H) > 1$, we simply define $S_H = PS_H$. The four boundary vertices of PS_H make any split subgraph 4-separable. The assertion of condition $2'$ of Lemma 4.3 immediately follows.

4.2. Constructing the square module for 2-connected, 4-separable, nontriangular graphs H with $g(H) > 1$. When graphs H of lower connectivity are considered, we face a situation that is analogous to the one we dealt with in section 3.2. Here we take similar measures as we define a modified presquare module PS'_H. The construction here is more complex, and several copies of the modified presquare are required to form one square module. It so happens that the square module obtained fails in meeting the definition if H is either 1-separable or triangular. These two cases are treated separately.

Separating sets of two, three, or four vertices are involved here, and hence some more definitions are required.

DEFINITION 4.2. *A* minimal separating set (MS) *of a graph* $G = (V, E)$ *is a set of vertices* $S \subset V$*, such that* $(G - S) = (V - S, E - \{e : e \text{ is incident to an element of } S\})$ *is not connected and no proper subset of* S *has this property. An* S-component *is a subgraph of* G*, induced by the union of* S *and a connected component of* $G - S$*.*

A k-separating sequence (k-SS) *is a sequence* $Q = \{S_1, S_2, \ldots, S_n\}$ *of distinct MSs of cardinality* $|S_i| \le k$ *such that their cardinalities* $|S_1|, \ldots, |S_n|$ *form a nondecreasing sequence and for every* $1 \le i \le n - 1$*, all* S_j*'s,* $j > i$*, are contained in the same* S_i*-component. Let* $n_j(Q)$ *denote the number of MSs in* Q *which are of cardinality* j*. We define a partial order among* k*-SSs of* G *by* $Q_1 > Q_2$ *if* $n_j(Q_1) > n_j(Q_2)$ *for the smallest* j*, where* $n_j(Q_1) \ne n_j(Q_2)$*. A* maximal k-SS *is maximal with respect to that partial order.*

Let $Q = \{S_1, S_2, \ldots, S_n\}$ *be a* k*-SS of* G*. Let* $C_0(Q, G)$ *denote the union of the* S_1*-components which do not contain* S_2 *and let* $C_n(Q, G)$ *be one of the* S_n*-components which do not contain* S_{n-1}*. (If* $n = 1$*, then* $C_1(Q, G)$ *is any component and* $C_0(Q, G)$ *is the union of all the others.)*

PROPOSITION 4.5. *Let* H *be a subgraph of* G *and* $Q = \{S_1, \ldots, S_n\}$ *be a maximal* k*-SS of* H *such that no edge of* $G - H$ *is incident to any vertex of* $H - C_n(Q, H)$*. If* H' *is an* H*-subgraph of* G *which includes an edge* $e_0 \in C_0(Q, H)$*, then* S *is a (maximal)* k*-SS of* H'*.*

Proof. Since no edge of $G - H$ is incident to $V(H - C_n)$, the sequence Q is also a k-SS of G. Being as big as H, the subgraph H' includes an edge $e_n \in C_n(Q, G)$. The removal of any S_i disconnects e_0 from e_n; hence there exists an MS of H', $S' \subseteq S_i$. We should prove that $S' = S_i$. Take the smallest i for which $S' \subsetneq S_i$. Since H' is isomorphic to H and Q is maximal, S' must equal S_t for some $t < i$ (otherwise,

$(S_1, S_2, \ldots, S', \ldots) > Q)$, that is, $S' = S_t \subsetneq S_i$ for some $t < i$. This is a contradiction since no MS is a proper subset of another. □

The modified presquare module PS'_H. Recalling the construction of PS_H and Proposition 4.4, form an H-decomposition D of $PS_H - \{e_1, e_3\}$, where $e_1 = (v_1, v_2)$ and $e_3 = (v_3, v_4)$. Let $Q = (S_1, \ldots, S_n)$ be a maximal 4-SS, where $S_n = \{x_1, \ldots, x_r\}$, such that $d(x_r)$ is minimum among all maximal 4-SSs. Select a vertex $z \notin S_n$ which is adjacent to x_r in $C_n(Q, H)$.

For the D-part H_{v_1}, which contains the edge (v_1, v_4), select $x_r(H_{v_1}) = v_1$ and $z(H_{v_1}) = v_4$. (No constraint holds regarding the roles of v_1 and v_4 in H_{v_1} since D is formed by decomposing complete graphs.) Also, verify that v_3 is not a vertex of H_{v_1}.

From the D-part H_{v_1}, construct $H_{v'_1}$, a new copy of H as follows: For every $u \in V(H_{v_1})$ such that $(u, z(H_{v_1}) = v_4) \notin E(H_{v_1})$, replace u in H_{v_1} by a new vertex u'. Also, add a new vertex v'_1 to replace v_1 in the role of $x_r(H_{v_1})$.

Similarly, let the role of x_r and z in the D-part H_{v_3}, which includes the edge (v_3, v_2), be played by the vertices v_3 and v_2 in that order. (Note that (v_3, v_2) can be any edge of an n-clique that is vertex disjoint from the clique which includes (v_1, v_4).) Using the same scheme as above, construct $H_{v'_3}$ from the D-part H_{v_3}. Also, add a new vertex v'_3 to replace v_3 in the role of $x_r(H_{v_3})$.

Define $D' = (D - \{H_{v_1}, H_{v_3}\}) \bigcup \{H_{v'_1}, H_{v'_3}\}$ and let PS'_H be the union of all $T \in D'$ and the edges $e'_1 = (v'_1, v_2)$ and $e'_3 = (v'_3, v_4)$. The boundary vertices of PS'_H are v'_1, v_2, v'_3, v_4 and the 4-cycle $e'_1, \ldots, e'_4$ on these vertices, in that order, is the boundary of the module. We refer to that specific copy as $PS'_H(v'_1, v_2, v'_3, v_4)$.

PROPOSITION 4.6. *The modified presquare module satisfies condition* 1 *of Definition* 4.1.

Proof. Obviously, D' is an H-decomposition of $PS'_H - \{e'_1, e'_3\}$. Furthermore, the neighborhood of v_2 in PS_H is identical to that of v_4. The vertex v_4 in H_{v_1} plays the role of $z \in V(H_{v_1})$. The neighbors of v_4, except for v_1, were not changed and v_2 is not a vertex of H_{v_1}. Similarly, the neighbors of v_2, except for v_3, were not changed and v_4 is not a vertex of H_{v_3}. Consequently, the neighborhoods of v_2 and v_4 are also identical in $H_{v'_1}$ and $H_{v'_3}$. Thus switching the roles of v_2 and v_4 in each $T \in D'$, we also obtain an H-decomposition for $PS'_H - \{e'_2, e'_4\}$. □

We are now in a position to prove the following stronger version of Proposition 4.4.

PROPOSITION 4.7. *If G is a modular extension of a modified presquare $PS'_H(v'_1, v_2, v'_3, v_4)$ which admits an H-decomposition D, then there exist two D-parts $H(PS'_H, v'_1)$ and $H(PS'_H, v'_3)$, entirely contained in PS'_H, such that each of the degree of v'_1 and that of v'_3 in $PS'_H - \{H(PS'_H, v'_1), H(PS'_H, v'_3)\}$ is at most* 1.

Proof. Let $Q = (S_1, \ldots, S_n)$ be the maximal 4-SS of the subgraph $H_{v'_1}$ of PS'_H, as defined above. By the construction of $H_{v'_1}$, the sequence Q is also a 4-SS of PS'_H and of G. To verify the proposition, take a D-part H' which includes an edge $e_0 \in C_0(Q, H)$. By Proposition 4.5, Q is a 4-SS of H'. Regarding the degree of v'_1, this vertex belongs to the nth SS of Q, $S_n = \{x_1, \ldots, x_r\}$. The sequence Q was selected to make the degree of x_r the minimum. Thus $d(v'_1)$ in H' is at least $d(x_r)$ (in H). If H' includes any edge out of PS'_H, then the separating 4-set of G, $\{v'_1, v_2, v'_3, v_4\}$, is also a separating 4-set of H' which obviously does not contain S_n (since only v'_1 is included in S_n) and can be appended to Q to form a greater 4-SS-sequence — a contradiction, which implies that $H' \subseteq PS'_H$. The degree of v'_1 in PS'_H is $d(x_r) + 1$. (It was inserted to replace x_r and then an edge e'_1 was added.) Recalling that $d(v'_1)$ in H' is at least $d(x_r)$, H' is the D-part $H(PS'_H, v'_1)$, as the proposition states. The same argument

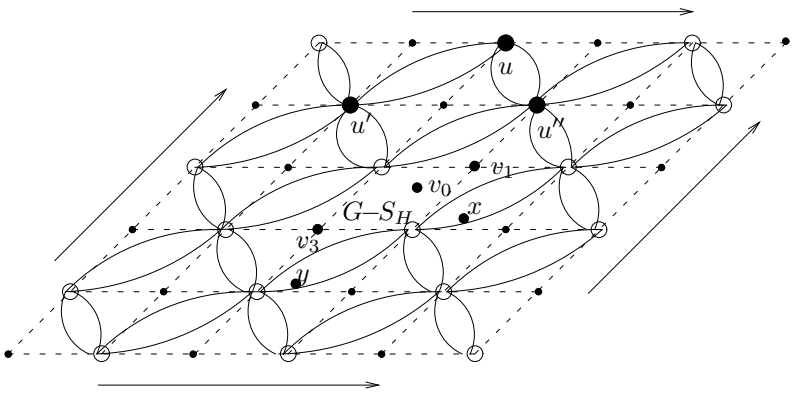


FIG. 10. *G'. The doted grid lines are not edges of the graph and are drawn for reference only. "$\circ$" and "$\bullet$" denote fat and thin vertices, respectively. The vertices x and y can be located in any of the squares surrounding v_1 and v_3.*

can be applied to v_3' to obtain $H(PS_H', v_3')$. □

In light of Proposition 4.7, we refer to the boundary vertices v_1' and v_3' as the *thin* vertices of the modified presquare, and, accordingly, v_2 and v_4 are the *fat* ones.

Constructing the square module S_H. Start with an $m \times m$ (or $(m+1) \times (m+1)$ if m is odd) rectangular grid on the two-dimensional torus. Let one square of the grid be the boundary of S_H and let the vertices of this square be denoted by v_1, v_2, v_3, v_4. Fill all of the other grid squares with copies of PS_H' whose boundaries intersect according to their geometric adjacency. While doing so, make each grid vertex either thin in all of the modified presquares which contain it or fat in all of them. Thus a "thin" "fat" 2-coloring of the grid vertices is induced. Let v_1 and v_3 be colored "thin." The graph obtained is the square module $S_H(v_1, v_2, v_3, v_4)$. A modular extension of S_H can be represented by filling the empty boundary square with $G - S$ (see Figure 10).

LEMMA 4.8. *S_H is a square module* (*cf. Definition* 4.1).

Proof. As a consequence of Propositions 4.6 and 4.7, condition 1 of Definition 4.1 is satisfied by S_H. We now prove that condition $2'$ of Lemma 4.3 is also satisfied.

Let D be an H-decomposition of a modular extension G of S_H. Let G' denote the graph obtained from G by deleting all of the D-parts $H(PS_H', v_1')$ and $H(PS_H', v_3')$, defined in Proposition 4.6, from all the PS_H''s in S_H. As a result of that proposition, each thin grid vertex, except v_1 and v_3, is completely saturated by the above D-parts and is hence isolated in G'. The remainder of each modified presquare is a "diagonal *strip*" connected to the rest of G' by the two fat grid vertices, to which we refer as the *end vertices* of that strip. There might remain a single edge connecting v_1 to an interior vertex x of S_H and one connecting v_3 to an interior vertex y (in addition to many vertices adjacent to v_1 and to v_3 in $G - S_H$).

Let H_0 be an H-subgraph of G' which contains a vertex v_0 of $G - S_H$. Assume that H_0 contains an interior vertex of S_H. As an H-subgraph, H_0 is too small to go around the whole torus. Let u be a grid vertex in H_0 which is furthest (in geometric distance on the torus) from the boundary square. At most two strips lead from u to the boundary and thus either H_0 is 1-separable or it is triangular by u and the other end vertices of the above two strips u' and u'' (see Figure 10). More careful attention should be paid to the case where H_0 is limited to the squares which surround the boundary and includes the edge (v_1, x) (and/or (v_3, y)). A short case analysis shows

that also in that scenario H_0 is either 1-separable or triangular by v_1, x, and a certain grid vertex whose identity depends on the exact location of x. Since H is 2-connected and nontriangular, the above leads to a contradiction and thus condition $2'$ is satisfied. □

4.3. The reduction for 1-separable nontriangular graphs. For a 1-separable graph H, we first select a particular 2-connected component C of H. The graph $H - C$ is called the *tail* of H. Given an instance I of 3^d-XC, we construct $G_C(I)$ and then expand it to obtain $G_H(I)$ by inserting the right number of tails, hanging at each vertex. Being 2-connected, C can replace H in the scheme described in the last section. Yet some minor modifications of the construction of $G_C(I)$ are required to cover the case where $g(C) = 1$. The presquare module is replaced by a structure which we call a new presquare module, NS_H; this is also an expansion of a known module, the modified presquare module, PS'_C.

Careful selection of C assures that an H-decomposition of $G_H(I)$ (and of NS_H) implies a C-decomposition of $G_C(I)$ (and of PS'_C). Finally, we claim that every C-decomposition of $G_C(I)$ indeed implies a solution for I.

This subsection is divided into three parts. In the first part, we define the notion of expansion and describe the new presquare module. In the second part, we show that an H-decomposition of an expanded graph implies a C-decomposition of the original one. We conclude this subsection by showing that C-decomposition of $G_C(I)$ implies a solution for I when $g(C) = 1$. For $g(C) > 1$, this fact was already shown in section 4.2. We start by stating the rules by which the subgraph 2-connected block C is selected.

Let x be a separating vertex of H such that $\{x\}$ is the last MS of a maximal 1-SS and let C be a 2-connected $\{x\}$-component. Choose x and C to make $|E(C)|$ the maximum among all such selections. If such C does not exist, then H contains a vertex of degree 1. NP-completeness for that case is proven in [18].

We are now in a position to define the expanded graph.

Definition 4.3.

- *Let D be an H-decomposition of G and let x and v be vertices of H and G, respectively. We define $N(x, v, D)$ to be the number of D-parts H' for which $x(H') = v$.*
- *Let x be a separating vertex of H and let C be an $\{x\}$-component of H. Let D be a C-decomposition of a graph G. We construct the $\{H, C, D\}$-expansion G' as follows: For each vertex v of G, construct $n(v) = N(x, v, D)$ copies of $H - C$, in which v is identified with x and the other vertices are new. These copies are denoted by $(H - C)_{i,v}$ for $1 \le i \le n(v)$ and $v \in V(G)$. The expanded graph G' is*

$$G' = G \bigcup_{v \in V(G)} \bigcup_{1 \le i \le n(v)} (H - C)_{i,v}.$$

The new presquare module NS_H. Let D be a C-decomposition of PS'_C. We define the new presquare module, $NS_H(D)$, to be an $\{H, C, D\}$-expansion of PS'_C.

In section 4.2, it is shown that the modified presquare module PS'_C satisfies both condition 1 of Definition 4.1 and Proposition 4.7. Moreover, these two conditions are sufficient for constructing a square module that satisfies Definition 4.1. We now show that the definition of NS_H is independent of the selection of D and prove the conditions satisfied by the new presquare module.

LEMMA 4.9.

1. *$PS_C(v_1,\ldots,v_4)-\{e_1,e_3\}$ admits a C-decomposition D_1 such that $N(x,v_2,D_1) = N(x,v_4,D_1)$.*
2. *There exist C-decompositions D_1 and D_2 of $PS'_C - \{e_1,e_3\}$ and $PS'_C - \{e_2,e_4\}$, respectively, such that $N(x,v_2,D_1) = N(x,v_4,D_1) = N(x,v_2,D_2) = N(x,v_4,D_2)$.*
3. *$NS_H = NS_H(D_1) = NS_H(D_2)$ is well defined and satisfies condition 1 of Definition 4.1.*
4. *The new presquare module NS_H satisfies Proposition 4.7 if every H-decomposition of NS_H implies a C-decomposition of a PS'_C subgraph of NS_H.*

Proof.

1. Recall that by its definition, $PS'_C - \{e_1,e_3\}$ is K_N-decomposable for some constant N for which K_N is C-decomposable. The total symmetry of K_N clearly allows the required structure of D_1.
2. Let D_1 be as above and let D_2 be the decomposition obtained from D_1 by switching the roles of v_2 and v_4.
3. Add to each D_i-part ($i = 1,2$) one copy of $H - C$ to obtain an H-decomposition in accordance with condition 1 of Definition 4.1.
4. Since PS'_C satisfies Proposition 4.7, the new presquare module NS_H also satisfies Proposition 4.7 if every H-decomposition of NS_H implies a C-decomposition of a PS'_C-subgraph. □

LEMMA 4.10. *Every H-decomposition of G_H implies a C-decomposition of G_C, where G_H is an $\{H,C,D\}$-expansion of G_C.*

Proof. Let D be an H-decomposition of G_H. If each $\{x\}$-component of H is isomorphic to C, then each D-part is an edge-disjoint union of C-subgraphs. Therefore, assume that not all of the $\{x\}$-components are isomorphic. We wish to prove that for every $D' \in D$, the set $E(D') \bigcap E(G_C)$ is an edge-disjoint union of copies of C. Take any separating vertex $v \in V(G_C) \subset V(G_H)$ and any $\{v\}$-component C' in G_H, which is edge-disjoint from G_C.

Note that C was selected as the end component of a maximal sequence of separating vertices and hence at most one $\{x\}$-component of H is 1-separable. First, assume that C' is 1-separable. Let H' be a D-part that intersects C' and contains a maximal path of separating vertices in C'. Since the size of H' equals the size of H, the vertex v is also a separating vertex of H'. Therefore, H' contains a maximal path of separating vertices in H and hence does not contain any separating vertex from another tail of H. Thus at most one 1-separable component of H is contained in each D-part. Since the number of such components in G_H equals $N = \frac{E(G_H)}{E(H)}$, each 1-separable component of H is contained in exactly one D-part.

The graph $G_H - G_C$ contains N edge-disjoint copies of $H - C$, i.e., N edge-disjoint copies of each 2-connected component of H, except C. Since $E(C)$ was chosen to be maximum, no component of $G_H - G_C$ is shared between more than one D-part. (Otherwise, there will be too many small components.) If we remove all of those components from D, we obtain a C-decomposition of G_C. □

From this presquare module, we construct the modified square module S'_H following the construction for 2-connected, nontriangular H, replacing each PS'_H by a copy of NS_H. Since the new presquare module satisfies both condition 1 of Definition 4.1 and Proposition 4.7, the square obtained satisfies the following conditions:

1. Each of $S'_H - \{e_1,e_3\}$ and $S'_H - \{e_2,e_4\}$ is H-decomposable.
2. No D-part of an H-decomposition D of a modular extension of S'_H is split.

This second condition replaces condition 2 of Definition 4.1 only when $g(C) > 1$. The rest of the proof for nontriangular, 1-separable graphs therefore takes different paths for $g(C) > 1$ and for $g(C) = 1$. Notice that since H is nontriangular, so is C.

The case where $g(C) > 1$. For an instance $I = (U, \mathcal{A})$ of 3^d-XC, construct $G = G_C(I)$ (as defined in section 2.2), replacing each square module by the above modified square S'_H. Let $S(G)$ be the union of all square modules in G. Let D be any C-decomposition of $G_C(I)$ and let $D' = \{P : P \in D, P \not\subset S(G)\}$. Notice that $n(v) = N(z, v, D')$ is independent of D (see section 4.2). Finally, define $G_H(I)$ to be an $\{H, C, D'\}$-expansion of G.

By Lemma 4.10, every H-decomposition of $G_H(I)$ implies a C-decomposition of $G_C(I)$ and hence a solution for the instance I.

The case where $g(C) = 1$. When $g(C) = 1$, we modify the construction of $G_C(I)$. The first step in the construction of the alternator was the selection of a vertex u, for which $d(u)$ was maximum. The selection of u changes when $g(C) = 1$, but otherwise the construction remains the same. Recall that the construction of $G_C(I)$ did not involve any constraints on $d(u)$ and $g(C)$. (The constraints were required only for the correctness proof.)

The graph $G_H(I)$ is therefore, constructed as follows: Select a vertex z in C and let $d = d(z)$. Given an instance $I = (U, \mathcal{A})$ of 3^d-XC, construct $G_{C,z}(I)$ with the following exception: For any 2-alternator module in $G_{C,z}(I)$ and $0 \le i \le d$, let $w_{i,i+1}$ be a "thin" vertex in $S^1_{i,i+1}$ (additions modulo $d+1$) and let $w_{i+1,i}$ be a "thin" vertex in $S^2_{i+1,i}$ (see the discussion in section 4 of the r-alternator module for the definition of $w_{i,j}$ and $S^q_{i,j}$). Similarly to the case where $g(C) > 1$, let $G_H(I)$ be an $\{H, C, D'\}$-expansion of $G_C(I)$.

The validity of the construction follows from Lemma 4.11 and the selection of z.

LEMMA 4.11. *H-decomposition is* NPC *if there exists a vertex z in C such that for every C-decomposition D of $G_{C,z}(I)$, each wing is contained in exactly one D-part.*

Proof. Suppose z is a vertex which satisfies the above and let D be a C-decomposition of $G_{C,z}(I)$. Take any 2-alternator of $G_{C,z}(I)$, $A = 2\text{-}A_C$, and let D_i, $0 \le i \le d$, be the D-parts which are internal to A but not to any square module. Since each wing is contained in exactly one D-part, $V(D_i)$ is $V(H_i^-) \bigcup \{v_i^{q_i}\}$ for some $q_i \in \{1, 2\}$.

We first show that $q = q_0 = \cdots = q_d$. Suppose without loss of generality that $q_0 = 1$. Since $w_{0,1}$ is "thin" in $S^1_{0,1}$ and $w_{1,0}$ is "thin" in $S^2_{1,0}$, the edge $(w_{1,0}, v_1^2)$ is not in D_1 and therefore q_1 also equals 1.

Now take any H-decomposition D of $G_{H,z}(I)$. By Lemma 4.10, D contains a C-decomposition D' of $G_{C,z}(I)$. As seen above, all the 2-alternators are decomposed properly and a solution for I is obtained. □

To complete this section we show how vertex z is selected in order to meet the condition of Lemma 4.11. (For $g(C) > 1$, this condition is a simple corollary of condition 2 of Definition 4.1.)

First, assume that x is adjacent to every vertex in C and select z to be x. Let D be a C-decomposition of $G_C(I)$, which exists by Lemma 4.10, and let D' be the set of D-parts which are not included in any square module. Take any vertex $w = w_{i,j}$. If w plays the role of x in a D'-part, then its vertex set V is included in $\{w_{i,k}, 0 \le k \ne i \le d(x)\} \bigcup \{w_{j,i}\} \bigcup \{v_i^q, q = 1, 2, \ldots\}$. If $w_{j,i} \in V$, then its degree is 1—a contradiction. Therefore, each wing is contained in at least one D'-part and a counting argument shows that it must contain all of the wing. Hence all vertices playing the role of x

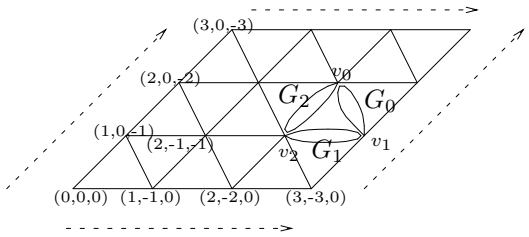


FIG. 11. *$H_{3,3}$ embedded in the (two-dimensional) torus (and modified for M_p).*

are v_i^q's. Since each v_i^q is connected to at most one wing, each wing is contained in exactly one D'-part.

Finally, we can assume that there exists a vertex v in C which is not adjacent to x. In that case, select z to be a vertex whose distance from x is maximal and denote its distance from x by l.

Again, let D' be a C-decomposition of $G_C(I)$ after removing the parts which are included in square modules. Take any D'-part P which contains $x(H_i^-)$. It is easy to see that $x(P)$ is $x(H_i^-)$. Moreover, P contains (entirely) the first $l-1$ levels of the BFS tree of H_i^-, rooted at x. Any vertex in $V(P)\setminus V(H_i^-)$ must be in the lth level of the tree and it must be adjacent to some vertex from the $(l-1)$st level. Since every $w_{j,i}$ is adjacent to exactly one vertex of another wing, each vertex of P which is not in H_i^- is not in any other wing. (Otherwise, C is 1-separable.) Thus each wing is contained in exactly one D'-part.

5. Triangular graphs. This section completes the proof of Theorem 1.1 by proving the following.

LEMMA 5.1. *H-decomposition is* NPC *if H is a triangular graph.*

The proof of Lemma 5.1 is a modification of Holyer's proof for triangles [10], where he presents a polynomial-time reduction of 3SAT to K_3-decomposition.

DEFINITION 5.1. *An instance of* 3SAT *is a set of clauses $C=\{C_1,\dots,C_r\}$ in variables $u_1,\dots,u_s$. Each clause C_i consists of three literals $l_{i,1}$, $l_{i,2}$, and $l_{i,3}$, where a literal $l_{i,j}$ is either a variable u_k or its negation $\overline{u_k}$. The problem is to determine whether C is satisfiable, that is, whether there exists a Boolean assignment to the variables which simultaneously satisfies all of the clauses in C. A clause is satisfied if one or more of its literals has value "true."*

The basic building block used for our reduction is a modified version of the Holyer graph $H_{3,p}$. For every $n\ge 3$ and $p\ge 3$, the Holyer graph is $H_{n,p}=(V_{n,p},E_{n,p})$, where

$$V_{n,p}=\left\{x=(x_1,\dots,x_n)\in\mathbb{Z}_p^n \ :\ \sum_{i=1}^{n}x_i\equiv 0\right\},$$

$$E_{n,p}=\{(x,y):\exists i,j \text{ such that } y_k\equiv x_k \text{ for } k\neq i,j \text{ and } y_i\equiv x_i+1, y_j\equiv x_j-1\},$$

and "$\equiv$" stands for congruence mod p (see Figure 11). Note that $H_{n,p}$ can be regarded as embedded in the $(n-1)$-dimensional torus $T^{n-1}=S^1\times\cdots\times S^1$ and that the local structure of $H_{n,p}$ is the same for each p. ($H_{n,p}$ should not be confused with the H-decomposition subgraph.)

LEMMA 5.2. *The graph $H_{3,p}$ has the following properties:*

1. *The degree of each vertex is* 6.
2. *The largest complete subgraph is a triangle.*

3. *The number of triangles which include each vertex is* 6.
4. *Each edge occurs in exactly two triangles.*
5. *Each two distinct triangles either are edge disjoint or have exactly one edge in common.*
6. *There are exactly two distinct edge partitions of $H_{3,p}$ into K_3's.*

All the above listed facts are trivial for $n = 3$. A proof in the wider frame of general n can be found in [10].

The two K_3 partitions of $H_{3,p}$ will be called "true" and "false" partitions, denoted by *T-partition* and *F-partition*. Two K_3's in $H_{3,p}$ are called neighbors if they have a common edge. Also, define a *patch* to be the subgraph of $H_{3,p}$ consisting of the vertices and edges of a particular K_3 and its neighbors. It is a T-patch if the central K_3 belongs to the T-partition, and it is an F-patch otherwise. Two patches P_1 and P_2 in $H_{3,p}$ are called *noninterfering* if the distance $d(x, y)$ in $H_{3,p}$ between vertices $x \in V(P_1)$ and $y \in V(P_2)$ is always at least 10.

Proof of Lemma 5.1 *for* 2*-connected graphs.* Let H be a 2-connected graph which is triangular by vertices v_0, v_1, and v_2 and components G_0, G_1, and G_2. Select v_0, v_1, v_2 and G_0, G_1, G_2 such that $|E(G_0)|$ is maximum among all possible triangular representations of H.

Let $I = \{C_1, \ldots, C_r\}$ in s variables $u_1, \ldots, u_s$, where each C_i consists of literals $l_{i,1}, l_{i,2}$, and $l_{i,3}$. We construct a graph $G_H(I)$ for which H-decomposition is equivalent to 3SAT on I.

Label each of the vertices of $H_{3,p}$ either v_0, v_1, or v_2 such that the labeling forms a proper 3-coloring and replace every edge (v_i, v_{i+1}) (mod 3) by a copy of the component G_i. Let all of these copies of G_0, G_1, and G_2 be disjoint except for the common original vertices of $H_{3,p}$. In the graph obtained, M_p, we have a copy of H replacing every triangle of $H_{3,p}$. Accordingly, M_p admits T and F H-decompositions and has its T- and F-patches (see Figure 11). Every H-subgraph of M_p obtained from a triangle of $H_{3,p}$ will be referred to as a K_3'.

Take a copy U_i of M_p to represent each variable u_i and copies $C_{l,1}$, $C_{l,2}$, and $C_{l,3}$ of M_p to represent each clause C_l. Join these copies of M_p together as follows: If $l_{i,j}$ is u_k, then identify an F-patch of $C_{i,j}$ with an F-patch of U_k. If $l_{i,j}$ is $\overline{u_k}$, then identify an F-patch of $C_{i,j}$ with a T-patch of U_k. Also, join $C_{i,1}$, $C_{i,2}$, and $C_{i,3}$ for each i by identifying one F-patch from each and then removing the edges of the central K_3'. Choose all patches which occur in a single M_p copy to be noninterfering. The graph obtained is $G_H(I)$.

If C is satisfiable, we partition $G_H(I)$ by partitioning U_k according to the truth of u_k in a satisfying assignment, choosing one "true" literal $l_{i,j}$ for each i and F-partitioning the corresponding $C_{i,j}$. On the other hand, suppose that there exists an H-decomposition D of $G_H(I)$. The maximality of G_0 implies $\{w, z\} \subset \{v_2\} \cup V(G_0)$ for each separating pair $\{w, z\}$ of H such that $w \in V(G_i)$ and $z \in V(G_j)$ $(i \neq j)$.

First, we assume that each of the three components of each K_3' is included in exactly one D-part. Thus it is enough to consider K_3-decompositions of $G_H'(I)$, obtained from $G_H(I)$ by replacing each such component by a simple edge.

Suppose that there is a K_3-decomposition of $G_H'(I)$, and consider a particular copy M of a modified M_p (that is, a copy of $H_{3,p}$) involved in the construction of $G_H(I)$. Take a D-part, say B, which is in M but not near any join. Clearly, the neighbors of B do not belong to D, the neighbors of neighbors of B do belong to D, and so on. Therefore, the edges of M, except perhaps those involved in joins, are all T-partitioned or all F-partitioned. Thus we may say that M is T- or F-partitioned.

Now suppose $l_{i,j}$ is u_k and consider the join between $C_{i,j}$ and U_k. We claim that the edges in the vicinity of this join are K_3-decomposable if and only if at least one of $C_{i,j}$ and U_k is T-partitioned. If (say) $C_{i,j}$ is T-partitioned, this accounts for all of the edges of $C_{i,j}$ near the joining patch except for those of the patch itself. The patch can be regarded as belonging to U_k, which can then be locally partitioned in either way. If, on the other hand, both $C_{i,j}$ and U_k are F-partitioned, a similar argument shows that the edges of the patch not belonging to the central K_3 are forced to belong to the F-partitions of both $C_{i,j}$ and U_k, which is a contradiction. Similarly, if $l_{i,j}$ is $\overline{u_k}$, then either $C_{i,j}$ is F-partitioned or U_k is T-partitioned.

Finally, we show that each of the three components of K_3' is included in exactly one D-part. Let S be the multiset of all SPs of all D-parts and consider one particular K_3', say A, in $G_H(I)$. We show that if a D-part does not include all components of K_3', then the total number of separating pairs in all D-parts (i.e., $|S|$) is too large. Obviously, S contains all SPs included in one of the three components of each K_3' in $G_H(I)$. Therefore, the only SPs which are not guaranteed to be in S are $\{v_2, w\}$, $w \in V(G_0) \setminus \{v_0, v_1\}$ for some K_3'-subgraph of $G_H(I)$. If there is no such SP in H, then a simple counting argument implies that each of the three components of A is contained in exactly one D-part. Hence assume that H contains $n > 2$ separating pairs, $\{v_2, w_1\}, \ldots, \{v_2, w_n\}$ such that $w_i \in G_0$ for $i = 1, \ldots, n$ ($v_0 = w_1$ and $v_1 = w_n$). If more than one D-part intersects the G_0 component of A, then there are more than $\binom{n}{2} + n - 2$ separating pairs in S, more than S could possibly contain. Therefore, G_0 intersects with exactly one D-part and each of G_0, G_1, and G_2 is included in exactly one D-part. □

To complete the proof of Theorem 1.1, we need only prove Lemma 5.1 for 1-separable graphs. H-decomposition of $G_H(I)$ is definitely implied by the satisfiability of I, exactly as described in the last proof. The other direction does not follow since the counting argument is highly dependent on 2-connectivity. To overcome this difficulty, we replace SSs by corresponding SPs. As in section 4.3, we define C, a 2-connected component of H. Lemma 4.10 then implies that H-decomposition of $G_H(I)$ contains a C-decomposition of a subgraph of $G_H(I)$, from which the satisfiability of I follows. □

6. Concluding remarks. The existence and structure of graph decompositions for various pairs (G, H) is a well-established branch of "classical" graph theory (see, e.g., [2] and [16] for a partial list of references), and the natural question regarding the computational complexity of the corresponding decision problems arose long ago. Despite substantial effort devoted to that issue, no general answer has been given until now.

The history of the problem solved by Theorem 1.1 is sketched in the introduction. It seems that the use of Wilson's theorem was the key step toward our general proof. Without it, the problem was settled only for complete graphs and a rather limited family of graphs H, all with $g(H) = 1$. Yet, also with Wilson's theorem at hand, many technical difficulties were tackled along the way.

Notice that our theorem does not completely settle the complexity status of all graph-decomposition problems. However, very recently, after the first version of this paper had already been submitted for publication, we heard that Bryś and Lonc [5] has presented a polynomial time-algorithm to decide H-decomposability whenever H has no component with more than two edges. Their presentation follows many partial results obtained during the last 15 years, most of which are listed in the introduction.

Apparently, graph decomposition presents a pattern which was already observed

in graph coloring, satisfiability, and many other graph-theoretical and combinatorial decision problems. The border between polynomial and NPC problems is crossed once a simple size parameter of the problem changes from 2 to 3. Unlike other combinatorial problems (e.g., graph isomorphism) whose complexity status is an open problem, graph decomposition, being in NPC, does not provide any consequences regarding the general complexity hierarchy. In that sense, our result brings no surprise. It just confirms the expected. It is surprising, however, how complicated the proof turned out to be, in comparison to NP-completeness proofs of other basic graph-theoretical problems. Of course, there might be a much simpler way to get to that result, a way that we — as well as others who have dealt with the problem since the early 1980s—have thus far missed.

REFERENCES

[1] N. Alon, *A note on the decomposition of graphs into isomorphic matchings*, Acta Math. Hungar., 42 (1983), pp. 221–223.

[2] J. C. Bermond and D. Sotteau, *Graph decomposition and G-designs*, in Proc. 5th British Combinatorial Conference, Aberdeen, 1975, Utilitas Math. Publishers, pp. 53–72.

[3] A. Bialostocki and Y. Roditty, $3K_2$*-decomposition of a graph*, Acta Math. Acad. Sci. Hungar., 40 (1982), pp. 201–208.

[4] A. E. Brouwer and R. M. Wilson, *The decomposition of graphs into ladder graphs*, zn 97/80, Stiching Mathematisch Centrum, Amsterdam, 1980.

[5] K. Bryś and Z. Lonc, *A complete solution of a Holyer problem*, in Proc. 4th Twente Workshop on Graph and Combinatorial Optimization, University of Twente, Enschede, The Netherlands, 1995.

[6] Y. Caro, *Decomposition and partition of trees into isomorphic subtrees*, M.Sc. thesis, Tel-Aviv University, Tel-Aviv, 1985.

[7] O. Favaron, Z. Lonc, and M. Truszczynski, *Decomposition of graphs into graphs with three edges*, Ars Combin., 20 (1985), pp. 125–146.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[9] I. Holyer, *The complexity of graph theory problems*, Ph.D. dissertation, Churchill College, Cambridge, UK, 1980.

[10] I. Holyer, *The NP-completeness of some edge partition problems*, SIAM J. Comput., 10 (1981), pp. 713–717.

[11] D. G. Kirkpatrick and P. Hell, *On the completeness of a generalized matching problem*, in Proc. 10th Annual ACM Symposium on Theory of Computing Machinery, ACM, New York, 1978, pp. 240–245.

[12] Z. Lonc, *Edge decomposition into copies of* SK_{12} *is polynomial*, J. Combin. Theory B, 69 (1997), pp. 164–182.

[13] J. Steiner, *Combinatorische aufgabe*, Z. Reine Angew. Math., 45 (1853), pp. 181–182.

[14] R. M. Wilson, *Decomposition of a Complete Graph into Subgraphs Isomorphic to a Given Graph*, Utilitas Mathematica Publishing, Winnipeg, MB, 1976, pp. 647–695.

[15] M. Preisler and M. Tarsi, *On the decomposition of graphs into copies of* $P_3 \bigcup tK_2$, Ars Combin., 35 (1993), pp. 325–333.

[16] J. Boska, *Graph Decompositions*, Springer-Verlag, Berlin, 1990.

[17] E. Cohen and M. Tarsi, *NP-completeness of graph-decomposition problems*, J. Complexity, 7 (1991), pp. 200–212.

[18] S. Masuyama and S. L. Hakimi, *Edge packing in graphs*, preprint.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1188–1207, August 1997

013

THE FOURTH MOMENT METHOD*

BONNIE BERGER†

Abstract. Higher moment analysis has typically been used to upper bound certain functions. In this paper, we introduce a new combinatorial method to *lower bound* the expectation of the absolute value of a random variable X by the expectation of a quartic in X. In the special case where we are looking at the absolute value of a (weighted) sum of $\{-1, +1\}$ unbiased random variables, we achieve tight bounds, using only a fourth moment, for the *total discrepancy* of a set system. Because the fourth moment depends only on 4-wise independence, our bounds will hold over polynomially sized distributions, and so these bounds will be directly applicable in removing randomness to obtain *NC* algorithms. We obtain the first *NC* algorithms for the problems of *total discrepancy, maximum acyclic subgraph, tournament ranking, the Gale–Berlekamp switching game,* and *edge discrepancy.* We show that for most of these applications it is truly *necessary* to consider a fourth moment by exhibiting a 3-wise independent distribution which does not achieve the required bounds. Our method is strong enough to give a new combinatorial bound on tournament ranking.

Key words. removing randomness, set discrepancy, Gayle–Berlekamp switching game, maximum acyclic subgraph

AMS subject classifications. 68Q22, 68R10, 05C20, 05C85

PII. S0097539792240005

1. Introduction.

In the design of parallel algorithms, randomness can often be used as a resource. Especially for parallel algorithms, however, generating many random bits can be expensive. We would like to be able to reduce the number of random bits that an algorithm needs to guarantee good performance—or even perhaps remove the randomness entirely. The key to the general methodologies for removing randomness from parallel algorithms [14, 17, 1, 18, 5, 3] involves showing that the analysis of the algorithm's good performance depends on *lesser independence.* This will allow us, for constant k-wise independent distributions, for example, to perform an exhaustive search [14, 17, 1] on a polynomially sized sample space.

For a large class of graph-theoretic randomized algorithms, the "goodness" of a particular solution is measured in terms of balance. More precisely, let $\mathcal{A} \subseteq 2^{\Gamma}$, $|\mathcal{A}| = |\Gamma| = n$, be a family of finite sets. Let $Y = \langle Y_1, \ldots, Y_n \rangle \in \{-1, +1\}^n$ be a 2-coloring of the underlying points in Γ (i.e., Y_i is the color of the ith element of Γ). Define $S_i(Y) = \sum_{j \in A_i} Y_j$, where $A_i \subseteq \mathcal{A}$. Then $\mathrm{disc}(Y) = \max_{1 \le i \le n} |S_i(Y)|$. The *set discrepancy* problem is to find a coloring Y such that $\mathrm{disc}(Y)$ is small. There is also a weighted version of the problem. Previous work has focused on upper bounding the expected maximum discrepancy of a set system over n sets; [2, 5, 19] give a good upper bound on the expected maximum discrepancy, using a $O(\log n)$th moment to do so. A good upper bound on the expected maximum discrepancy means that all sets are guaranteed to be nearly balanced, using only $O(\log n)$-wise independence. Since [2, 5, 19] also show how to derandomize algorithms whose analysis depends only on $O(\log n)$-wise independence, they obtain *NC* algorithms for set discrepancy, edge coloring, and lattice approximation.

On the other hand, problems such as tournament ranking take advantage of the fact that there will always be some positive discrepancy in each set. In this paper, we

*Received by the editors June 10, 1992; accepted for publication (in revised form) September 7, 1995. This research was supported by an NSF Postdoctoral Research Fellowship. A preliminary version of this paper appeared in *Proc. 2nd Annual ACM–SIAM Symposium on Discrete Algorithms*, SIAM, Philadelphia, 1991, pp. 373–383.

http://www.siam.org/journals/sicomp/26-4/24000.html

†Laboratory for Computer Science and Department of Applied Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139 (bab@theory.lcs.mit.edu).

will be concerned with problems of this type: where a good solution is constructed by taking advantage of probabilistic *imbalance.* We will be interested in the *total discrepancy* of a set system. This is similar to set discrepancy above, except that we want to find a coloring Y such that $\texttt{tot-disc}(Y) = \sum_{i=1}^{n} |S_i(Y)|$ is large. We address the weighted version of this problem in section 3. A good lower bound on the expected *sum* of the discrepancies of the individual sets implies that the expected disparity between wins and losses will be large. To achieve *NC* algorithms for these problems which use randomness to obtain large total discrepancy, we will give such a lower bound based on lesser independent distributions.

We introduce a new combinatorial method to bound the expectation of an absolute value of a random variable X from below by a fourth moment. In the special case where we are looking at the absolute value of a (weighted) sum of $\{-1, +1\}$ unbiased random variables, we achieve a lower bound, using a fourth moment, for the *total discrepancy* of a set system. (It is easy to show that our bound is tight using only a second moment; see Lemma 3.3.) Since a fourth moment requires only 4-wise independence, this will allow us to use parallel derandomization techniques. Our lower bound for total discrepancy will match the previously well-known lower bound which requires n-wise independence [26].

We obtain the first *NC* algorithm for lower bounding total discrepancy, and this in turn will give us the first *NC* algorithms for the *Gale–Berlekamp switching game* and *edge discrepancy.* We also apply our fourth moment inequality to devise the first *NC* approximation algorithm for getting strictly more than half the arcs in the *maximum acyclic subgraph* problem and *tournament ranking.* These problems will be defined later in section 5. Although our algorithms for these problems are discrepancy-like in flavor, dependencies between random variables will prevent us from simply applying our discrepancy lower bounds. Instead, these problems will require a more involved application of our fourth moment method. The maximum acyclic subgraph problem, or its alternate formulation as the *feedback arc set* problem, arises in many applications, including machine-shop scheduling [16], generation of test vectors for circuits [24], register minimization in VLSI design [24], feedback minimization in systems [22], and circuit simulation through functional abstraction [15].

Finally, we show that it is truly *necessary* to consider a fourth moment. We exhibit a discrepancy problem and a 3-wise independent distribution for which we get very small expected total discrepancy. In fact, surprisingly, every point in the sample space of this 3-wise independent distribution yields very small total discrepancy. This shows that for lower bounding total discrepancy, 4-wise independence is both necessary and sufficient.

This threshold behavior for k-wise independence is particularly interesting since it shows that there are natural problems, which need only constant-wise independence but for which Luby's [18] parallel binary search procedure for pairwise and 3-wise independent sample spaces can be proved to be too weak. Parallel binary search techniques can sometimes be preferable to exhaustive search, even over constant-wise independent distributions, because exhaustive search can sometimes blow up the number of processors needed by the algorithm. Therefore, in order to obtain more efficient *NC* algorithms over constant-wise independent distributions, our result shows that we require more powerful methods to perform binary search in parallel, such as those found in [2, 5, 19].

A preliminary version of this paper appeared in [4].

2. The method. In this section, we present a new method for lower bounding the expectation of an absolute value with a fourth moment. Suppose we have an arbitrary random variable S and we are looking for a lower bound on $E[|S|]$. The argument is complicated by the fact that we are taking an expectation of an absolute value. In order to make things more manageable, we might consider S^2 in place of $|S|$. But then the problem arises that if $|S|$ is a little more than its expected value, then the contribution that S^2 makes to its expected value is proportionally much larger. This means that even if the expectation of S^2 is known to be large, we cannot say much about $E[|S|]$. Our solution will be to find a $c > 0$ for which $f(S) = c(S^2 - S^4/q) \leq |S|$ (where $q > 0$). Intuitively, we compensate for S^2 growing large by subtracting off a multiple of S^4, which grows even faster. We show that we can choose our constants c and q so that if $E[f(S)]$ is large, then $E[|S|]$ is large. This notion is captured in the following lemma.

LEMMA 2.1. *For all $X \geq 0$ and for all $q > 0$*

$$X \geq \frac{3\sqrt{3}}{2\sqrt{q}} \left(X^2 - \frac{X^4}{q} \right).$$

Proof. We want to find a c such that $X \geq (1/c\sqrt{q})(X^2 - X^4/q)$. Equivalently, we want a $c \geq (qX - X^3)/q^{3/2}$. If we differentiate the right-hand side of this equation to find the maximum value, we find that it occurs when $X = \sqrt{q/3}$. Plugging in this X, we get that the maximum value is $2/(3\sqrt{3})$. So we can set c to be this constant. Now plugging in for c, we get that

$$X \geq \frac{3\sqrt{3}}{2\sqrt{q}} \left(X^2 - \frac{X^4}{q} \right),$$

as claimed. □

THEOREM 2.2. *For any random variable S and for all $q > 0$,*

$$E[|S|] \geq \frac{3\sqrt{3}}{2\sqrt{q}} \left(E[S^2] - \frac{E[S^4]}{q} \right).$$

Proof. The proof follows from Lemma 2.1 by linearity of expectation. □

A special case of this theorem is particularly interesting, and we give it in the following corollary.

COROLLARY 2.3 (see [9, p. 194]). *For any random variable S,*

$$E[|S|] \geq \frac{E[S^2]^{3/2}}{E[S^4]^{1/2}}.$$

Proof. Plugging $q = 3E[S^4]/E[S^2]$ into Theorem 2.2, we obtain the desired result. Alternatively, Furedi has pointed out that this special-case form of the result can be proved as a consequence of Hölder's inequality [12, p. 140]. □

3. Tight bounds for total discrepancy using 4-wise independence. We are interested in lower bounding the expected total discrepancy of a family of sets using less independence. Weighted total discrepancy is similar to the unweighted case except that for each set $A_i \subseteq \mathcal{A}$, we let $\epsilon_{11}, \ldots, \epsilon_{nn} \in \mathbf{R}^1$ be the weights on its elements, where

[1] We can also extend these methods to obtain similar results for $\epsilon_{ij} \in \mathbf{C}$.

$\epsilon_{ij} = 0$ if $j \notin A_i$, and we define $S_i = S_i(Y) = \epsilon_{i1}Y_1 + \cdots + \epsilon_{in}Y_n$. The *weighted total discrepancy* problem is to find a coloring Y such that $\texttt{wt-tot-disc}(Y) = \sum_{i=1}^{n} |S_i|$ is large.

We apply the fourth moment method as given in section 2 to obtain a tight bound on total discrepancy in the case when the Y_i are chosen 4-wise independently at random. *NC* algorithms resulting from our bounds appear in section 5.2. First, we bound the expected discrepancy of a single set, A, since we can then use linearity of expectation to get a bound on the total discrepancy. Without loss of generality, suppose A has n elements.

LEMMA 3.1. *Let $Y_1, \ldots, Y_n$ be 4-wise independent variables, each being 1 or -1 with equal probability, and let $\epsilon_1, \ldots, \epsilon_n \in \mathbf{R}$. If $S = \epsilon_1 Y_1 + \cdots + \epsilon_n Y_n$, then*

$$E[|S|] \geq \sqrt{\frac{\epsilon_1^2 + \cdots + \epsilon_n^2}{3}}.$$

In the special case of unweighted discrepancy, when $\epsilon_i \in \{-1, +1\}$, then

$$E[|S|] \geq \sqrt{\frac{n}{3}}.$$

Proof. To get a lower bound on $E[|S|]$ using Theorem 2.2, we need to compute $E[S^2]$ and $E[S^4]$.

Before we compute the expected values, let us note the following facts:

1. $E[Y_i^t]$ is 0 when t is odd and 1 when t is even.
2. $E[Y_{i_1} \cdots Y_{i_t}] = E[Y_{i_1}] \cdots E[Y_{i_t}]$ when $t \leq 4$.

For notational convenience, henceforth in this paper, when we sum over an index i, we mean $i = 1, \ldots, n$, and when we sum over indices $i_1, \ldots, i_t$, we assume that they are all different.

Observe that

$$E[S^2] = \sum_i \epsilon_i^2 E[Y_i^2] + \sum_{i \neq i'} \epsilon_i \epsilon_{i'} E[Y_i Y_{i'}] = \sum_i \epsilon_i^2. \tag{3.1}$$

Also, observe that

$$\begin{aligned} E[S^4] &= \sum_i \epsilon_i^4 E[Y_i^4] + 4 \sum_{i,i'} \epsilon_i^3 \epsilon_{i'} E[Y_i^3 Y_{i'}] + 3 \sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2 E[Y_i^2 Y_{i'}^2] \\ &\quad + 6 \sum_{i,i',i''} \epsilon_i^2 \epsilon_{i'} \epsilon_{i''} E[Y_i^2 Y_{i'} Y_{i''}] + \sum_{i,i',i'',i'''} \epsilon_i \epsilon_{i'} \epsilon_{i''} \epsilon_{i'''} E[Y_i Y_{i'} Y_{i''} Y_{i'''}] \\ &= \sum_i \epsilon_i^4 + 3 \sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2. \end{aligned} \tag{3.2}$$

Plugging these values for $E[S^2]$ and $E[S^4]$ into Theorem 2.2 and choosing $q = (3\sum_i \epsilon_i^4 + 9\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2)/\sum_i \epsilon_i^2$, we get that

$$E[|S|] \geq \frac{3\sqrt{3}}{2\sqrt{\frac{3\left(\sum_i \epsilon_i^4 + 3\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2\right)}{\sum_i \epsilon_i^2}}} \left(\sum_i \epsilon_i^2 - \frac{\sum_i \epsilon_i^4 + 3\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2}{\frac{3\left(\sum_i \epsilon_i^4 + 3\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2\right)}{\sum_i \epsilon_i^2}} \right)$$
$$= \frac{3\sqrt{\sum_i \epsilon_i^2}}{2\sqrt{\sum_i \epsilon_i^4 + 3\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2}} \left(\frac{2\sum_i \epsilon_i^2}{3} \right)$$
$$= \frac{(\sum_i \epsilon_i^2)^{3/2}}{(\sum_i \epsilon_i^4 + 3\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2)^{1/2}}.$$

Squaring the first and last parts of this equation, we get that

$$E[|S|]^2 \geq \frac{(\sum_i \epsilon_i^2)^3}{\sum_i \epsilon_i^4 + 3\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2} = \frac{(\sum_i \epsilon_i^2)(\sum_i \epsilon_i^4 + \sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2)}{\sum_i \epsilon_i^4 + 3\sum_{i,i'} \epsilon_i^2 \epsilon_{i'}^2} \geq \frac{\sum_i \epsilon_i^2}{3}. \qquad \square$$

By linearity of expectation, we obtain the following theorem, which provides a lower bound on the expected total discrepancy.

THEOREM 3.2. *Let $Y_1, \ldots, Y_n$ be 4-wise independent variables, each being 1 or -1 with equal probability. Then the expected total discrepancy*

$$E\left[\sum_{i=1}^n |S_i|\right] \geq \frac{\sum_{i=1}^n \sqrt{\epsilon_{i1}^2 + \cdots + \epsilon_{in}^2}}{\sqrt{3}}.$$

In the case of complete independence of the Y_i's, it has been shown [26] that the expected total discrepancy $E\left[\sum_{i=1}^n |S_i|\right] \geq \sqrt{2/\pi} \sum_{i=1}^n \sqrt{\epsilon_{i1}^2 + \cdots + \epsilon_{in}^2}$; that is, this gives a constant of $\sqrt{2/\pi}$ versus $1/\sqrt{3}$ in the theorem above.

The bounds in the preceding lemma and theorem are tight up to constant factors, as we show in the following lemma.

LEMMA 3.3. *Let $Y_1, \ldots, Y_n$ be pairwise independent variables, each being 1 or -1 with equal probability, and let $\epsilon_1, \ldots, \epsilon_n \in \mathbf{R}$. If $S = \epsilon_1 Y_1 + \cdots + \epsilon_n Y_n$, then*

$$E[|S|] \leq \sqrt{\epsilon_1^2 + \cdots + \epsilon_n^2}.$$

Proof. The proof follows simply from the fact that $E[X]^2 \leq E[X^2]$ for any random variable X and from the bound in equation (3.1). $\square$

In Theorem 3.2, we showed that the expected total discrepancy is large using only 4-wise independence. In what follows, we will show a slightly stronger result, namely, that the discrepancy is large with positive probability. Our proof will require the following inequality due to Cantelli [23], which achieves a slightly better bound than Chebyshev's inequality.

PROPOSITION 3.4 (Cantelli). *If X is a random variable with mean μ and variance σ^2, then for all $r \geq 0$,*

$$\Pr[X < \mu - r\sigma] \leq \frac{1}{r^2 + 1}.$$

By combining Cantelli's inequality with Lemma 3.1, we can directly show that there exists an $\alpha > 0$ such that $\Pr[|S| \geq \alpha\sqrt{\epsilon_1^2 + \cdots + \epsilon_n^2}]$ is bounded from below

by a constant. However, we can prove a more general result by combining Cantelli's inequality with fourth moment analysis, as in the following lemma.

LEMMA 3.5. *For any random variable S and $0 < \alpha < 1$,*

$$\Pr\left[\,|S| \geq \alpha\sqrt{E[S^2]}\,\right] \;\geq\; \frac{(1-\alpha^2)^2}{(1-\alpha^2)^2 + \frac{E[S^4]}{E[S^2]^2} - 1}.$$

Proof. We apply Cantelli's inequality (Proposition 3.4) to show that

$$\Pr\left[\,|S| \geq \alpha\sqrt{E[S^2]}\,\right] \;=\; \Pr\left[\,S^2 \geq \alpha^2 E[S^2]\,\right] \;\geq\; \frac{r^2}{r^2+1},$$

where $r = (1-\alpha^2)E[S^2]/\sqrt{E[S^4] - E[S^2]^2}$. Simplifying, this yields the desired bound. □

THEOREM 3.6. *Let $Y_1, \ldots, Y_n$ be 4-wise independent variables, each being 1 or -1 with equal probability, let $\epsilon_1, \ldots, \epsilon_n \in \mathbf{R}$, and let $0 < \alpha < 1$. If $S = \epsilon_1 Y_1 + \cdots + \epsilon_n Y_n$, then*

$$\Pr\left[\,|S| \geq \alpha\sqrt{\epsilon_1^2 + \cdots + \epsilon_n^2}\,\right] \;\geq\; \frac{(1-\alpha^2)^2}{(1-\alpha^2)^2 + 2}.$$

Proof. Use Lemma 3.5 and the fact that from equations (3.1) and (3.2), we have that $E[S^2] = \epsilon_1^2 + \cdots + \epsilon_n^2$ and $E[S^4] \leq 3(\epsilon_1^2 + \cdots + \epsilon_n^2)^2$. □

We set $\alpha = 1/\sqrt{3}$ to get the following corollary.

COROLLARY 3.7.

$$\Pr\left[\,|S| \geq \frac{\sqrt{\epsilon_1^2 + \cdots + \epsilon_n^2}}{\sqrt{3}}\,\right] \;\geq\; \frac{2}{11}.$$

We note that Theorem 3.6, when used directly in lower bounding the expected total discrepancy, does not yield as good a constant as we obtained in Theorem 3.2 above, i.e., we get $1/3$ versus the $1/\sqrt{3}$ achieved above. However, Corollary 3.7 will be useful later when we consider the problem of finding a sample point for which a constant fraction of the sets in a set system have large discrepancy.

4. Fourth moments are necessary. We now show that a fourth moment is actually necessary to achieve a lower bound on total discrepancy. We do this by exhibiting a discrepancy problem and a 3-wise independent distribution for which we get very small expected total discrepancy. In fact, every point in the sample space of this 3-wise independent distribution yields very small total discrepancy. Note that this also shows pairwise independence is not sufficient for lower bounding total discrepancy.

Distribution $\mathcal{D}$ (also see [18]) is as follows. Let $n = 2^{l-1}$ and $a_i = \langle i_1, \ldots, i_{l-1}, 1\rangle$, where $\text{bin}(i) = \langle i_1, \ldots, i_{l-1}\rangle$ is the binary expansion of i. Let $\omega = \langle \omega_1, \ldots, \omega_l\rangle$ be picked uniformly from Z_2^l. Define random variables $X_0, \ldots, X_{n-1}$ such that

$$X_i = a_i \cdot \omega.$$

CLAIM 4.1. *Distribution $\mathcal{D}$ is 3-wise independent.*

Proof. It is fairly easy to show (see [2]) that X_{i_1}, X_{i_2}, and X_{i_3} are independent and unbiased iff a_{i_1}, a_{i_2}, and a_{i_3} are linearly independent as vectors over Z_2. Assume that X_{i_1}, X_{i_2}, and X_{i_3} are dependent. Then

$$\alpha_1 a_{i_1} + \alpha_2 a_{i_2} + \alpha_3 a_{i_3} = 0,$$

where $\alpha_1, \alpha_2, \alpha_3 \in Z_2$ are not all 0. Consequently, an even number of α's are 1 (since in distribution $\mathcal{D}$ the last bit of a_i is always 1). In addition, there cannot be exactly two α's that are 1: the a_i's being distinct implies that $a_{i_j} + a_{i_k} \neq 0$. Therefore, $\alpha_1, \alpha_2, \alpha_3 = 0$, which is a contradiction. Thus X_{i_1}, X_{i_2}, and X_{i_3} are independent and unbiased. □

Observe that distribution $\mathcal{D}$ is not 4-wise independent; in particular, X_4, X_5, X_6, and X_7 are dependent since $X_7 = X_4 + X_5 + X_6$.

THEOREM 4.1. *For all $i, j \in \{0, \ldots, n-1\}$, define $\epsilon_{ij} = (-1)^{a_i \cdot a_j}$.*[2] *Let $S_i = \epsilon_{i0} Y_0 + \cdots + \epsilon_{i,n-1} Y_{n-1}$, where $Y_j = (-1)^{X_j}$ and the X_j's are chosen from distribution $\mathcal{D}$. Then for any $X = \langle X_0, \ldots, X_{n-1} \rangle$ in the sample space,*

$$\sum_{i=1}^{n} |S_i| \;=\; n,$$

i.e., the total discrepancy is very small.

Proof. Pick an arbitrary ω. Let $\omega' = \langle \omega_1, \ldots, \omega_{l-1} \rangle$. Note that if $a_j \cdot \omega = 0$, then $Y_j = 1$, and if $a_j \cdot \omega = 1$, then $Y_j = -1$. By definition,

$$\begin{aligned}
\sum_i |S_i| &= \sum_i \left| \sum_j \epsilon_{ij} Y_j \right| \\
&= \sum_i \left| \sum_j (-1)^{a_i \cdot a_j} (-1)^{a_j \cdot \omega} \right| \\
&= \sum_i \left| \sum_j (-1)^{a_j \cdot (a_i + \omega)} \right|.
\end{aligned}$$

Observe that $a_i + \omega$ has a 1 in exactly those positions where a_i and ω differ.

CLAIM 4.2. $\text{bin}(i) = \omega'$ *implies that* $|S_i| = n$.

Proof. Since $\text{bin}(i) = \omega'$ and for all i, $a_{il} = 1$, $a_i + \omega = \vec{0}$ when $\omega_l = 1$ and $\langle 0, \ldots, 0, 1 \rangle$ when $\omega_l = 0$. Then $a_j \cdot (a_i + \omega) = 0$ when $\omega_l = 1$ and 1 when $\omega_l = 0$, which implies that $|S_i| = n$ in either case. □

CLAIM 4.3. *For all i such that* $\text{bin}(i) \neq \omega'$, $|S_i| = 0$.

Proof. Since $\text{bin}(i) \neq \omega'$, a_i and ω differ in at least one position among $1, \ldots, l-1$. Half of the a_j's have an even number of 1's in positions where a_i and ω differ and half of the a_j's have an odd number of 1's in these positions. Hence $|S_i| = 0$. □

By Claims 4.2 and 4.3 and the fact that exactly one i has $\text{bin}(i) = \omega'$, we get that $\sum_i |S_i| = n$, thereby proving the theorem. □

Note that the bound in the theorem is the worst possible because it is always easy to get a total discrepancy of n by making the discrepancy of a single set be n.

As a simple consequence Theorem 4.1, we obtain a very small expected total discrepancy over a 3-wise independent distribution. Note that any cubic will have

[2] The matrix $\{\epsilon_{ij}\}$ is an example of a Hadamard matrix.

the same expected value under any 3-wise independent distribution. Thus as a consequence of Theorem 4.1, there is no cubic that will give a good lower bound on the expected total discrepancy, i.e., there is no analogous "third moment method" for lower bounding total discrepancy. This indicates that there is a threshold of independence where we need to go to 4-wise independence to get an adequate lower bound for total discrepancy. Since Luby [18] could only handle up to 3-wise independence efficiently in parallel, this serves to further motivate the need for the general framework of [5] to remove randomness in the case of more than 3-wise (and up to polylogarithmic) independence.

5. *NC* algorithms.

5.1. Background on removing randomness. For many applications [26, 21, 14, 17, 1, 18, 5, 3], the problem of removing randomness from an algorithm can be solved by finding an $\hat{X} = \langle X_1, \ldots, X_n \rangle$ such that $F(\hat{X}) \geq E[F(X)]$, for some function F which measures the "goodness" of a sample point, and some sample space $\mathcal{S}$ over which the expectation is to be computed. The problem is then how best to find a good sample point (e.g., an $\hat{X}$ such that $F(\hat{X}) \geq E[F(X)]$) in $\mathcal{S}$. If the space of sample points is small (e.g., polynomial), then this can be accomplished by brute force [14, 17, 1]; namely, we could try all points until we get a good one, for one must exist. Although a fully independent sample space is large, polynomial-size sample spaces for 4-wise independent distributions exist.

PROPOSITION 5.1 (see [1]). *For any constant k, there exists a k-wise independent distribution over $X_1, \ldots, X_n$, where the X_i's are unbiased binary random variables, with only $O(n^{\lfloor k/2 \rfloor})$ sample points.*

PROPOSITION 5.2 (see [13, 17, 1]). *For any constant k, there exists a k-wise independent distribution over $X_1, \ldots, X_n$, where the X_i's are uniform in $[0, n-1]$, with only $O(n^k)$ sample points.*

To say that our inequality depends only on 4-wise independence means that it will hold for any 4-wise independent distribution. Therefore, there is an $O(n^4)$ ($O(n^2)$ for unbiased binary random variables) size sample space which we can search exhaustively in *NC* by assigning a processor for each sample point. Notice, however, that we get a blowup in the number of processors, as compared to the *RNC* algorithm, by a factor of the number of sample points.

Alternatively, as long as S is a sum of a polynomial number of functions of at most $\log n$ variables each, we could also use the Berger–Rompel [5] general framework to remove randomness from functions of the form $B(S) = (3\sqrt{3}/(2\sqrt{q}))(S^2 - S^4/q)$, where $q > 0$. This is because raising S to the fourth or second power yields a sum of a polynomial number of functions of at most four times as many variables as before. The general framework can also handle a sum of a polynomial number of functions $B(S)$. Note, however, that the general framework cannot be used to handle functions of the special-case form given in Corollary 2.3.

We discuss the tradeoff between using exhaustive search to derandomize versus the more complicated methods of the general framework of [5] for the specific case of derandomizing total discrepancy in section 5.2 and for the case of maximum acyclic subgraph in section 5.4.2.

5.2. Total discrepancy is in *NC*. Theorem 3.2 immediately gives us an *RNC* algorithm for total discrepancy where if we pick the Y_j's 4-wise independently with

$\{-1,+1\}$ equally likely probability, we get a solution with expected total discrepancy

$$E\left[\sum_{i=1}^{n}|S_i|\right] \;\geq\; \frac{\sum_{i=1}^{n}\sqrt{\epsilon_{i1}^2+\cdots+\epsilon_{in}^2}}{\sqrt{3}}.$$

To get an *NC* algorithm, we can either try all sample points in a 4-wise independent sample space or do parallel binary search, as discussed in section 5.1. Which method is better depends on Δ, the maximum set size.

Trying all sample points has a processor blowup of $O(n^2)$ for each sample point, an additional factor of $O(\Delta)$ processors to sum over the at most Δ elements of each set, and another factor of $O(n)$ processors to sum over the n sets. This gives an *NC* algorithm which uses $O(n^3\Delta)$ processors, runs in $O(\log n)$ time, and returns a solution with total discrepancy

$$\sum_{i=1}^{n}|S_i| \;\geq\; \frac{\sum_{i=1}^{n}\sqrt{\sum_{j\in A_i}\epsilon_{ij}^2}}{\sqrt{3}},$$

which, of course, is equivalent to the lower bound that we got for the *RNC* algorithm.

On the other hand, we could do a parallel binary search, as implemented by the general framework of [5]. Using our fourth moment inequality, the total discrepancy can be expressed as a sum of $O(n\Delta^4)$ functions, each depending on at most four random variables. This gives an *NC* algorithm which uses $O(n\Delta^4)$ processors, runs in $O(\log n)$ time, and returns a solution with the same total discrepancy as exhaustive search.

In summary, if $\Delta \geq n^{2/3}$, exhaustive search is better since it uses fewer processors, and parallel binary search is better otherwise.

Remark. Using 4-wise independence, we can also find in *NC* a single setting of the Y_j's such that the total discrepancy $\sum_{i=1}^{n}|S_i|$ is $\Theta(n^{3/2})$, i.e., this single setting meets both the upper and lower bounds.

Remark. These methods can be extended to get a sample point that has a constant fraction of the individual discrepancies large as well by using methods similar to Corollary 3.7.

5.3. The Gale–Berlekamp switching game and edge discrepancy are in *NC*. Our methods can be applied to get the first *NC* algorithms for the Gale–Berlekamp switching game and the edge discrepancy problem.

In the *Gale–Berlekamp switching game,* we want an algorithm which, given an $n\times n$ matrix of $a_{ij}\in\{-1,+1\}$, outputs an n-vector of $X_i\in\{-1,+1\}$ and an n-vector of $Y_i\in\{-1,+1\}$ so that

$$\sum_{i,j=1}^{n} a_{ij}X_iY_j \;\geq\; cn^{3/2}$$

for some constant $c>0$.

Brown and Spencer [7, 26] gave a deterministic polynomial-time algorithm for this problem (for $c=\sqrt{2/\pi}$) by picking the Y_j's n-wise independently at random and then utilizing the method of conditional probabilities to derandomize. We, on the other hand, will pick the Y_j's 4-wise independently with $\{-1,+1\}$ equally likely probability and then derandomize using parallel methods.

The proof of our algorithm's correctness follows along the same lines of Brown and Spencer's [7, 26] proof for the sequential case.

LEMMA 5.3. *Picking $Y_1, \ldots, Y_n$ 4-wise independently, each 1 or -1 with equal probability, we obtain an RNC algorithm for the Gale–Berlekamp switching game, where $c = 1/\sqrt{3}$ in the solution.*

Proof. Let $R_i = \sum_{j=1}^{n} a_{ij} Y_j$. Set $X_i = 1$ if $R_i \geq 0$ and -1 otherwise. Then

$$E\left[\sum_{i,j=1}^{n} a_{ij} X_i Y_j\right] = E\left[\sum_{i=1}^{n} R_i X_i\right] = E\left[\sum_i |R_i|\right] = E\left[\sum_i \left|\sum_j a_{ij} Y_j\right|\right] \geq \frac{n^{3/2}}{\sqrt{3}}$$

by Theorem 3.2 if we let $\epsilon_{ij} = a_{ij} \in \{-1, +1\}$. □

To get an *NC* algorithm, trying all sample points works best in this case since $\Delta = n$ (see the discussions in sections 5.1 and 5.2). Thus we get the following theorem.

THEOREM 5.4. *There is an NC algorithm for the Gale–Berlekamp switching game which uses $O(n^4)$ processors, runs in $O(\log n)$ time, and returns a solution where $c = 1/\sqrt{3}$.*

Subsequent to our work, a higher-dimensional version of the switching game result was obtained by Tetali [27].

In the *edge discrepancy* problem, we are given an edge coloring $\chi : E \to \{-1, +1\}$ of $K_n = (V, E)$ and we want to find a subset of the vertices $S \subseteq V$ so that

$$\left|\sum_{e \in E_S} \chi(e)\right| \geq cn^{3/2},$$

where E_S are the edges of the subgraph induced by S.

Erdös and Spencer discovered a polynomial-time algorithm for edge discrepancy through a reduction to the Gale–Berlekamp switching game (see [10] and [26, pp. 47–48]). This reduction, which can easily be done in *NC*, in conjunction with Theorem 5.4 gives us the following theorem.

THEOREM 5.5. *There is an* NC *algorithm for the edge discrepancy problem which uses $O(n^4)$ processors, runs in $O(\log n)$ time, and returns a solution where $c = 1/(12\sqrt{3})$.*

5.4. The maximum acyclic subgraph problem and tournament ranking are in *NC*. We can also apply our methods to obtain the first *NC* algorithm that gets strictly greater than half the arcs for the *maximum acyclic subgraph* problem, and *tournament ranking* as a special case. The maximum acyclic subgraph problem is as follows: given a directed graph $G = (V, A)$, where $|V| = n$ and $|A| = m$, find the largest subset of the arcs which does not contain a cycle. This problem is *NP*-complete, so we settle for approximate solutions. (See [6] for more background on the problem.)

Berger and Shor [6] showed how to handle 2-cycles optimally and therefore reduced the problem to finding algorithms for 2-cycle-free graphs. So we henceforth consider only such graphs. Furthermore, they devised randomized and deterministic polynomial-time algorithms and an *RNC* algorithm (which we describe below) and proved that these algorithms find an acyclic subgraph $\hat{A} \subseteq A$ with at least $|A|/2 + \Omega(\sum_{i=1}^{n} \sqrt{\deg(i)})$ arcs, where $\deg(i)$ is the degree of vertex i in G.

Using Theorem 2.2 and a more delicate analysis than we used for lower bounding the discrepancy of a set, we are able to prove that the Berger–Shor *RNC* algorithm

works using 5-wise (versus n-wise in [6]) independence. Then the parallel derandomization techniques can be applied. Moreover, our proof techniques, which utilize the fourth moment method, give the following improved lower bound on the size of the acyclic subgraph returned by our *NC* algorithm:

$$\frac{|A|}{2} - 1 + \frac{\sqrt{6}}{40}\sum_{i=1}^{n}\sqrt{\deg(i)} + \frac{\sqrt{3}}{20}\sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|,$$

where $d_{\text{out}}(i)$ is the out-degree of vertex i in G and $d_{\text{in}}(i)$ is the in-degree of i. This bound is tight, within a constant factor on the low-order terms [6].

We remark that if $\deg(i) = |V| - 1$, for all i, an acyclic subgraph corresponds simply to a tournament ranking. A *tournament ranking* is an ordering of players from "best" to "worst," where we maximize the number of pairs of players A and B for which player A has beaten player B and we have assigned A a higher rank than B. Our acyclic subgraph result generalizes and improves upon the corresponding $|A|/2 + \Omega(\sum_{v\in V}\sqrt{|V|})$ bound for tournaments discovered by Spencer [25, 26]. (A sequential algorithm that achieves this latter bound was first presented in [20].) The improved bound that we get is a natural consequence of our methods, which are constructive.

Let $G = (V, A)$, $|V| \geq 2$, be a tournament. We are able to find in *NC* a ranking of the players so that there are at least

$$\frac{|A|}{2} - 1 + \frac{\sqrt{3}}{40}|V|^{3/2} + \frac{\sqrt{3}}{20}\sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|$$

wins of higher-ranked players over lower ranked ones. Notice that the last term results in an improved ranking when there are players who have an unbalanced number of wins and losses. This bound is tight within a constant factor on the low-order terms for tournaments [8].

5.4.1. An improved *RNC* algorithm which uses 5-wise independence. Berger and Shor [6] gave the following *RNC* approximation algorithm for the maximum acyclic subgraph problem in 2-cycle free graphs. Suppose that we have a processor for each vertex and each arc in G and an incoming and outgoing adjacency list in shared memory. Then once we have assigned a randomized ordering to the vertices of G, we can process all of them in parallel. To produce a randomized ordering, randomly assign each vertex i a rank $r_i \in \{1, \ldots, \sigma\}$, $\sigma \in \mathbf{N}$ (we will discuss possible choices for σ later). When we process a vertex, we will only consider its neighbors with a higher number in the assigned ordering. We will place in the acyclic subgraph $\hat{A}$ either all of the out-arcs to or all of the in-arcs from the higher-numbered neighbors, whichever set is larger. It can easily be seen that this procedure can be handled with a linear number of processors in logarithmic time using standard parallel algorithm and data structure techniques.

However, this randomized algorithm is not in a form to which any of the known parallel derandomization procedures can be applied; its analysis is based on that of its sequential counterpart, whose analysis depends on n-wise independence. Moreover, Greenlaw [11] has shown that if the algorithm is modified to process vertices in order of highest-degree vertex in the remaining graph, then the problem of finding the acyclic subgraph produced by this algorithm is *P-complete.*

In this paper, we will show that if we assign the randomized ordering 5-wise independently (as opposed to n-wise independently), we will be able to obtain an even

better lower bound on $E[|\hat{A}|]$, the expected number of arcs in the acyclic subgraph, than was achieved through the sequential case analysis in [6]. The key point is that we will then be able to derandomize this *RNC* algorithm to get an *NC* algorithm.

For $\langle r_1, \ldots, r_n \rangle \in \{1, \ldots, \sigma\}^n$ chosen from a 5-wise independent distribution, we wish to show that $E[|\hat{A}|]$ is large. It will be convenient for us to define the following random variables. Let Y_{ij} be the indicator random variable for $r_j > r_i$, i.e., Y_{ij} is 1 if $r_j > r_i$ and 0 otherwise. Furthermore, let Z_{ij} be $+Y_{ij}$ if $\langle i, j \rangle \in A$ and $-Y_{ij}$ if $\langle j, i \rangle \in A$. (Note that + and – are fixed by direction.) Let

$$X_i = \left| \sum_{j \in N(i)} Z_{ij} \right|.$$

Intuitively, X_i is the discrepancy, or difference in the number of incoming and outgoing arcs, at the ith vertex when it is processed. Let $d_i = \sum_{j \in N(i)} Y_{ij}$. So d_i is the degree of the ith vertex when it is processed. Let S denote the number of arcs that have both of their endpoints with the same rank. Thus $S = \sum_{<i,j> \in A} s_{ij}$, where s_{ij} is 1 if $r_i = r_j$ and 0 otherwise.

For the purposes of analysis, suppose that the algorithm breaks ties between ranks by a random permutation on all of the ranks once they are chosen. Call this algorithm Aperm and the original algorithm A5wise. Then the sizes of the acyclic subgraphs (i.e., the $|\hat{A}|$'s) produced by the two algorithms will differ by at most the number of arcs connecting nodes of equal rank. Thus the expected difference in the $|\hat{A}|$'s produced by the two algorithms is at most the expected number of arcs connecting nodes of equal rank; in particular, the expected behavior of A5wise is at least the expected behavior of Aperm minus this difference.

More formally, let us first examine the expected behavior of Aperm, that is, the case when all the ranks are distinct. Expanding $|\hat{A}|$ out in terms of the random variables defined above, we get

$$|\hat{A}| = \sum_{i=1}^{n} \left(\frac{d_i}{2} + \frac{X_i}{2} \right) = \frac{1}{2}|A| + \frac{1}{2} \sum_{i=1}^{n} X_i.$$

By linearity of expectation,

$$E[|\hat{A}|] = \frac{1}{2}|A| + \frac{1}{2} \sum_{i=1}^{n} E[X_i]. \tag{5.1}$$

We would now like to prove that $\sum_{i=1}^{n} E[X_i]$ is large. Observe that X_i is the absolute value of a sum of random variables Z_{ij}. So as in section 2, we will bound its expectation from below by a fourth moment. Applying Theorem 2.2, we have that for all $q > 0$, $E[X_i] \geq (3\sqrt{3}/2\sqrt{q})(E[X_i^2] - E[X_i^4]/q)$. We will show that if the r_i's are chosen 5-wise independently in algorithm Aperm,

$$\frac{3\sqrt{3}}{2\sqrt{q_i}} \left(E[X_i^2] - \frac{E[X_i^4]}{q_i} \right) \geq \frac{\sqrt{6}}{20} \sqrt{\deg(i)} + \frac{\sqrt{3}}{10} |d_{\text{out}}(i) - d_{\text{in}}(i)|,$$

where $q_i = \deg(i) + 2(d_{\text{out}}(i) - d_{\text{in}}(i))^2$.

To do this, we need to compute $E[X_i^2]$ and $E[X_i^4]$. The following claim will be helpful.

CLAIM 5.1. *$(X_i)^k$ is a function which is a sum of terms depending on at most $k+1$ ranks r_i each.*

Proof. X_i^k is a polynomial which is a sum of terms depending on k Z_{ij}'s each. It would appear that k Z_{ij}'s are dependent on $2k$ r_i's. However, since each term of k Z_{ij}'s is really dependent on one i and at most k $j \in N(i)$, each term is a function of at most $k+1$ r_i's. □

We can thus use the fact that, although $Y_{ij_1}, Y_{ij_2}, \ldots, Y_{ij_t}$ are not independent, we can still compute their expected product by the following lemma for $t \leq 4$.

LEMMA 5.6. *$E[Y_{ij_1} Y_{ij_2} \cdots Y_{ij_t}] = 1/(t+1)$ when $j_1, j_2, \ldots, j_t$ are all distinct and $t \leq 4$.*

Proof. This is because the product $Y_{ij_1} Y_{ij_2} \cdots Y_{ij_t}$ is 0 except when i has a lower rank than all of $j_1, \ldots, j_t$. We note that if ranks $\langle r_i, r_{j_1}, \ldots, r_{j_t} \rangle$ are chosen from at least a $(t+1)$-wise independent distribution and ties are broken through a random permutation, then all $(t+1)$-tuples of distinct rank values are equally likely. Then by symmetry, since each of $r_i, r_{j_1}, \ldots, r_{j_t}$ is equally likely to be the least element, only $1/(t+1)$ has r_i least. □

Before we compute $E[X_i^2]$ and $E[X_i^4]$, let us give some useful definitions. Let $B = \{\langle i, j \rangle \in A\}$ and $C = \{\langle j, i \rangle \in A\}$. Let $b = d_{\text{out}}(i) = |B|$ and $c = d_{\text{in}}(i) = |C|$. Let $z = b+c$ and $x = |b-c|$. Let $B_k = \{(j_1, \ldots, j_k) | j_1, \ldots, j_k \in B \text{ all distinct}\}$ and let $C_k = \{(j_1, \ldots, j_k) | j_1, \ldots, j_k \in C \text{ all distinct}\}$. Let $\beta = \sum_{j \in B} Y_{ij}$ and $\gamma = \sum_{j \in C} Y_{ij}$. Without loss of generality, we assume that $b \geq c$.

LEMMA 5.7. *$E[X_i^2] = z/6 + x^2/3$.*

Proof. Observe that

$$E[X_i^2] = E[(\beta - \gamma)^2] = E[\beta^2 - 2\beta\gamma + \gamma^2].$$

We will compute this term by term using linearity of expectation. First,

$$E[\beta^2] = \sum_{j \in B} E[Y_{ij}^2] + \sum_{(j,j') \in B_2} E[Y_{ij} Y_{ij'}] = \sum_{j \in B} E[Y_{ij}] + \sum_{(j,j') \in B_2} E[Y_{ij} Y_{ij'}] = b\frac{1}{2} + (b^2 - b)\frac{1}{3}$$

by Lemma 5.6. By symmetry,

$$E[\gamma^2] = c\frac{1}{2} + (c^2 - c)\frac{1}{3}.$$

Also,

$$E[-2\beta\gamma] = -2 \sum_{j \in B} \sum_{j' \in C} E[Y_{ij} Y_{ij'}] = -2bc\frac{1}{3}.$$

Combining all terms,

$$E[X_i^2] = \frac{1}{6}(b+c) + \frac{1}{3}(b-c)^2 = \frac{1}{6}z + \frac{1}{3}x^2. \qquad \square$$

LEMMA 5.8. *$E[X_i^4] = (1/30)(z + 2x^2)(3z + 3x^2 - 1)$.*

Proof. Observe that

$$E[X_i^4] = E[(\beta - \gamma)^4] = E[\beta^4 - 4\beta^3\gamma + 6\beta^2\gamma^2 - 4\beta\gamma^3 + \gamma^4].$$

We will likewise compute this term by term using linearity of expectation. First,

$$\begin{aligned}E[\beta^4] &= \sum_{j\in B} E[Y_{ij}^4] + \binom{4}{1}\sum_{(j,j')\in B_2} E[Y_{ij}^3 Y_{ij'}] + 3\sum_{(j,j')\in B_2} E[Y_{ij}^2 Y_{ij'}^2] \\ &\quad + \binom{4}{2}\sum_{(j,j',j'')\in B_3} E[Y_{ij}^2 Y_{ij'} Y_{ij''}] + \sum_{(j,j',j'',j''')\in B_4} E[Y_{ij}Y_{ij'}Y_{ij''}Y_{ij'''}] \\ &= b\frac{1}{2} + \binom{4}{1}b(b-1)\frac{1}{3} + 3b(b-1)\frac{1}{3} + \binom{4}{2}b(b-1)(b-2)\frac{1}{4} \\ &\quad + b(b-1)(b-2)(b-3)\frac{1}{5}.\end{aligned}$$

By symmetry,

$$\begin{aligned}E[\gamma^4] &= c\frac{1}{2} + \binom{4}{1}c(c-1)\frac{1}{3} + 3c(c-1)\frac{1}{3} + \binom{4}{2}c(c-1)(c-2)\frac{1}{4} \\ &\quad + c(c-1)(c-2)(c-3)\frac{1}{5}.\end{aligned}$$

Next,

$$\begin{aligned}E[-4\beta^3\gamma] &= -4\left[\sum_{j\in B}\sum_{j'''\in C} E[Y_{ij}^3 Y_{ij'''}] + \binom{3}{2}\sum_{(j,j')\in B_2}\sum_{j'''\in C} E[Y_{ij}^2 Y_{ij'}Y_{ij'''}]\right. \\ &\quad \left. + \sum_{(j,j',j'')\in B_3}\sum_{j'''\in C} E[Y_{ij}Y_{ij'}Y_{ij''}Y_{ij'''}]\right] \\ &= -4\left[bc\frac{1}{3} + \binom{3}{2}b(b-1)c\frac{1}{4} + b(b-1)(b-2)c\frac{1}{5}\right].\end{aligned}$$

By symmetry,

$$E[-4\beta\gamma^3] \;=\; -4\left[bc\frac{1}{3} + \binom{3}{2}bc(c-1)\frac{1}{4} + bc(c-1)(c-2)\frac{1}{5}\right].$$

Also,

$$\begin{aligned}E[6\beta^2\gamma^2] &= 6\left[\sum_{j\in B}\sum_{j''\in C} E[Y_{ij}^2 Y_{ij''}^2] + \sum_{j\in B}\sum_{(j'',j''')\in C_2} E[Y_{ij}^2 Y_{ij''}Y_{ij'''}]\right. \\ &\quad \left. + \sum_{(j,j')\in B_2}\sum_{j''\in C} E[Y_{ij}Y_{ij'}Y_{ij''}^2] + \sum_{(j,j')\in B_2}\sum_{(j'',j''')\in C_2} E[Y_{ij}Y_{ij'}Y_{ij''}Y_{ij'''}]\right] \\ &= 6\left[bc\frac{1}{3} + bc(c-1)\frac{1}{4} + b(b-1)c\frac{1}{4} + b(b-1)c(c-1)\frac{1}{5}\right].\end{aligned}$$

Combining all terms,

$$\begin{aligned}E[X_i^4] &= \frac{(b-c)^4}{5} - \frac{(b-c)^2}{15} + \frac{3(b-c)^2(b+c)}{10} + \frac{(b+c)^2}{10} - \frac{b+c}{30} \\ &= \frac{x^4}{5} - \frac{x^2}{15} + \frac{3x^2 z}{10} + \frac{z^2}{10} - \frac{z}{30} \\ &= \frac{(z+2x^2)(3z+3x^2-1)}{30}.\end{aligned}$$

□

By Lemmas 5.7 and 5.8, when $q = z + 2x^2$,

$$\frac{3\sqrt{3}}{2\sqrt{q}}\left(E[X_i^2] - \frac{E[X_i^4]}{q}\right) \geq \frac{\sqrt{6}}{20}\sqrt{z} + \frac{\sqrt{3}}{10}\,x.$$

This fact and Theorem 2.2 imply that

$$E[X_i] \geq \frac{\sqrt{6}}{20}\sqrt{z} + \frac{\sqrt{3}}{10}\,x.$$

This implies

$$\sum_{i=1}^{n} E[X_i] \geq \frac{\sqrt{6}}{20}\sum_{i=1}^{n}\sqrt{\deg(i)} + \frac{\sqrt{3}}{10}\sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|.$$

Therefore, plugging the above equation into equation (5.1), we have for algorithm Aperm that

$$E[|\hat{A}|] \geq \frac{|A|}{2} + \frac{\sqrt{6}}{40}\sum_{i=1}^{n}\sqrt{\deg(i)} + \frac{\sqrt{3}}{20}\sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|. \tag{5.2}$$

Now let us consider the expected behavior of the actual algorithm A5wise since we do not really have access to an algorithm that produces truly random permutations. Recall from the discussion above that $E[|\hat{A}|]$ for A5wise is at least $E[|\hat{A}|]$ for Aperm minus the expected number of arcs with equal rank (i.e., $E[S]$). Therefore, from equation (5.2) and this fact, we have shown for algorithm A5wise that

$$E[|\hat{A}|] \geq \frac{1}{2}|A| - E[S] + \frac{1}{2}\sum_{i=1}^{n} E[X_i].$$

We already lower bounded equation (5.1) in equation (5.2), so all that remains is to upper bound $E[S]$, which is achieved by the following lemma.

Lemma 5.9. $E[S] \leq |A|/\sigma$.

Proof. For any pairwise independent distribution, $E[s_{ij}] = 1/\sigma$. □

We have succeeded in proving the following theorem for A5wise.

Theorem 5.10. *For $\langle r_1, \ldots, r_n\rangle \in \{1, \ldots, \sigma\}^n$ chosen from a 5-wise independent distribution, the expected size of the acyclic subgraph found by the RNC algorithm is*

$$E[|\hat{A}|] \geq \frac{|A|}{2} - \frac{|A|}{\sigma} + \frac{\sqrt{6}}{40}\sum_{i=1}^{n}\sqrt{\deg(i)} + \frac{\sqrt{3}}{20}\sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|.$$

Observe that the larger we make σ, the better the lower bound we get on the expected size of our acyclic subgraph. However, in terms of asymptotics, $\sigma \geq 16\sqrt{\Delta}$, where Δ is the maximum degree of any vertex in the graph, suffices (since the second term is then dominated by the third). This will be useful in the next section since we will see that there is a tradeoff between the bound we achieve on $|\hat{A}|$ and the number of processors, or running time, we get when we derandomize.

Corollary 5.11. *For $\langle r_1, \ldots, r_n\rangle \in \{1, \ldots, 16\sqrt{\Delta}\}^n$ chosen from a 5-wise independent distribution,*

$$E[|\hat{A}|] \geq \frac{|A|}{2} + \Omega\left(\sum_{i=1}^{n}\sqrt{\deg(i)} + \sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|\right).$$

5.4.2. The *NC* algorithm. We can derandomize the algorithm of the previous section in parallel. Since that randomized algorithm obtained a large acyclic subgraph using 5-wise independence, we can derandomize by simply trying all sample points of a 5-wise independent distribution, or, as we will show, we can use the techniques of [5] to do a parallel binary search. While the former approach is simpler, in this case, the latter requires fewer processors.

Before we describe either of these approaches, it will be convenient to define a benefit function [14, 18]

$$B(W) \;=\; \frac{1}{2}|A| - S + \frac{3\sqrt{3}}{4}\sum_{i=1}^{n}\frac{1}{\sqrt{q_i}}\left(X_i^2 - \frac{X_i^4}{q_i}\right),$$

where $W = \langle r_1, \ldots, r_n\rangle \in \{1, \ldots, \sigma\}^n$ and $q_i = \deg(i) + (d_{\text{out}}(i) - d_{\text{in}}(i))^2$. By the analysis leading to the proof of Theorem 5.10, we know that $E[B(W)]$ for W chosen over a 5-wise independent distribution is large.

We can therefore try all sample points W in a 5-wise independent distribution to find a W such that $B(W) \geq E[B(W)]$. This would require one processor for each sample point, of which there are $O(n^5 + \sigma^5)$. If we pick $\sigma = n$, we get a large acyclic subgraph, but we may need $O(n^5)$ processors.

Alternatively, we can do a binary search in parallel to obtain the following theorem. Recall that $\Delta = \max_{1\leq i\leq n}\deg(i)$.

THEOREM 5.12. *There is an NC algorithm for maximum acyclic subgraph which uses $O(n\Delta^4)$ processors, runs in $O(\log^2 n \log\sigma)$ time, and produces an acyclic subgraph $\hat{A}$ of size*

$$|\hat{A}| \;\geq\; \frac{|A|}{2} \;-\; \frac{|A|}{\sigma} \;+\; \frac{\sqrt{6}}{40}\sum_{i=1}^{n}\sqrt{\deg(i)} \;+\; \frac{\sqrt{3}}{20}\sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|.$$

Proof. To find a good W, we apply one of the basic approaches for handling multivalues given in [5] to the problem at hand. We have 5-wise independent multivalued random variables $W = \langle r_1, \ldots, r_n\rangle$, where $\text{bin}(r_k) = r_{k1}r_{k2}\cdots r_{kl}$ ($r_{kt} \in \{0, 1\}$ and $l = \lceil\log\sigma\rceil$); in other words, W can be thought of as an $n \times l$ Boolean matrix that has the r_k's as its rows. We compute the r_k's bit by bit, setting the tth bit of all of the r_k's simultaneously. Let $U^{(t)}$ be the tth column of W once all of the bits in the column are set. Similarly, $U_k^{(t)} = r_{kt}$ once its bit is set. At step t, we will show below how to compute the tth bit of all the r_k's, $U^{(t)}$ such that

$$\begin{aligned} &E[B(W)|r_{kj} = U_k^{(j)} \text{ for } 1\leq k\leq n,\ 1\leq j\leq t] \\ &\quad\geq E[B(W)|r_{kj} = U_k^{(j)} \text{ for } 1\leq k\leq n,\ 1\leq j\leq t-1]. \end{aligned}$$

In other words, find a setting for the tth bits of the r_k's such that the expected benefit from setting the first t bits of the r_k's is at least the expected benefit from setting the first $t-1$ bits of the r_k's. If we let

$$F^{(t)}(U^{(t)}) = E[B(W)|r_{kj} = U_k^{(j)} \text{ for } 1\leq k\leq n,\ 1\leq j\leq t],$$

then the above is equivalent to finding a $U^{(t)}$ with $F^{(t)}(U^{(t)}) \geq E[F^{(t)}(U^{(t)})]$.

A simple inductive argument then shows that for all t,

$$E[B(W)|r_{kj} = U_k^{(j)} \text{ for } 1\leq k\leq n,\ 1\leq j\leq t] \geq E[B(W)].$$

It follows that letting W be such that $r_{kj} = U_k^{(j)}$ for all k and j implies that $B(W) \geq E[B(W)]$.

To find a setting of $U^{(t)}$ such that $F^{(t)}(U^{(t)}) \geq E[F^{(t)}(U^{(t)})]$, we utilize the method given in [5] for binary searching a distribution with n 5-wise independent 0/1 random variables. This works as follows. We construct an $n \times O(\log n)$ matrix M whose rows are 5-wise linearly independent. It was shown that for a randomly chosen vector $\omega \in \{0,1\}^{O(\log n)}$, taking the matrix–vector product of $M\omega$ produces an X which consists of n 5-wise independent 0/1 random variables. By setting ω one bit at a time so that the expected benefit of X, given the settings of ω to that point, is always nondecreasing, it is possible to get a setting for X such that the benefit of X is at least the expected benefit of X.

It remains to show how to compute the expected benefit from setting the first t bits of the r_k's, given a partial setting of ω. First, let us define some terms. From Claim 5.1, we know that our benefit function $B(W)$ is a sum of terms depending on at most five ranks r_k each. Let $\alpha = \{ij_1, \ldots, ij_\tau\}$, where $\tau \in [1,4]$ and $j_1, \ldots, j_\tau$ are all distinct, be the indices of the arcs involved in such a term. Recall that Y_{ij} is the indicator random variable for $r_j > r_i$ and s_{ij} is the indicator random variable for $r_i = r_j$. Then $B(W)$ is a sum of $O(n\Delta^4)$ functions $g_\alpha(W) = Y_{ij_1}Y_{ij_2}\cdots Y_{ij_\tau}$ and $O(n\Delta)$ functions $s_{ij}(W) = I\{r_i = r_j\}$, each depending on at most five r_k's each. Note that g_α is the indicator function for $Y_{ij_1}Y_{ij_2}\cdots Y_{ij_\tau}$, i.e., g_α is 1 if r_i is the minimum among $r_i, r_{j_1}, \ldots, r_{j_\tau}$ and 0 otherwise.

Let

$$f_\alpha^{(t)}(U^{(t)}) = E[g_\alpha(W) | r_{kj} = U_k^{(j)} \text{ for } 1 \leq k \leq n,\ 1 \leq j \leq t],$$

which is the expectation that r_i is minimum among $r_{j1}, \ldots, r_{jt}$, given that the first t bits of the r_k's are specified. Let

$$f_{ij}^{(t)}(U^{(t)}) = E[s_{ij}(W) | r_{kj} = U_k^{(j)} \text{ for } 1 \leq k \leq n,\ 1 \leq j \leq t],$$

which is the expectation that the first t bits of the r_k's are specified. This allows us to write $F^{(t)}(U^{(t)})$ as a sum of $O(n\Delta^4)$ functions $f_\alpha^{(t)}$ and $f_{ij}^{(t)}$, each depending on at most five ranks r_{kt}. Assuming that, given $U^{(1)}, \ldots, U^{(t-1)}$, we can construct functions $f_\alpha^{(t)}$ and $f_{ij}^{(t)}$, given a partial setting of ω, we can compute $F^{(t)}(U^{(t)})$ given the partial setting of ω. We then choose the bit of ω that was set so as to maximize $F^{(t)}(U^{(t)})$. After all bits of ω have been set in this way, we have found a $U^{(t)}$ such that $F^{(t)}(U^{(t)}) \geq E[F^{(t)}(U^{(t)})]$.

We now show how to construct, for any t and for any settings of the first $t-1$ bits $U^{(1)}, \ldots, U^{(t-1)}$, the functions $f_\alpha^{(t)}(U^{(t)})$ given a partial setting of ω. Given term α, the $(t-1)$-bit prefix of each rank $r_i, r_{j_1}, \ldots, r_{j_\tau}$, and partial information on the tth bit of each of these ranks, $f_\alpha^{(t)}$ is the probability that r_i is the minimum rank among $r_i, r_{j_1}, \ldots, r_{j_\tau}$. To calculate $f_\alpha^{(t)}$, we sort the ranks of term α into groups of those that have a strictly smaller t-bit prefix than r_i (group Γ_1), those that have the same t-bit prefix as r_i (group Γ_2), and those that have a strictly larger t-bit prefix than r_i (group Γ_3). If $|\Gamma_1| > 0$, then $f_\alpha^{(t)} = 0$. Otherwise, we focus on those ranks with the same t-bit prefix because only these are in contention for the minimum rank value.

Again for the purposes of analysis, we utilize algorithm Aperm, which we specified to perform a random permutation of those ranks that are equal upon setting the last (i.e., the lth) bit of the ranks. We will analyze what happens to algorithm Aperm

up to the permutation (i.e., the bits of the lth column are set but the identical ranks have not been permuted yet). Intuitively, we do this because we cannot derandomize $O(n!)$ possibilities.

By way of example, suppose we have five ranks involved in the g_α term. There are 2^5 different ways to set the tth bits for these ranks. For each setting of the five bits, we have a collection of ranks whose leading order $t-1$ bits are the same. We have partial information on their tth bit (from the partial setting of ω) but no information on their remaining bits (i.e., they are completely random). For each possible setting of the five bits, we want to know the probability of that r_i is the minimum rank:

$$\begin{aligned}&\Pr[r_i \text{ is min} \mid \text{partial info on } \omega]\\ &= \sum_{32 \text{ settings}} (\Pr[\text{get that setting} \mid \text{partial info on } \omega] \cdot \Pr[r_i \text{ is min} \mid \text{get that setting}]).\end{aligned}$$

Calculating $\Pr[r_i \text{ is min} \mid \text{get that setting}]$ merely requires a table lookup: it is 0 or $1/((\#$ of identical ranks including setting of tth bit)$+1)$ since all of the ranks are assumed to be distinct for algorithm Aperm.

To calculate $\Pr[\text{get that setting} \mid \text{partial info on } \omega]$, assume without loss of generality that we have a five-row matrix M' such that all the rows are independent. We are setting the bits of a $l' = O(\log n)$ bit vector ω one at a time. Some of ω's values have been set, let us say t', and the rest are random. We want to know the probability that when we plug in the remaining $l'-t'$ unfixed values into vector ω, the five bits of the ranks will get the specified setting. This probability can be calculated by solving a linear system of equations with $l'-t'$ unknowns. Suppose u is the dimension of the solution space. Then the probability is $2^u/2^{l'-t'} = 2^{u-l'+t'}$. The solution to the system of equations can be computed quickly since M' is of size $5 \times O(\log n)$ and ω is of size $\log n$, so it can be done sequentially in NC.

Next, we show how to construct, for any t and for any settings of the first $t-1$ bits $U^{(1)}, \ldots, U^{(t-1)}$, the functions $f_{ij}^{(t)}(U^{(t)})$ given a partial setting of ω. Note that the indicator s_{ij} is only for the part of the algorithm up to the random permutation. As with $f_\alpha^{(t)}$, to compute $f_{ij}^{(t)}$, we first show how to compute

$$E[s_{ij}(W) | r_{kj} = U_k^{(j)} \text{ for } 1 \le k \le n,\ 1 \le j \le t];$$

it then suffices to plug in the given $U^{(1)}, \ldots, U^{(t-1)}$ and every possible setting of the variables $\{U_k^{(t)} | k \in \{i,j\}\}$ to construct $f_{ij}^{(t)}$.

Given arc $\langle i,j \rangle$ and the first t bits of ranks r_i and r_j, $f_{ij}^{(t)}$ is the probability that $r_i = r_j$. Clearly, if r_i and r_j have different t-bit prefixes, then $f_{ij}^{(t)} = 0$. Otherwise,

$$\begin{aligned}f_{ij}^{(t)} &= \Pr[r_i = r_j | r_i \text{ and } r_j \text{ have same } t\text{-bit prefix}]\\ &= \frac{1}{2^{l-t}}.\end{aligned}$$

Now we want to argue that the above procedure finds a sample point W (i.e., a collection of ranks) for which $|\hat{A}|$ is large. For any collection of ranks, the following is true:

$$|\hat{A}| = \sum_{i=1}^{n} \left(\frac{d_i}{2} + \frac{X_i}{2} \right) = \frac{1}{2}|A| - \frac{1}{2}S + \frac{1}{2}\sum_{i=1}^{n} X_i. \tag{5.3}$$

Note that the sum over all d_i is equivalent to $|A| - S$ since S arcs are not processed, and each processed arc is counted only once. In particular, let us suppose that equation (5.3) applies to the collection of 5-wise independent ranks obtained before the final random permutation is performed by algorithm APERM. S is then the number of arcs with equal rank at this point. By the discussion in the preceding section, the $\sum_i X_i$ for these ranks is at least $\sum_i X_i - S$ for the collection of ranks produced by any final permutation. The number of arcs we get for the acyclic subgraph $|\hat{A}|$ when the sample point W is derived by the above process is at least

$$\begin{aligned}
&\frac{1}{2}|A| - \frac{1}{2}S + \frac{1}{2}\sum_{i=1}^{n} X_i \\
&\qquad \text{(where } X_i \text{ is derived from 5-wise independent distribution for } r_k\text{'s)} \\
&\geq \frac{1}{2}|A| - \frac{1}{2}S + \frac{1}{2}\sum_{i=1}^{n}(E[X_i] - S) \\
&\qquad \text{(where } E[X_i] \text{ is derived from tie-breaking } r_k\text{'s by a random permutation)} \\
&= \frac{1}{2}|A| - S + \frac{1}{2}\sum_{i=1}^{n} E[X_i] \\
&\geq \frac{1}{2}|A| - S + \frac{3\sqrt{3}}{4}\sum_{i=1}^{n}\frac{1}{\sqrt{q_i}}\left(E[X_i^2] - \frac{E[X_i^4]}{q_i}\right),
\end{aligned}$$

for $q_i = \deg(i) + 2(d_{\text{out}}(i) - d_{\text{in}}(i))^2$ (by Theorem 2.2). The latter part of the equation is the expected benefit $E[B(W)]$. (S was fixed after the 5-wise independent ranks were determined.)

Thus we have selected a sample point that achieves at least the expected benefit up to the random permutation that is left. The expected benefit was computed to be large in the previous section (see Theorem 5.10). By the fact that for each column of M, we deal with at most $O(n\Delta^4)$ terms and by the running-time analysis in [5], we have achieved the desired Theorem 5.12. □

COROLLARY 5.13. *There is an NC algorithm for maximum acyclic subgraph which uses $O(n\Delta^4)$ processors, runs in $O(\log^3 n)$ time, and produces an acyclic subgraph of size*

$$|\hat{A}| \;\geq\; \frac{|A|}{2} \;+\; \Omega\left(\sum_{i=1}^{n}\sqrt{\deg(i)} \;+\; \sum_{i=1}^{n}|d_{\text{out}}(i) - d_{\text{in}}(i)|\right).$$

In the case where $\deg(i) = |V| - 1$, Theorem 5.12 and Corollary 5.13 imply *NC* algorithms and analogous lower bounds for tournament ranking.

Acknowledgments. I am happy to thank Lenore Cowen, Dan Kleitman, Tom Leighton, and John Rompel for helpful discussions. Thanks to Prof. Dudley for helpful references. Also, thanks to Joel Spencer for suggesting the title and pointing out that the lower bound on discrepancy could be immediately applied to get an *NC* algorithm for the switching game.

REFERENCES

[1] N. ALON, L. BABAI, AND A. ITAI, *A fast and simple randomized parallel algorithm for the maximal independent set problem*, J. Algorithms, 7 (1986), pp. 567–583.

[2] B. Berger, *Data structures for removing randomness*, Technical Report MIT/LCS/TR-436, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1988.
[3] B. Berger, *Using randomness to design efficient deterministic algorithms*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1990.
[4] B. Berger, *The fourth moment method*, in Proc. 2nd Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1991, pp. 373–383.
[5] B. Berger and J. Rompel, *Simulating* $(\log^c n)$*-wise independence in NC*, J. Assoc. Comput. Mach., 38 (1991), pp. 1026–1046.
[6] B. Berger and P. Shor, *Approximation algorithms for the maximum acyclic subgraph problem*, in Proc. 1st Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1990, pp. 236–243.
[7] T. Brown and J. Spencer, *Minimization of* $\pm$ *matrices under line shifts*, Colloq. Math. (Poland), 23 (1971), pp. 165–171.
[8] W. Fernandez de la Vega, *On the maximum cardinality of a consistent set of arcs in a random tournament*, J. Combin. Theory B, 35 (1983), pp. 328–332.
[9] L. Devroye and L. Gjörfi, *Nonparametric Density Estimation: The* $L1$ *View*, John Wiley, New York, 1985.
[10] P. Erdös and J. Spencer, *Imbalances in* k*-colorations*, Networks, 1 (1972), pp. 379–385.
[11] R. Greenlaw, *The parallel complexity of approximation algorithms for the acyclic subgraph problem*, Technical Report 90-61, Department of Computer Science, University of New Hampshire, Durham, NH, 1990.
[12] G. Hardy, J. E. Littlewood, and G. Pòlya, *Inequalities*, Cambridge Mathematical Library, Cambridge, MA, 1988.
[13] A. Joffe, *On a set of almost deterministic* k*-independent random variables*, Ann. Probability, 2 (1974), pp. 161–162.
[14] R. M. Karp and A. Wigderson, *A fast parallel algorithm for the maximal independent set problem*, J. Assoc. Comput. Mach., 32 (1985), pp. 762–773.
[15] R. H. Lathrop, R. J. Hall, and R. S. Kirk, *Functional abstraction from structure in VLSI simulation models*, in Proc. 24th IEEE–ACM Design Automation Conference, IEEE, Piscataway, NJ, 1987, pp. 822–828.
[16] J. Leung, private communication, 1989.
[17] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1053.
[18] M. Luby, *Removing randomness in parallel computation without a processor penalty*, J. Comput. System Sci., 47 (1993), pp. 250–286.
[19] R. Motwani, J. Naor, and M. Naor, *The probabilistic method yields deterministic parallel algorithms*, J. Comput. System Sci., 49 (1994), pp. 478–516.
[20] S. Poljak, V. Rödl, and J. Spencer, *Tournament ranking with expected profit in polynomial time*, SIAM J. Discrete Math., 1 (1988), pp. 372–376.
[21] P. Raghavan, *Probabilistic construction of deterministic algorithms: Approximating packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130–143.
[22] V. Ramachandran, *Finding a minimum feedback arc set in reducible flow graphs*, J. Algorithms, 9 (1988), pp. 299–313.
[23] I. R. Savage, *Probability inequalities of the Tchebycheff type*, J. Res. Nat. Bureau Standards, 65B (1972), pp. 211–222.
[24] M. H. Shirley, *Generating circuit tests by exploiting designed behavior*, Technical Report MIT/AI/TR-1099, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1988.
[25] J. Spencer, *Optimal ranking of tournaments*, Networks, 1 (1972), pp. 135–138.
[26] J. Spencer, *Ten Lectures on the Probabilistic Method*, SIAM, Philadelphia, PA, 1987.
[27] P. Tetali, *Derandomization of discrepancy results*, manuscript, 1991.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1208–1244, August 1997

014

TESTING SHARED MEMORIES*

PHILLIP B. GIBBONS† AND EPHRAIM KORACH‡

Abstract. Sequential consistency is the most widely used correctness condition for multiprocessor memory systems. This paper studies the problem of testing shared-memory multiprocessors to determine if they are indeed providing a sequentially consistent memory. It presents the first formal study of this problem, which has applications to testing new memory system designs and realizations, providing run-time fault tolerance, and detecting bugs in parallel programs.

A series of results are presented for testing an execution of a shared memory under various scenarios, comparing sequential consistency with linearizability, another well-known correctness condition. Linearizability imposes additional restrictions on the shared memory, beyond that of sequential consistency; these restrictions are shown to be useful in testing such memories.

Key words. sequential consistency, linearizability, multiprocessors, shared memory, testing, NP-completeness

AMS subject classifications. 68M15, 68M07, 68Q60, 68Q22

PII. S0097539794279614

1. Introduction.

Shared-memory multiprocessors typically promise application and system programmers some high-level view of the memory system. High-level correctness conditions such as sequential consistency [26] provide a conceptually simple framework for programming parallel machines. In a *sequentially consistent* memory, each execution is indistinguishable (by the processors) from an execution of a (very fast) serial memory in which only one read or write occurs at a time, in an order consistent with the respective sequences of reads and writes at the individual processors [26, 3]. Sequential consistency is the most widely used correctness condition for multiprocessor memory systems.

A memory system promising sequential consistency may fail to provide it for a number of reasons. First, high-performance shared-memory multiprocessors (cf. [5, 7, 12, 27]) employ a variety of techniques to improve their memory system performance (e.g., buffering, pipelining, caching, multiple paths to memory, parallel access to memory banks); these serve to distance the implementation from the sequential consistency abstraction. Subtle design errors can occur in the memory system architecture or in the supporting compilers due to the complexity of the design and the difficulty in reasoning about asynchronous, concurrent systems. Second, various hardware components may fail; such failures are more common in parallel machines due to the multitude of components devoted to providing the large shared-memory system. Third, certain implementations used in practice provide only an approximation to sequential consistency, e.g., *processor consistency*, as a tradeoff for improved performance [16]. Fourth, shared-memory multiprocessors may support *release consistency* [16, 21], which provides a sequentially consistent memory (only) for programs that are free of data races. (A *data race* occurs when two or more processors access the same location, with at least one writing, without intervening synchronization.) The memory system may fail to provide sequential consistency if the program contains data races.

* Received by the editors December 20, 1994; accepted for publication (in revised form) September 15, 1995.
http://www.siam.org/journals/sicomp/26-4/27961.html
† Bell Laboratories, Lucent Technologies, Room 2D-148, 600 Mountain Avenue, Murray Hill, NJ 07974 (gibbons@research.bell-labs.com).
‡ Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel (korach@bgumail.bgu.ac.il). Part of this author's work was done while visiting Bell Laboratories, Murray Hill, NJ.

In this paper, we study the problem of testing shared-memory multiprocessors to determine if they are indeed providing a sequentially consistent memory. We focus on the basic problem of testing whether the memory system provided sequential consistency for a given execution of a parallel program. For the given execution, the test provides certification of the consistency or inconsistency of the memory system during that execution, providing useful feedback whenever the memory system is suspect or the program may contain data races, as discussed above. The test can also be used as a building block in testing new memory system designs and realizations by verifying each execution in a suite of test executions.

This paper provides a formal and systematic study of the complexity of testing the correctness of an execution of a shared memory based on the reads and writes observed by the individual processors. We define the problem *verifying sequential consistency of shared-memory executions* (VSC) and prove that it is an NP-complete problem. This motivates the study of various restrictions on the problem to further characterize its complexity. Some of the variants that we consider are also motivated by their potential utility in testing executions of existing parallel machines.

We compare results obtained for testing for sequential consistency with results obtained for testing for linearizability, another well-known correctness condition. In a *linearizable* shared memory, each execution is indistinguishable from an execution of a serial memory, in which each read or write occurs at a distinct point in time between when it is issued by the processor and when the system acknowledges its completion [22]. We define the problem *verifying linearizability of shared-memory executions* (VL) and show that the additional restrictions imposed by linearizability beyond that of sequential consistency are quite useful in testing such memories. In particular, we present $O(n \log n)$-time algorithms for several variants of the VL problem whose corresponding VSC variants are NP-complete.

In an independent work, Wing and Gong [34] defined and studied the problem of testing and verifying linearizability for arbitrary shared data structures (e.g., a FIFO queue with push and pop operations). They developed a simulation environment for testing implementations (written using a C-Threads package) of shared data structures, using as a building block a procedure for verifying individual executions. This procedure uses a greedy algorithm that runs in exponential time; for this reason, Wing and Gong suggested testing implementations by verifying many, short executions (at most several hundred operations each). They also point out that the general problem they consider is NP-complete. Finally, they presented specifications, implementations, and proofs of correctness (i.e., linearizability) for a number of shared data structures. In contrast to their work, we focus specifically on shared memories, consider both sequential consistency and linearizability, and present fast algorithms for several important variants of these testing problems.

Other related previous work includes work on devising a suite of simple programs to help test whether the memory system is providing sequential consistency or a weaker correctness condition [11], work on detecting violations of sequential consistency within the memory system itself [14, 15], work on testing the serializability of database transactions [31], work on detecting data races (e.g., [2, 23, 28, 29]), work on proving that weak memory systems provide sequential consistency for programs that are free of data races (e.g., [1, 20, 21]), work on testing uniprocessor memories [9], work on algorithms for testing data structures on uniprocessors (e.g., [10]), work on verifying specific properties of cache-coherence protocols (e.g., [32] and the references therein), work on computing with faulty shared memories [4], work on de-

termining minimal ordering constraints needed to preserve sequential consistency [33], and work on comparing implementations of sequential consistency versus linearizability (e.g., [8]). However, none of this work addresses the general testing questions considered in this paper.

1.1. The testing problems. During an execution of a parallel program on a given multiprocessor, processors request to read or write particular shared-memory locations as dictated by the program, and the memory system responds to each request with a return value or acknowledgment. Associated with each processor is a total order on its shared-memory operations, denoted its *program order*. Associated with each read operation, $read(a, d, t_1, t_2)$, issued by a processor are the address a of the shared-memory location read, the value d returned for the read, the time t_1 the read was issued, and the time t_2 of the response. Likewise, associated with each write operation, $write(a, d, t_1, t_2)$, issued by a processor are the address a of the shared-memory location written, the value d written, the time t_1 the write was issued, and the time t_2 of the response. The times t_1 and t_2 define an interval of time for an operation: t_1 is the *start-of-interval* time for the operation and t_2 is the *end-of-interval* time for the operation.

We consider testing procedures in which for each processor we are given its sequence of shared-memory operations. Our goal is to determine whether or not the sequences can be interleaved as required by the correctness condition. For example, both sequential consistency and linearizability require that in the interleaved sequence, each read operation returned the value that was written by the last preceding write to the same location (the usual read/write semantics).

Sequential consistency. Sequential consistency is based on respecting the program orders and the read/write semantics, while ignoring the start-of-interval and end-of-interval times. We define the *verifying sequential consistency of shared-memory executions* (VSC) problem as follows.

VERIFYING SEQUENTIAL CONSISTENCY OF SHARED-MEMORY EXECUTIONS

INSTANCE: Variable set A, value set D, finite collection of nonempty sequences $S_1, \ldots, S_p$, each consisting of a finite set of memory operations of the form "$read(a, d)$" or "$write(a, d)$," where $a \in A, d \in D$.

QUESTION: Is there a sequence S, an interleaving of $S_1, \ldots, S_p$, such that for each $read(a, d)$ in S, there is a preceding $write(a, d)$ in S with no other $write(a, d')$ between the two?

We denote such a sequence S as a *legal schedule* for the instance; such a schedule certifies that the memory system provided sequential consistency for the execution corresponding to the instance. If there is no legal schedule, then the memory system failed to provide sequential consistency.

Figure 1 depicts a positive instance of the VSC problem. Figure 2 depicts a negative instance.

Linearizability. In contrast, linearizability adds the further constraint that the schedule S must respect the time intervals for the operations. We define the *verifying linearizability of shared-memory executions* (VL) problem as follows.

VERIFYING LINEARIZABILITY OF SHARED-MEMORY EXECUTIONS

INSTANCE: Variable set A, value set D, finite collection of nonempty sequences $S_1, \ldots, S_p$, each consisting of a finite set of memory operations of the form "$read(a, d, t_1, t_2)$" or "$write(a, d, t_1, t_2)$," where $a \in A$, $d \in D$, and t_1 and t_2 are positive rationals, $t_1 < t_2$, defining

$$S_1: \quad write(a,0), write(b,1), read(a,1).$$
$$S_2: \quad read(b,1), write(a,1), write(c,0).$$

FIG. 1. *A positive instance of the VSC problem. In fact, there are two different legal schedules:* $write(a,0)$, $write(b,1)$, $read(b,1)$, $write(a,1)$, $read(a,1)$, $write(c,0)$ *and* $write(a,0)$, $write(b,1)$, $read(b,1)$, $write(a,1)$, $write(c,0)$, $read(a,1)$.

$$S_1: \quad write(a,0), write(a,1), write(b,1).$$
$$S_2: \quad read(b,1), read(a,0).$$

FIG. 2. *A negative instance of the VSC problem. It is not possible to merge* S_1 *and* S_2 *into a legal schedule. For example, in the schedule* $write(a,0)$, $write(a,1)$, $write(b,1)$, $read(b,1)$, $read(a,0)$, *the operation* $write(a,1)$ *is between* $write(a,0)$ *and* $read(a,0)$. *Although* $write(a,1)$ *is an unread write, its existence makes this a negative instance.*

an interval of time such that all intervals in an individual sequence are pairwise disjoint, and t_1 and t_2 are unique rationals in the overall instance.

QUESTION: Is there an assignment of a distinct time to each operation such that

1. each time is within the interval associated with the operation;
2. for each $read(a,d,\tau_1,\tau_2)$, there is a $write(a,d,t_1,t_2)$ assigned an earlier time, with no other $write(a,d',t_1',t_2')$ assigned a time between the two?

Such a time assignment defines a *legal schedule* S that totally orders the operations in the instance. The memory system provided linearizability for the execution corresponding to the instance if and only if there is a legal schedule.

As in the VSC problem, a legal schedule is an interleaving of the individual processor sequences. Thus any linearizable execution is also sequentially consistent. However, as shown below, not all sequentially consistent executions are linearizable.

Figure 3 depicts a positive instance of the VL problem. Thus if the start-of-interval and end-of-interval times are removed from the instance, it is necessarily a positive instance of the VSC problem. Figure 4 depicts a negative instance of the VL problem; this particular instance corresponds to a positive instance of the VSC problem.

For both the VSC and VL problems, the formalization assumes that each variable (i.e., each shared-memory location) must be written before it is read; generalizations to handle reads of the initial state of memory are straightforward. A schedule has a *reads-from violation* if it has a read operation such that either there is no preceding write operation with the same address and value or there is an intervening write operation with the same address but a different value; such a schedule is not legal.

1.2. Results in this paper. Table 1 highlights and compares our main results for the VSC and VL problems. Both problems are NP-complete, as indicated in the table. Three restricted versions of the two problems are studied: bounding the number of operations in each processor sequence, bounding the number of locations in the instance, and bounding the number of processors. A "w.l.o.g." in Table 1

$$S_1: \quad write(a,1,1,3), write(b,1,4,6), write(c,1,7,8).$$
$$S_2: \quad read(b,1,2,5), read(a,1,9,10).$$

FIG. 3. *A positive instance of the VL problem. There are many possible legal time assignments, each of which schedules the write of a, then the write of b, then the read of b, then the write of c, and finally the read of a. If the start-of-interval and end-of-interval times are removed from the instance, we have a positive instance of the VSC problem.*

$$S_1: \quad write(a,0,1,2), read(a,0,5,6).$$
$$S_2: \quad write(a,1,3,4).$$

FIG. 4. *A negative instance of the VL problem. Since the write of value* 1 *must be assigned a time between the write of value* 0 *and the read of value* 0*, there is no legal VL schedule. However, the schedule* $write(a,0)$*,* $read(a,0)$*,* $write(a,1)$ *is a legal VSC schedule.*

indicates that the complexity of the problem is not affected by the given restriction. In addition, we define and study four important variants on the VSC and VL problems; these variants differ in the additional information provided as input, as detailed in section 4. Perhaps somewhat surprisingly, the VSC problem is NP-complete for all but one of the variants considered, in contrast with the results obtained for the VL problem. We also show how our algorithmic results can be extended to handle atomic read–modify–write operations, with no asymptotic penalty.

Our algorithms have small constants and hence are suitable for testing real shared memories (see [17, 19] for implementation details).

Since this is the first paper to study these problems systematically, a number of our NP-completeness results are obtained using somewhat standard techniques. For more interesting constructions, we refer the reader particularly to Theorems 4.3, 4.11, 3.5, and 2.7 of this paper.

The combinatorial questions that arise in studying these testing problems are interesting due to the asymmetry between reads and writes, and the fact that the constraints on legal schedules imposed by reads and writes to the same location cannot be represented as a partial order.

Outline of the paper. The remainder of this paper is organized as follows. Section 2 presents our results for the VSC problem, showing the problem is NP-complete, even with only two operations per processor, only two locations, or only three processors. These results are contrasted with results for the serializability problem for database transactions. Section 3 presents our results for the VL problem, showing that the problem is NP-complete even with only one operation per processor and only one location and, on the other hand, presenting an $O(n \log n)$-time algorithm when the number of processors is fixed. Section 4 presents our results for four variants of the VSC and VL problems that provide additional information as input, specifically, a read-mapping, write-order, read&write only, and conflict-order. It also presents our extensions to handle atomic read-modify-write operations.

Preliminary versions of this work appeared in [18, 19].

2. Verifying sequential consistency. We begin this section with a proof that the VSC problem is NP-complete, even with only two operations per processor (section 2.1). Then the VSC problem is shown to be NP-complete with only two locations

TABLE 1
A summary of the main results of this paper.

variant	*VSC result*	*VL result*
general problem	NP-complete	NP-complete
2 operations per proc	NP-complete	w.l.o.g.
2 locations	NP-complete	w.l.o.g.
3 processors	NP-complete	$O(n \log n)$
read-mapping	NP-complete	$O(n \log n)$
write-order	NP-complete	$O(n \log n)$
read&write only	NP-complete	NP-complete
conflict-order	$O(n \log n)$	$O(n \log n)$

(section 2.2) or only three processors (section 2.3). We conclude this section with some comparisons between the VSC problem (and the results obtained) and the serializability problem for database transactions (section 2.4).

2.1. The VSC problem is NP-complete. The VSC problem is in NP since given a schedule of the reads/writes in the processor sequences, we can test that the schedule is consistent with the processor sequences and does not have reads-from violations, in linear time in one pass through the schedule, simulating the operations.

THEOREM 2.1. *The VSC problem, restricted to instances in which each sequence contains at most two memory operations and each variable occurs in at most two write operations, is* NP*-complete.*

Proof. We use a reduction from the 3-Satisfiability (3SAT) problem [13]. Consider a 3SAT instance $\mathcal{F}$ with n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. We use the notation $(v_i, S(v_i))$ to represent either the variable v_i (when $S(v_i) = \mathsf{T}$) or its complement $\overline{v}_i$ (when $S(v_i) = \mathsf{F}$) in a clause.

To reduce 3SAT to VSC, techniques are needed for simulating an OR and an AND, as well as an assignment of variables that remains in effect until the formula is evaluated. We observe that in all legal schedules, the following must hold:

1. The second operation in a processor sequence must not precede the first.
2. A read operation must not precede the first write operation with the same address and value.
3. A second write operation to an address, writing a different value than the first such write, must not precede any read operation of the first write (in order to avoid a reads-from violation).

Thus assignment to variable v_i can be simulated using the following four sequences, V_i^1, V_i^2, V_i^3, and V_i^4 (listed in columns):

$$\begin{array}{cccc} \underline{V_i^1} & \underline{V_i^2} & \underline{V_i^3} & \underline{V_i^4} \\ W(v_i, \mathsf{T}) & R(x,1) & W(v_i, \mathsf{F}) & R(x,1) \\ & R(v_i, \mathsf{T}) & & R(v_i, \mathsf{F}) \end{array},$$

where a write $W(x,1)$ (shown below) occurs only after the satisfiability of $\mathcal{F}$ has been simulated. Then both writes to v_i cannot occur before $W(x,1)$; this ensures that the initial assignment to each v_i must remain in effect until the satisfiability of $\mathcal{F}$ has been simulated.

An OR is simulated by having two writes to the same location of the same value: a read can be scheduled after either write. For each clause $C_j = (v_p, S(v_p)) \vee (v_q, S(v_q)) \vee$

V_1^1	V_1^2	V_1^3	V_1^4				
$W(v_1,\mathsf{T})$	$R(x,1)$	$W(v_1,\mathsf{F})$	$R(x,1)$				
	$R(v_1,\mathsf{T})$		$R(v_1,\mathsf{F})$	C_1^1	C_1^2	C_1^3	C_1^4
				$R(v_1,\mathsf{T})$	$R(v_3,\mathsf{F})$	$R(v_4,\mathsf{T})$	$R(d_1,\mathsf{T})$
V_2^1	V_2^2	V_2^3	V_2^4	$W(d_1,\mathsf{T})$	$W(d_1,\mathsf{T})$	$W(c_1,\mathsf{T})$	$W(c_1,\mathsf{T})$
$W(v_2,\mathsf{T})$	$R(x,1)$	$W(v_2,\mathsf{F})$	$R(x,1)$				
	$R(v_2,\mathsf{T})$		$R(v_2,\mathsf{F})$	C_2^1	C_2^2	C_2^3	C_2^4
				$R(v_1,\mathsf{F})$	$R(v_2,\mathsf{F})$	$R(v_4,\mathsf{T})$	$R(d_2,\mathsf{T})$
V_3^1	V_3^2	V_3^3	V_3^4	$W(d_2,\mathsf{T})$	$W(d_2,\mathsf{T})$	$W(c_2,\mathsf{T})$	$W(c_2,\mathsf{T})$
$W(v_3,\mathsf{T})$	$R(x,1)$	$W(v_3,\mathsf{F})$	$R(x,1)$				
	$R(v_3,\mathsf{T})$		$R(v_3,\mathsf{F})$	A^0	A^1	A^2	
				$W(x,0)$	$R(c_1,\mathsf{T})$	$R(c_2,\mathsf{T})$	
V_4^1	V_4^2	V_4^3	V_4^4	$W(x,1)$	$R(x,0)$	$R(x,0)$	
$W(v_4,\mathsf{T})$	$R(x,1)$	$W(v_4,\mathsf{F})$	$R(x,1)$				
	$R(v_4,\mathsf{T})$		$R(v_4,\mathsf{F})$				

FIG. 5. *An example of the construction for transforming a* 3SAT *instance to a VSC instance in which each processor sequence has at most two operations and each address is written at most twice. Shown here are the sequences obtained for the* 3SAT *instance* $(v_1 \vee \overline{v}_3 \vee v_4) \wedge (\overline{v}_1 \vee \overline{v}_2 \vee v_4)$.

$(v_r, S(v_r))$, we have four sequences C_j^1, C_j^2, C_j^3, and C_j^4:

C_j^1	C_j^2	C_j^3	C_j^4
$R(v_p, S(v_p))$	$R(v_q, S(v_q))$	$R(v_r, S(v_r))$	$R(d_j,\mathsf{T})$
$W(d_j,\mathsf{T})$	$W(d_j,\mathsf{T})$	$W(c_j,\mathsf{T})$	$W(c_j,\mathsf{T})$

.

Four sequences, as opposed to three, are used in this construction since we are permitted at most two writes to a location. By observations 1 and 2 above, this ensures that the location c_j is not set to T unless clause C_j is satisfied by the guessed truth assignment.

Finally, the AND of the clauses is simulated by the following $m+1$ sequences, $A^0, A^1, \ldots, A^m$:

A^0	A^1	A^2	$\cdots$	A^m
$W(x,0)$	$R(c_1,\mathsf{T})$	$R(c_2,\mathsf{T})$		$R(c_m,\mathsf{T})$
$W(x,1)$	$R(x,0)$	$R(x,0)$		$R(x,0)$

.

The leftmost sequence ensures that $W(x,1)$ is the "second" write to its address. Thus by observation 3 above, x is not set to 1 unless all clauses have been satisfied by the guessed assignment.

This construction uses $4n+5m+1$ sequences and $n+2m+1$ distinct addresses. An example is given in Figure 5.

LEMMA 2.2. *Let $\mathcal{F}$ be an instance of a* 3SAT *problem, and let $\mathcal{V}$ be the instance of the VSC problem constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. We will construct a schedule in which, for each i, the first write to v_i scheduled corresponds to the satisfying truth assignment. Let $T(v_1), \ldots, T(v_n)$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$, be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. first, $W(v_1, T(v_1)), \ldots, W(v_n, T(v_n))$;

2. then for $j = 1, 2, \ldots, m$, all sequences C_j^t whose read is $R(v_k, T(v_k))$ for some k, followed by C_j^4 if either C_j^1 or C_j^2 has been scheduled;
3. then $W(x, 0)$ from A^0, followed by the sequences A^1 to A^m, followed by $W(x, 1)$ from A^0;
4. then for $i = 1, 2, \ldots, n$, if $T(v_i) = \mathsf{T}$, the sequences V_i^2, V_i^3, and V_i^4; otherwise, if $T(v_i) = \mathsf{F}$, the sequences V_i^4, V_i^1, and V_i^2;
5. finally, for $j = 1, 2, \ldots, m$, any remaining sequences C_j^1, C_j^2, or C_j^3, followed by C_j^4 if it has yet to be scheduled.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive VSC instance. If S is a legal schedule for $\mathcal{V}$, then the first value written to each v_i will be our satisfying assignment. We show that any unsatisfied clause corresponds to a cycle in S; a contradiction. For $i = 1, 2, \ldots, n$, let $T(v_i) = \mathsf{T}$ if $W(v_i, \mathsf{T})$ is before $W(v_i, \mathsf{F})$ in S; otherwise, let $T(v_i) = \mathsf{F}$. Suppose $T(v_1), \ldots, T(v_n)$ is *not* a satisfying assignment for $\mathcal{F}$, and let

$$C_k = (v_p, S(v_p)) \vee (v_q, S(v_q)) \vee (v_r, S(v_r))$$

be an unsatisfied clause. Let $\neg S(v_p)$, $\neg S(v_q)$, and $\neg S(v_r)$ denote the complement of $S(v_p)$, $S(v_q)$, and $S(v_r)$, respectively. Since C_k is not satisfied, $W(v_p, \neg S(v_p))$ is before $W(v_p, S(v_p))$, $W(v_q, \neg S(v_q))$ is before $W(v_q, S(v_q))$, and $W(v_r, \neg S(v_r))$ is before $W(v_r, S(v_r))$ in S. Since there are no reads-from violations in S and only two writes to v_p, all of the $R(v_p, \neg S(v_p))$ operations are before any of the $R(v_p, S(v_p))$ operations in S; similarly for v_q and v_r. Since S is a legal schedule, some $W(c_k, \mathsf{T})$ precedes the $R(c_k, \mathsf{T})$ in A^k, which precedes $W(x, 1)$, which precedes all $R(x, 1)$, including the one that precedes $R(v_p, \neg S(v_p))$, which as argued above precedes the $R(v_p, S(v_p))$ from C_k^1, which precedes the $W(d_k, \mathsf{T})$ on C_k^1. Likewise for v_q, some $W(c_k, \mathsf{T})$ precedes the $W(d_k, \mathsf{T})$ on C_k^2. One of these $W(d_k, \mathsf{T})$'s must precede the $R(d_k, \mathsf{T})$ on C_k^4 which precedes the $W(c_k, \mathsf{T})$ on C_k^4. A similar argument for v_r shows that some $W(c_k, \mathsf{T})$ also precedes $W(c_k, \mathsf{T})$ on C_k^3. This is a contradiction since only C_k^3 and C_k^4 contain $W(c_k, \mathsf{T})$. □

Since the above transformation can be done in polynomial time, Theorem 2.1 is proved. □

Note that instances with only one memory operation per processor can be solved by simply checking that there exists a write operation with the same address and value as each read operation.

We also observe that instances with long processor sequences can always be transformed to equivalent instances with at most three memory operations per processor.

OBSERVATION 2.3. *There is a linear-time reduction from the VSC problem with p sequences, n operations, and k variables to the VSC problem with n sequences, $O(n)$ operations, and $k + 1$ variables such that each sequence contains at most three operations.*

Proof. Let α be an address not in the original instance. For $i = 1, \ldots, p$, we replace the ith sequence in the instance, $s_1^{(i)}, s_2^{(i)}, \ldots, s_{m_i}^{(i)}$, with the following m_i sequences:

$$\begin{array}{ccccc} s_1^{(i)} & R(\alpha, x_1^{(i)}) & \cdots & R(\alpha, x_{m_i-2}^{(i)}) & R(\alpha, x_{m_i-1}^{(i)}) \\ W(\alpha, x_1^{(i)}) & s_2^{(i)} & & s_{m_i-1}^{(i)} & s_{m_i}^{(i)} \\ & W(\alpha, x_2^{(i)}) & & W(\alpha, x_{m_i-1}^{(i)}) & \end{array},$$

where $\forall i, j, i', j'$, $x_j^{(i)} = x_{j'}^{(i')}$ if and only if $i = i'$ and $j = j'$. There are a total of $\sum_{i=1}^{p}(3m_i - 2) = 3n - 2p$ operations. Since each new data value is unique, the

operations in the ith sequence of the original instance appear in sequence order in any legal schedule of the constructed instance. The reader may verify that this constructed instance is a positive instance if and only if the original instance is a positive instance. □

2.2. The VSC problem with two locations. We show that the VSC problem is NP-complete even when only two locations are used.

THEOREM 2.4. *The VSC problem restricted to instances with only two variables is* NP*-complete.*

Proof. Our reduction from 3SAT is depicted in Figure 6. We use two variables, a and b. Variable a is used to select a truth setting. The writes to a in the first sequence set instance variables to *true*; the writes to a in the second sequence set instance variables to *false*. For each instance variable, the second such write establishes the truth setting. Variable b is used to ensure that exactly one assignment is selected per instance variable by forcing both writes to a for this instance variable to be scheduled before any writes to a for the next instance variable.

The three sequences for a clause C_j begin with two reads, corresponding to a particular literal in the clause. The first read can be scheduled only after the variable's truth setting has been established; the second read can then be scheduled if the assignment to the variable matches the assignment needed for the clause.

The final write in the first sequence together with the first read in the third sequence ensure that all of the first and second sequences must be scheduled before any of the third sequence. The $2n$ writes at the end of the third sequence are used to clean up the remaining reads after the satisfiability of the 3SAT instance has been simulated. The reads in the third sequence demand that for each clause, at least one of the writes is scheduled prior to the cleanup: in order to schedule the $R(b, 2n + j)$ operation, we must first schedule a $W(b, 2n+j)$ operation on behalf of clause C_j. This in turn requires that the two reads to a in some sequence for C_j be scheduled prior to the cleanup, and this is possible only if that particular literal in C_j is satisfied.

LEMMA 2.5. *Let $\mathcal{F}$ be an instance of a* 3SAT *problem, and let $\mathcal{V}$ be the instance of the VSC problem constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. We will construct a schedule in which, for each v_i, the latter of $W(a, \mathsf{T}_i)$ and $W(a, \mathsf{F}_i)$ scheduled corresponds to the satisfying truth assignment. Let $T(v_1), \ldots, T(v_n)$ be a satisfying assignment for $\mathcal{F}$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$. We construct the following schedule for $\mathcal{V}$:

1. Repeat the following for $i = 1, 2, \ldots, n$: Consider group i. If $T(v_i) = \mathsf{T}$, schedule $W(a, \mathsf{F}_i)$, then $W(a, \mathsf{T}_i)$. Otherwise, schedule $W(a, \mathsf{T}_i)$, then $W(a, \mathsf{F}_i)$. Next, schedule $W(b, 2i-1)$, $R(b, 2i-1)$, $W(b, 2i)$. Then schedule all $R(b, 2i)$ operations. If $T(v_i) = \mathsf{T}$, schedule all $R(a, \mathsf{T}_i)$ operations. Otherwise, schedule all $R(a, \mathsf{F}_i)$ operations.
2. Schedule $W(a, \alpha)$, $W(b, \beta)$, $R(b, \beta)$.
3. Repeat the following for $p = 1, 2, \ldots, m$: Schedule any $W(b, 2n+p)$ for which both reads above it have been scheduled. Since each clause of $\mathcal{F}$ is satisfied, at least one such write can be scheduled. Then schedule $R(b, 2n + p)$.
4. Finally, we schedule the cleanup writes together with all unscheduled reads in the $3m$ clause sequences. At the very end, schedule all remaining writes to b in these clause sequences.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance. If S is a legal schedule for $\mathcal{V}$, then

First, we have the following three sequences:

$W(a, \mathsf{T}_1)$	$W(a, \mathsf{F}_1)$	$R(b, \beta)$
$W(b, 1)$	$R(b, 1)$	$R(b, 2n+1)$
$R(b, 2)$	$W(b, 2)$	$R(b, 2n+2)$
$W(a, \mathsf{T}_2)$	$W(a, \mathsf{F}_2)$	:
$W(b, 3)$	$R(b, 3)$	$R(b, 2n+m)$
$R(b, 4)$	$W(b, 4)$	$W(a, \mathsf{T}_1)$
:	:	$W(a, \mathsf{F}_1)$
$W(a, \mathsf{T}_n)$	$W(a, \mathsf{F}_n)$	$W(a, \mathsf{T}_2)$
$W(b, 2n-1)$	$R(b, 2n-1)$	$W(a, \mathsf{F}_2)$
$R(b, 2n)$	$W(b, 2n)$	:
$W(a, \alpha)$		$W(a, \mathsf{T}_n)$
$W(b, \beta)$		$W(a, \mathsf{F}_n)$

Then for each clause C_j, say $C_j = v_p \vee \overline{v}_q \vee v_r$, we have the following three sequences:

$R(b, 2p)$	$R(b, 2q)$	$R(b, 2r)$
$R(a, \mathsf{T}_p)$	$R(a, \mathsf{F}_q)$	$R(a, \mathsf{T}_r)$
$W(b, 2n+j)$	$W(b, 2n+j)$	$W(b, 2n+j)$

FIG. 6. *Transforming an instance of* 3SAT *to an instance of VSC with just two locations, a and b. There are n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$ in the* 3SAT *instance. We construct a VSC instance with $3m+3$ sequences. $W(a, d)$ designates a write to location a of the value d; $R(a, d)$ designates a read from location a of the value d. Some data values are expressed as arithmetic expressions, e.g., $2n+m$, $2n+j$, $2p$, $2q$, and $2r$, indicating a value that is the result of evaluating the expression using the values of n, m, j, p, q, and r appropriate to the overall instance, the particular clause, or the particular variable.*

let $\mathcal{T}$ be the truth assignment corresponding to, for each v_i, the latter of $W(a, \mathsf{T}_i)$ and $W(a, \mathsf{F}_i)$ (from the first two sequences) scheduled. Let $S = S_1 S_2 S_3$, where S_1 is the prefix of S up to and including $W(b, \beta)$ and S_3 is the suffix of S starting with $W(a, \mathsf{T}_1)$ from the cleanup. All operations in the first two sequences are in S_1, but none of the last sequence is in S_1. Since the value of b is β at the end of S_1 and any writes to b not in S_1 write values in $[2n+1..2n+m]$, all of the reads of b in the clause sequences are in S_1.

Consider a clause C_p and its three sequences. If none of the three reads to a in these sequences are in S_1, then since the value of a is α at the end of S_1, all must be in S_3. But therefore all of the writes $W(b, 2n+p)$ are also in S_3. However, $R(b, 2n+p)$ must be in S_2, a contradiction. Consider a literal v_i in C_p (the case of $\overline{v}_i$ is symmetric). Since any $R(b, 2i)$ must be after the $W(b, 2i)$, which in turn must be after $W(b, 2i-1)$, this implies that $R(a, \mathsf{T}_i)$ must be after both $W(a, \mathsf{T}_i)$ and $W(a, \mathsf{F}_i)$. Therefore, in the three sequences of C_p, at least one of the reads to a in S_1 must read the value written by the second of these two writes. It follows that C_p is satisfied by T. □

Since the above transformation can be done in polynomial time, Theorem 2.4 follows. □

A simple modification of the previous construction shows that the VSC problem is NP-complete even when both the number of locations and the number of operations per sequence are small constants.

COROLLARY 2.6. *The VSC problem restricted to instances with only two variables is* NP*-complete, even if each sequence contains at most three memory operations.*

Proof. Since we are restricted to only two variables, the general reduction of Observation 2.3 cannot be applied. Instead, we divide the first sequence in Figure 6, with its $3n+2$ operations, into $n+1$ sequences, one with the first two operations only (i.e., creating the sequence $W(a, \mathsf{T}_1)$, $W(b, 1)$), and the remaining n with subsequent sets of three operations. We divide the second sequence in Figure 6, with its $3n$ operations, into n sequences of three operations each. Finally, we replace the last sequence in Figure 6, with its $2n + m + 1$ operations, with the following $m + n$ sequences of three operations each: the sequence $R(b, \beta)$, $R(b, 2n + 1)$, $W(a, \alpha_2)$; for $i = 2, \ldots, m - 1$, the sequence $R(a, \alpha_i)$, $R(b, 2n + i)$, $W(a, \alpha_{i+1})$; the sequence $R(a, \alpha_m)$, $R(b, 2n + m)$, $W(b, \beta_2)$; and finally, for $i = 1, \ldots, n$, the sequence $R(b, \beta_2)$, $W(a, \mathsf{T}_i)$, $W(a, \mathsf{F}_i)$. The reader may verify that this new construction is a positive instance if and only if the construction in Figure 6 is a positive instance. Thus the corollary follows from the proof of Theorem 2.4. □

2.3. The VSC problem for three processors. Many multiprocessors have only a small number of processors, e.g., 8, 16, or 32. We have shown that the VSC problem with $O(n)$ processors is NP-complete; in this section, we show that the VSC problem with just three processors is still NP-complete.

The previous NP-completeness proofs in this paper use a disjoint set of processors for each clause; some also use a disjoint set of processors for each variable. The difficulty in proving an NP-completeness result for a small fixed number of processors is that we do not have as much freedom to schedule operations in an arbitrary order, since the total order on operations at a processor must be respected by any legal schedule. Since processors are a scarce resource, care must be taken to ensure that the construction permits steady progress through each processor sequence, unless the intent is to construct a negative instance. In particular, for each read operation r, there is a corresponding write operation that can be scheduled closely after the operation preceding r at its processor.

Our reduction is from POSITIVE ONE-IN-THREE 3SAT, a variant of 3SAT in which no clause contains a negated literal and we seek a truth assignment such that each clause has exactly one true literal (and hence two false literals). This problem is known to be NP-complete [13]. We construct the instance of the VSC problem, using three processors, depicted in Figure 7.

The idea behind this construction is that the desired truth assignment is the second scheduled write to each v_i. Any legal schedule proceeds in stages, enforced by the seven-operation construction marked $(*)$. For each clause, each of the three processors is satisfied by a particular one-in-three assignment. The subtle part of the construction are the writes marked $(**)$. For any of the three ways to satisfy this clause, this construction frees up the other two processors (by negating variables), yet returns all variables to their original setting (for the next clause). Conversely, for any assignment that does not satisfy this clause, there is no legal scheduling of the operations in this stage.

THEOREM 2.7. *The VSC problem restricted to three processors is* NP*-complete.*

Proof. Let $\mathcal{F}$ be an instance of a POSITIVE ONE-IN-THREE 3SAT problem, and let $\mathcal{V}$ be the instance of the VSC problem constructed as depicted in Figure 7. We will show that $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is a positive instance.

Suppose $\mathcal{F}$ is one-in-three satisfiable. We will construct a schedule in which, for each i, the second write to v_i scheduled corresponds to the satisfying truth assignment.

$\underline{P1}$	$\underline{P2}$	$\underline{P3}$	
$W(v_1, \mathsf{T})$	$W(v_1, \mathsf{F})$		
$\cdots$	$\cdots$		
$W(v_n, \mathsf{T})$	$W(v_n, \mathsf{F})$		
$W(z, 1)$	$W(z, 2)$	$R(z, 1)$	$(*)$
		$R(z, 2)$	$(*)$
$R(z, 3)$	$R(z, 3)$	$W(z, 3)$	$(*)$
$R(v_{p_1}, \mathsf{T})$	$R(v_{q_1}, \mathsf{T})$	$R(v_{r_1}, \mathsf{T})$	(C_1)
$R(v_{q_1}, \mathsf{F})$	$R(v_{r_1}, \mathsf{F})$	$R(v_{p_1}, \mathsf{F})$	(C_1)
$R(v_{r_1}, \mathsf{F})$	$R(v_{p_1}, \mathsf{F})$	$R(v_{q_1}, \mathsf{F})$	(C_1)
$W(v_{p_1}, \mathsf{F})$	$W(v_{q_1}, \mathsf{F})$	$W(v_{r_1}, \mathsf{F})$	$(**)$
$W(v_{q_1}, \mathsf{T})$	$W(v_{r_1}, \mathsf{T})$	$W(v_{p_1}, \mathsf{T})$	$(**)$
	$\cdots$		
$W(z, 3m-2)$	$W(z, 3m-1)$	$R(z, 3m-2)$	$(*)$
		$R(z, 3m-1)$	$(*)$
$R(z, 3m)$	$R(z, 3m)$	$W(z, 3m)$	$(*)$
$R(v_{p_m}, \mathsf{T})$	$R(v_{q_m}, \mathsf{T})$	$R(v_{r_m}, \mathsf{T})$	(C_m)
$R(v_{q_m}, \mathsf{F})$	$R(v_{r_m}, \mathsf{F})$	$R(v_{p_m}, \mathsf{F})$	(C_m)
$R(v_{r_m}, \mathsf{F})$	$R(v_{p_m}, \mathsf{F})$	$R(v_{q_m}, \mathsf{F})$	(C_m)
$W(v_{p_m}, \mathsf{F})$	$W(v_{q_m}, \mathsf{F})$	$W(v_{r_m}, \mathsf{F})$	$(**)$
$W(v_{q_m}, \mathsf{T})$	$W(v_{r_m}, \mathsf{T})$	$W(v_{p_m}, \mathsf{T})$	$(**)$

FIG. 7. *Transforming an instance of* POSITIVE ONE-IN-THREE 3SAT *to an instance of VSC. There are* n *variables,* $v_1, \ldots, v_n$, *and* m *clauses,* $C_1, \ldots, C_m$, *where* $C_i = (v_{p_i}, \mathsf{T}) \vee (v_{q_i}, \mathsf{T}) \vee (v_{r_i}, \mathsf{T})$, *for* p_i, q_i, *and* $r_i \in \{1, 2, \ldots, n\}$.

Let $T(v_1), \ldots, T(v_n)$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$, be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. First, for $i = 1, 2, \ldots, n$, the pair $W(v_i, \mathsf{T})$, $W(v_i, \mathsf{F})$ if $T(v_i) = \mathsf{F}$, or the pair $W(v_i, \mathsf{F})$, $W(v_i, \mathsf{T})$ if $T(v_i) = \mathsf{T}$;
2. then $W(z, 1)$, $W(z, 2)$, $R(z, 1)$, $R(z, 2)$, $W(z, 3)$, $R(z, 3)$, $R(z, 3)$.
3. Since clause $C_1 = (v_{p_1}, \mathsf{T}) \vee (v_{q_1}, \mathsf{T}) \vee (v_{r_1}, \mathsf{T})$ is satisfied by a one-in-three assignment, exactly one of the following is true: (1) $T(v_{p_1}) = \mathsf{T}$, $T(v_{q_1}) = \mathsf{F}$, and $T(v_{r_1}) = \mathsf{F}$; (2) $T(v_{p_1}) = \mathsf{F}$, $T(v_{q_1}) = \mathsf{T}$, and $T(v_{r_1}) = \mathsf{F}$; or (3) $T(v_{p_1}) = \mathsf{F}$, $T(v_{q_1}) = \mathsf{F}$, and $T(v_{r_1}) = \mathsf{T}$. Suppose the first case holds (the other cases follow by symmetry). Schedule $R(v_{p_1}, \mathsf{T})$, $R(v_{q_1}, \mathsf{F})$, $R(v_{r_1}, \mathsf{F})$, $W(v_{p_1}, \mathsf{F})$, and $W(v_{q_1}, \mathsf{T})$ from $P1$; then $R(v_{q_1}, \mathsf{T})$, $R(v_{r_1}, \mathsf{F})$, $R(v_{p_1}, \mathsf{F})$, $W(v_{q_1}, \mathsf{F})$, and $W(v_{r_1}, \mathsf{T})$ from $P2$; then $R(v_{r_1}, \mathsf{T})$, $R(v_{p_1}, \mathsf{F})$, $R(v_{q_1}, \mathsf{F})$, $W(v_{r_1}, \mathsf{F})$, and $W(v_{p_1}, \mathsf{T})$ from $P3$. Note that at this point, the current "value" of each v_i is the same as it was before this step of the construction.
4. Repeat the construction of the previous two steps for clauses $C_2, \ldots, C_m$ in an analogous manner.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive VSC instance. If S is a legal schedule for $\mathcal{V}$, then the second value written to each v_i will be our satisfying assignment. For $i = 1, 2, \ldots, n$, let $T(v_i) = \mathsf{T}$ if the first $W(v_i, \mathsf{F})$ from $P1$ precedes in S the first $W(v_i, \mathsf{T})$ from $P2$; otherwise, let $T(v_i) = \mathsf{F}$. A simple induction establishes that any prefix S_j of S ending in $W(z, 3j)$, $j = 1, \ldots, m$, contains precisely the following

events: all events in the prefix of $P1$ ending in $W(z, 3j-2)$, all events in the prefix of $P2$ ending in $W(z, 3j-1)$, and all events in the prefix of $P3$ ending in $W(z, 3j)$.

For $j = 1, \ldots, m$, consider the operations in $P1$, $P2$, and $P3$ in S immediately following S_j:

$$\begin{array}{lll} R(z,3j) & R(z,3j) & \\ R(v_{p_j},\mathsf{T}) & R(v_{q_j},\mathsf{T}) & R(v_{r_j},\mathsf{T}) \\ R(v_{q_j},\mathsf{F}) & R(v_{r_j},\mathsf{F}) & R(v_{p_j},\mathsf{F}) \\ R(v_{r_j},\mathsf{F}) & R(v_{p_j},\mathsf{F}) & R(v_{q_j},\mathsf{F}) \end{array} .$$

Due to these reads, it is not possible to extend S_j to a legal schedule unless the current "value" of exactly one of v_{p_j}, v_{q_j}, and v_{r_j} is T. Assume that v_{p_j} is T, while v_{q_j} and v_{r_j} are F. A careful inspection of the operations for C_j reveals that the following is a subsequence of S_{j+1} after S_j: $R(v_{p_j},\mathsf{T})$, $R(v_{q_j},\mathsf{F})$, $R(v_{r_j},\mathsf{F})$, $W(v_{p_j},\mathsf{F})$, $W(v_{q_j},\mathsf{T})$ (all from $P1$), $R(v_{q_j},\mathsf{T})$, $R(v_{r_j},\mathsf{F})$, $R(v_{p_j},\mathsf{F})$, $W(v_{q_j},\mathsf{F})$, $W(v_{r_j},\mathsf{T})$ (from $P2$), $R(v_{r_j},\mathsf{T})$, $R(v_{p_j},\mathsf{F})$, $R(v_{q_j},\mathsf{F})$, $W(v_{r_j},\mathsf{F})$, $W(v_{p_j},\mathsf{T})$ (from $P3$). This subsequence may be interleaved with the remaining operations in $S_{j+1} - S_j$, namely, $R(z,3j)$, $R(z,3j)$, $W(z,3j+1)$, $W(z,3j+2)$, $R(z,3j+1)$, $R(z,3j+2)$, and $W(z,3(j+1))$. By symmetry, these properties also hold in the case where v_{q_j} is T, while v_{r_j} and v_{p_j} are F, or the case where v_{r_j} is T, while v_{p_j} and v_{q_j} are F.

We claim that for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, the "value" of v_i through S_j is $T(v_i)$. This is established via a simple induction, where the base case, $j = 1$, holds by definition. Moreover, by the characterization of S_{j+1} of the previous paragraph, it follows by inspection that for $i = 1, \ldots, n$, the "value" of v_i through S_{j+1} is the same as through S_j.

It follows that $T(v_1), \ldots, T(v_n)$ is a satisfying one-in-three assignment for each clause in $\mathcal{F}$, and hence $\mathcal{F}$ is one-in-three satisfiable. □

The construction depicted in Figure 7 uses only $n+1$ locations. Alternatively, it can be modified to have each location assume at most two values.

2.4. Comparison with serializability. The VSC problem is reminiscent of the serializability problem for database transactions. The most similar variant is that of view serializability. In the *view serializability* problem [31], we are given a *history* H, i.e., a total order on a set of reads and writes, where each read or write is associated with a particular database transaction, and each read or write contains an address but not a value. Each read is assumed to read from the last preceding write in H to the same address. The task is to determine if there is a total order on the *transactions* that preserves this mapping of reads to writes. The view serializability problem is NP-complete [31].

The VSC problem differs from the view serializability problem in at least four ways. First, in the VSC problem, legal schedules may interleave operations from different processors: the sequence of operations at a processor must be in order but need not be consecutive in a legal schedule. For example, the instance in Figure 1 is a positive VSC instance, but a negative instance for view serializability: both $S = S_1S_2$ and $S = S_2S_1$ have reads-from violations. Second, the *input* to a view serializability problem, a consistent total order of the reads and writes, is the desired *output* of the VSC problem. Third, when a database system is correctly enforcing serializability, each transaction that is not aborted will view an unchanged database during the course of its operations. Thus a single read of an address suffices to learn the value in that location for the duration of the transaction, and in general, it may be assumed that within each transaction, there is at most one read and one write to each location.

These restrictions are not appropriate for the VSC problem since communicating processors exchange values by writing and reading memory. Fourth, the VSC problem does not provide a mapping of reads to writes that must be preserved. However, in section 4, we will consider a variant of the VSC problem (the VSC-read problem) in which the input includes such a mapping, and a legal schedule must preserve this mapping.

For the VSC problem, we observe that the ability to interleave input sequences is not all that helpful unless the number of processors is bounded. Specifically, the construction in Observation 2.3 converts any VSC instance into one in which there is a legal schedule if and only if there is a legal schedule such that each processor sequence is a consecutive subsequence of the schedule. Thus the VSC problem is NP-complete even when interleaving is not permitted. On the other hand, if the number of processors is bounded, then interleaving is quite powerful. Whereas the VSC problem with three processors is NP-complete, there is a trivial linear-time algorithm for serializability when the number of transactions is restricted to a constant k: with k "processors," there are only a constant number, $k!$, of possible serializations to check.

3. Verifying linearizability. In this section, we present results for testing linearizability. We begin with some preliminary remarks in section 3.1, including an efficient reduction from the VL problem to the VSC problem. In section 3.2, we show that the VL problem is NP-complete. (The NP-completeness proof together with the reduction can be used for an alternative proof of Corollary 2.6.) Then in section 3.3, we present a polynomial-time algorithm for the VL problem with $O(\log n)$ processors.

3.1. Preliminaries. As discussed in section 1, linearizability is more restrictive than sequential consistency, in that a legal schedule must respect the time intervals for the operations. This added constraint makes implementations of linearizability provably slower than implementations of sequential consistency [8]. In the interest of memory system performance, shared-memory multiprocessors such as the Kendall Square KSR1 [12] support sequential consistency instead of linearizability. On the other hand, linearizability has the advantage over sequential consistency that each address, or, more generally, each shared data object, can be considered in isolation. Herlihy and Wing [22] proved that a system is linearizable if and only if each object in the system is linearizable. Thus linearizable objects can be implemented, verified, and executed independently. In this paper, we use the following implication of the Herlihy and Wing theorem.

FACT 3.1. *$\mathcal{V}$ is a positive instance of the VL problem if and only if, for each address a, the subinstance of $\mathcal{V}$ comprised solely of the operations on a is a positive instance.*

Thus as indicated in Table 1, we can assume without loss of generality that a VL instance has but a single location. This assumption does not alter the complexity of the problem: if $f(n)$ is the running time on an instance of size n, then since $f(n) \geq n$, $f(n) = f(\sum_{i=1}^{k} n_i) \geq \sum_{i=1}^{k} f(n_i)$, where $n_1, \ldots, n_k$ are the respective sizes of the subinstances on each of the k locations.

Note as well that if the number of processors is not bounded, then we can also assume without loss of generality that a VL instance has but a single operation per processor, as indicated in Table 1. This follows since the intervals for operations by a single processor do not overlap, so the total order between them is enforced whether or not they are considered to be part of the same processor sequence.

A VL instance can be reduced to a VSC instance that uses additional operations that reflect the scheduling constraints defined by the start-of-interval and end-of-interval times. Specifically, we have the following reduction from the VL problem to the VSC problem.

THEOREM 3.2. *There is an $O(n \log n)$-time reduction from the VL problem with n operations and k variables to k instances of the VSC problem with two variables, at most three memory operations per sequence, and a total of $O(n)$ operations over all instances.*

Proof. By Fact 3.1, it suffices to construct distinct VSC instances for each of the k variables in the VL instance. Consider one such variable, a, and let $\mathcal{V}$ be the subinstance comprised of the operations on a, with n_a operations. Let $t_1, t_2, \ldots, t_{n_a}$ be the end-of-interval times in $\mathcal{V}$ in increasing order. We construct the following VSC instance $\mathcal{V}'$ using two types of sequences. For each operation $read(a, d, t_i, t_j)$ or $write(a, d, t_i, t_j)$ in $\mathcal{V}$, we have in $\mathcal{V}'$ the following type I sequence: (1) $read(b, \tau)$, where τ is the largest end-of-interval time in $\mathcal{V}$ less than t_i (if any); (2) $read(a, d)$ or $write(a, d)$; and (3) $write(b, t_j)$. In addition, for each end-of-interval time t_j, $j < n_a$, we have a type II sequence (1) $read(b, t_j)$, (2) $read(b, t_{j+1})$. Clearly, $\mathcal{V}'$ is a VSC instance with two variables, at most three memory operations per sequence, and $O(n_a)$ operations. We will show that the operations on b encode the scheduling constraints defined by the start-of-interval and end-of-interval times.

We begin by showing that if two operations have nonoverlapping intervals in $\mathcal{V}$, then the order between them is respected by any legal schedule for $\mathcal{V}'$.

LEMMA 3.3. *Consider any two operations π_i and π_j in $\mathcal{V}$ and the corresponding operations π_i' and π_j' in $\mathcal{V}'$. If the end-of-interval time for π_i is less than the start-of-interval time for π_j, then π_i' precedes π_j' in any legal schedule for $\mathcal{V}'$.*

Proof. Let S' be a legal schedule for $\mathcal{V}'$. Consider the subsequence S_r' of S' of all $read(b, t)$ operations in $\mathcal{V}'$. Since there is exactly one $write(b, t)$ operation for each end-of-interval t, all $read(b, t)$ operations for the given t are consecutive in S_r' and follow this $write(b, t)$ operation in S'. Moreover, due to the type II sequences, the $read(b, t)$ operations in S_r' are in nondecreasing order of t. If the end-of-interval time t_i for π_i is less than the start-of-interval time t_j for π_j, then there is a $read(b, t_k)$ operation preceding π_j' in a type I sequence such that $t_k \geq t_i$. It follows that π_i' precedes $write(b, t_i)$ precedes $read(b, t_i)$ precedes $read(b, t_k)$ precedes π_j' in S'. ∎

We now prove the correctness of our construction.

LEMMA 3.4. *$\mathcal{V}'$ is a positive VSC instance if and only if $\mathcal{V}$ is a positive VL instance.*

Proof. Suppose $\mathcal{V}$ is a positive VL instance, and let A be an assignment of times for a legal schedule for $\mathcal{V}$. For each operation π' in $\mathcal{V}'$, define $A'(\pi')$ as follows:

1. If π' is an operation on a, then $A'(\pi') = A(\pi)$, where π is the corresponding operation in $\mathcal{V}$.
2. If π' is an operation on b with data value t, then $A'(\pi') = t$.

Let S be a sequence of the operations in $\mathcal{V}'$ in nondecreasing order according to A' such that among operations with the same A' value, the operation on a (if any) precedes the write operation on b precedes any read operations on b. We claim that S is a legal schedule for $\mathcal{V}'$. First, observe that any type II sequence appears in order in S. Moreover, consider the type I sequence constructed for an operation $\pi = read(a, d, t_1, t_2)$ or $write(a, d, t_1, t_2)$ in $\mathcal{V}$: $read(b, \tau)$ (if any), $\pi' = read(a, d)$ or $write(a, d)$, and $write(b, t_2)$. By construction and since A corresponds to a legal

schedule,

$$\tau < t_1 \leq A(\pi) = A'(\pi') \leq t_2,$$

and hence the type I sequence also appears in order in S. Thus S is an interleaving of the individual sequences. Second, there are no reads-from violations on a since the order of operations on a corresponds to a legal schedule for $\mathcal{V}$. Moreover, there are no reads-from violations on b since for each $read(b, \tau)$ operation, τ is an end-of-interval time, and hence a $write(b, \tau)$ operation is the last preceding write operation in S.

Conversely, suppose $\mathcal{V}'$ is a positive VSC instance, and let S' be a legal schedule for $\mathcal{V}'$. Let $S'_a = \pi'_1, \pi'_2, \ldots, \pi'_{n_a}$ be the subsequence of S' comprised of the operations on a. Let $S_a = \pi_1, \pi_2, \ldots, \pi_{n_a}$ be the corresponding operations in $\mathcal{V}$. Since S' is legal, there are no reads-from violations in S_a. Let ϵ_0 be the minimum difference between any pair of times (start-of-interval or end-of-interval) in $\mathcal{V}$; let $\epsilon = \epsilon_0/n$. We assign times to operations in S_a inductively as follows. The first operation π_1 is assigned a time equal to its start-of-interval time. For $k = 2, \ldots, n_a$, the operation π_k in S_a is assigned a time equal to the maximum of its start-of-interval time and ϵ greater than the time assigned to π_{k-1}. We claim that even in this latter case, the time assigned to π_k is within its interval. To see this, let π_i, $1 \leq i < k$, be the last operation preceding π_k in S_a that is assigned a time equal to its respective start-of-interval time t_i. The time assigned to π_k is $t_i + (k-i)\epsilon$. Since π'_i precedes π'_k in S', it follows by Lemma 3.3 that the start-of-interval time for π_i is less than the end-of-interval time for π_k by at least ϵ_0. Thus since

$$t_i + (k-i)\epsilon < t_i + n\epsilon = t_i + \epsilon_0,$$

the time assigned to π_k is within its interval. □

As the reader may verify, this transformation can be done in $O(n \log n)$ time. Theorem 3.2 follows. □

3.2. The VL problem is NP-complete. In this section, we prove the NP-completeness of the VL problem by a reduction from the Satisfiability problem (SAT) [13]. By Fact 3.1, it is necessary to consider a reduction based on a single location. The VL problem is in NP since given an assignment of times, we can test in polynomial time that each operation is assigned a time within its interval and that the schedule defined by the assignment has no reads-from violations.

THEOREM 3.5. *The VL problem is* NP*-complete.*

Proof. Consider an instance $\mathcal{F}$ of SAT with n variables, $v_1, v_2, \ldots, v_n$, and m clauses, $C_1, C_2, \ldots, C_m$. Without loss of generality, assume that each variable and its negation appear in at least one clause, but not the same clause, and that there are no repeated variables in a clause. We construct an instance of the VL problem with at most $5nm + 4n + m$ operations, corresponding to $\mathcal{F}$. To simplify the description, the construction has multiple operations sharing the same start-of-interval or end-of-interval times; these ties can be broken arbitrarily to ensure unique time values. In particular, we use only integral times in our simplified description, with at most $m + 2$ intervals sharing the same start or end time, so unique positive rationals can be readily selected to break any ties.

Figure 8 depicts an example construction. For each clause C_j, $j = 1, \ldots, m$, we have a $read(a, c_j, 1, 3n{+}1)$ operation, denoted the *clause read* for C_j. For each variable v_i, $i = 1, \ldots, n$, assignment to v_i is simulated using two operations: $write(a, i, 3i - 1, 3i)$ and $write(a, \bar{\imath}, 3i - 1, 3i)$.

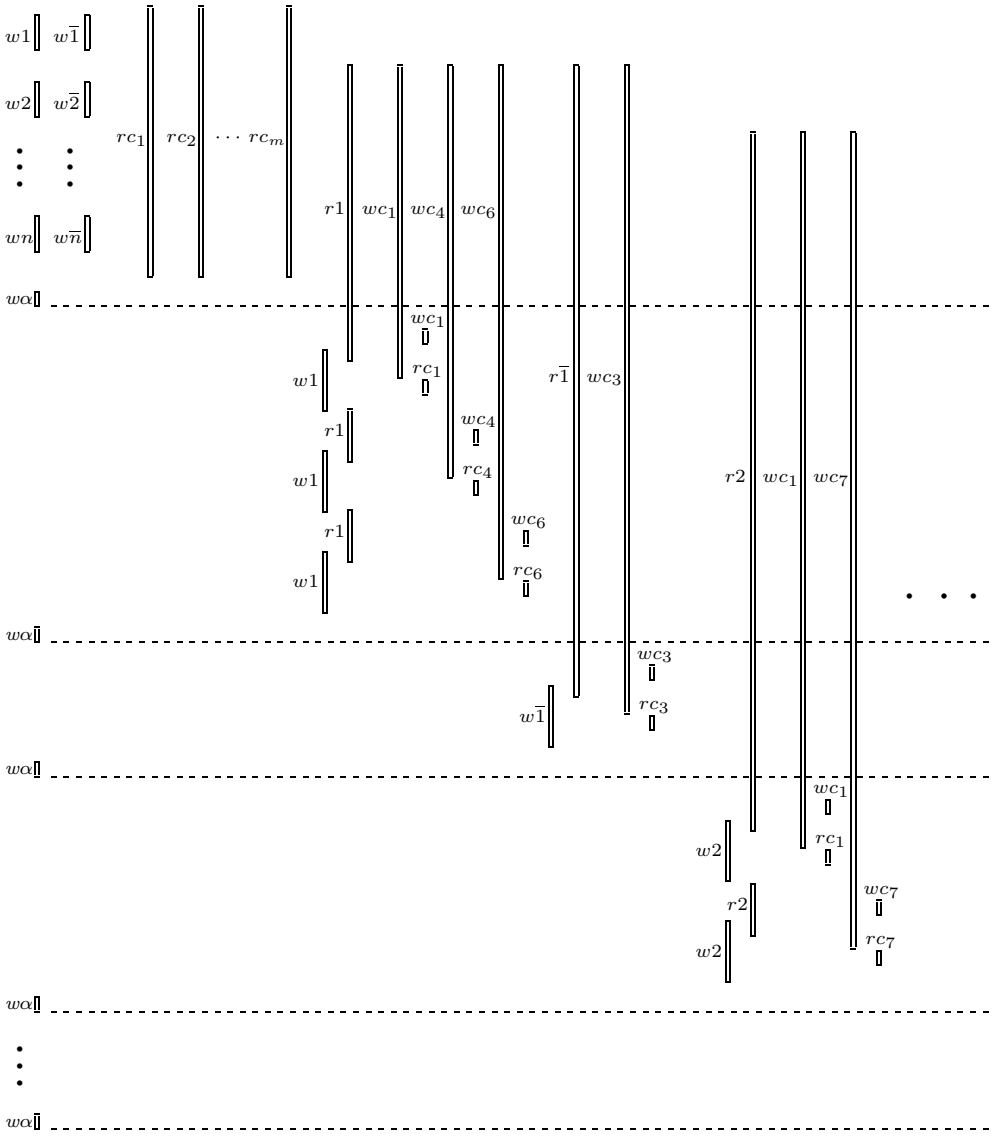


FIG. 8. *Transforming an instance of* SAT *to an instance of VL with a single location. There are* n *variables,* $v_1, \ldots, v_n$*, and* m *clauses,* $C_1, \ldots, C_m$*. The full construction has at most* $5nm+4n+m$ *operations. Here the literal* v_1 *appears in exactly clauses* C_1*,* C_4*, and* C_6*, the literal* $\overline{v}_1$ *appears in clause* C_3 *only, the literal* v_2 *appears in exactly clauses* C_1 *and* C_7*, and so forth. Every column corresponds to a processor sequence. Vertical boxes depict the intervals of time for the respective read* (r) *or write* (w) *operations to the single location; time progresses from top to bottom in the figure. The number or symbol following each* r *or* w *indicates the value read or written.*

We have $2n$ write operations, used to partition the problem into $2n$ phases, one for each literal, as follows. Consider $i = 1, \ldots, n$. Let m_i ($m_{\overline{i}}$) be the number of clauses containing the literal v_i ($\overline{v}_i$, respectively). By assumption, $m_i > 0$, $m_{\overline{i}} > 0$, and $m_i + m_{\overline{i}} \leq m$. Let $\Delta_i = (7m+4)(i-1) + 3n + 3$ and let $\Delta_{\overline{i}} = \Delta_i + 7m_i + 2$. There is a $write(a, \alpha, \Delta_i - 1, \Delta_i)$ and a $write(a, \alpha, \Delta_{\overline{i}} - 1, \Delta_{\overline{i}})$.

There is a set of operations for each literal v_i, denoted the *group* of operations for i. For $i = 1, \ldots, n$, if $C_{i_1}, C_{i_2}, \ldots, C_{i_{m_i}}$ are the clauses containing the literal v_i, we have the following $5m_i$ intervals. First, for C_{i_1}, we have

- $ri^{(1)} = read(a, i, 3i + 1, \Delta_i + 4)$,
- $wi^{(1)} = write(a, i, \Delta_i + 3, \Delta_i + 7)$,
- $wc_{i_1}^{(1)} = write(a, c_{i_1}, 3i + 1, \Delta_i + 5)$,
- $wc_{i_1}^{(2)} = write(a, c_{i_1}, \Delta_i + 1, \Delta_i + 2)$,
- $rc_{i_1} = read(a, c_{i_1}, \Delta_i + 5, \Delta_i + 6)$.

Then for C_{i_k}, $k = 2, \ldots, m_i$, we have

- $ri^{(k)} = read(a, i, \Delta_i + 7(k-1), \Delta_i + 7(k-1) + 4)$,
- $wi^{(k)} = write(a, i, \Delta_i + 7(k-1) + 3, \Delta_i + 7(k-1) + 7)$,
- $wc_{i_k}^{(1)} = write(a, c_{i_k}, 3i + 1, \Delta_i + 7(k-1) + 5)$,
- $wc_{i_k}^{(2)} = write(a, c_{i_k}, \Delta_i + 7(k-1) + 1, \Delta_i + 7(k-1) + 2)$,
- $rc_{i_k} = read(a, c_{i_k}, \Delta_i + 7(k-1) + 5, \Delta_i + 7(k-1) + 6)$.

The operations $wc_{i_1}^{(1)}, wc_{i_2}^{(1)}, \ldots, wc_{i_{m_i}}^{(1)}$ are denoted the *clause writes* for v_i.

Likewise, there is a set of operations for each literal $\overline{v}_i$, denoted the *group* of operations for $\bar{\imath}$. For $i = 1, \ldots, n$, if $C_{\bar{i}_1}, C_{\bar{i}_2}, \ldots, C_{\bar{i}_{m_{\bar{i}}}}$ are the clauses containing the literal $\overline{v}_i$, we have the $5m_{\bar{i}}$ intervals obtained from the previous definition by replacing $\Delta_{\bar{\imath}}$ with Δ_{i+1} and leaving "$3i + 1$" unchanged but otherwise replacing i with $\bar{\imath}$ throughout.

This completes the construction.

The idea behind the construction is the following. Consider a variable v_1, and refer to Figure 8. Recall that all reads and writes are to the same location. Both $w1$ and $w\overline{1}$ on the left must be scheduled before any $r1$ or $r\overline{1}$; the second write scheduled corresponds to the truth assignment. If $w1$ is scheduled second, then the first $r1$ can be scheduled, followed by the clause writes wc_1, wc_4, and wc_6 for v_1. If all clauses can be satisfied by the truth setting, then the set of all clause writes will ensure that all m clause reads can be scheduled during their common interval. Consider the three $w1$ operations and the two $r1$ operations below the first dashed line, together with the $r1$ operation that crosses this line. Since the crossing $r1$ has been scheduled, the two $r1$ operations below the line can pair up with the first two $w1$ operations; this in turn permits the rc_1 (below the line) to be paired with the wc_1 directly above it. On the other hand, if $w\overline{1}$ had been scheduled second instead, then the first $w1$ is paired with the crossing $r1$. Hence the first $w1$ is scheduled between the intervals for the wc_1 and the rc_1 below the line; this implies that a clause write wc_1 is needed below the line. For this reason, all clause writes for a literal not in the truth assignment must be scheduled below the first dashed line. It follows that all clause reads can be scheduled if and only if we have a satisfying truth assignment.

LEMMA 3.6. *Let $\mathcal{F}$ be an instance of a* SAT *problem, and let $\mathcal{V}$ be the instance of the VL problem constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. We will construct a schedule in which, for each v_i, the latter of $write(a, i, 3i - 1, 3i)$ and $write(a, \bar{\imath}, 3i - 1, 3i)$ scheduled corresponds to the satisfying truth assignment. Let $T(v_1), \ldots, T(v_n)$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$, be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. Repeat the following for $i = 1, 2, \ldots, n$:
   If $T(v_i) = \mathsf{T}$, schedule $write(a, \bar{\imath}, 3i - 1, 3i)$ at time $3i - 1$, then $write(a, i, 3i - 1, 3i)$ at time $3i$. Consider the $5m_i$ operations in the group for i, together with

the m_i corresponding clause reads, $read(a, c_{i_1}, 1, 3n+1)$ through $read(a, c_{i_{m_i}}, 1, 3n+1)$. Schedule $ri^{(1)} = read(a, i, 3i+1, \Delta_i+4)$ at time $3i+1$. Then after time $3i+1$ and before time $3i+2$, for $k = 1, \ldots, m_i$, schedule each of m_i clause writes paired with any unscheduled, corresponding clause reads: $wc_{i_k}^{(1)} = write(a, c_{i_k}, 3i+1, \Delta_i + 7(k-1) + 5)$ followed by, if it has not yet been scheduled, $read(a, c_{i_k}, 1, 3n+1)$.

The case where $T(v_i) = \mathsf{F}$ is symmetric, and left to the reader.

2. Since $T(v_1), \ldots, T(v_n)$ is a satisfying assignment for $\mathcal{F}$, all clause reads have been scheduled by this point. Schedule $write(a, \alpha, \Delta_1 - 1, \Delta_1)$ at time Δ_1.
3. Repeat the following for $i = 1, 2, \ldots, n$:
   If $T(v_i) = \mathsf{T}$, consider the $4m_i - 1$ unscheduled operations in group i and the $5\overline{m}_i$ unscheduled operations in group $\bar{\imath}$:
   (a) Schedule $wc_{i_1}^{(2)}$ at time $\Delta_i + 2$, then rc_{i_1} at time $\Delta_i + 5$, then $wi^{(1)}$ at time $\Delta_i + 7$.
   (b) Then repeat for $k = 2, \ldots, m_i$: Schedule $ri^{(k)}$ at time $\Delta_i + 7(k-1) + 1$, then $wc_{i_k}^{(2)}$ at time $\Delta_i + 7(k-1) + 2$, then rc_{i_k} at time $\Delta_i + 7(k-1) + 5$, and finally $wi^{(k)}$ at time $\Delta_i + 7(k-1) + 7$.
   (c) This completes group i. Schedule $write(a, \alpha, \Delta_{\bar{\imath}} - 1, \Delta_{\bar{\imath}})$ at time $\Delta_{\bar{\imath}}$.
   (d) Schedule $wc_{\bar{\imath}_1}^{(2)}$ at time $\Delta_{\bar{\imath}} + 1$, then $w\bar{\imath}^{(1)}$ at time $\Delta_{\bar{\imath}} + 3$, then $r\bar{\imath}^{(1)}$ at time $\Delta_{\bar{\imath}} + 4$, then $wc_{\bar{\imath}_1}^{(1)}$ at time $\Delta_{\bar{\imath}} + 5$, and finally $rc_{\bar{\imath}_1}$ at time $\Delta_{\bar{\imath}} + 6$.
   (e) Then repeat for $k = 2, \ldots, m_{\bar{\imath}}$: Schedule $wc_{\bar{\imath}_k}^{(2)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 1$, then $w\bar{\imath}^{(k)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 3$, then $r\bar{\imath}^{(k)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 4$, then $wc_{\bar{\imath}_k}^{(1)}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 5$, and finally $rc_{\bar{\imath}_k}$ at time $\Delta_{\bar{\imath}} + 7(k-1) + 6$.
   (f) This completes group $\bar{\imath}$. Schedule $write(a, \alpha, \Delta_{i+1} - 1, \Delta_{i+1})$ at time Δ_{i+1}.

   The case where $T(v_i) = \mathsf{F}$ is symmetric and left to the reader.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance. If $\mathcal{S}$ is a legal schedule for $\mathcal{V}$, then let $\mathcal{T}$ be the truth assignment defined as follows: for each v_i, if $write(a, \bar{\imath}, 3i-1, 3i)$ precedes $write(a, i, 3i-1, 3i)$ in $\mathcal{S}$, then $\mathcal{T}(v_i) = \mathsf{T}$; otherwise, $\mathcal{T}(v_i) = \mathsf{F}$.

We observe the following for $i = 1, \ldots, n$: Both $write(a, i, 3i-1, 3i)$ and $write(a, \bar{\imath}, 3i-1, 3i)$ precede both $ri^{(1)}$ and $r\bar{\imath}^{(1)}$ in $\mathcal{S}$. Thus if $\mathcal{T}(v_i) = \mathsf{F}$, then while $r\bar{\imath}^{(1)}$ may be scheduled in $\mathcal{S}$ prior to time $3i+3$, $ri^{(1)}$ cannot be. The $ri^{(1)}$ operation must be scheduled after the only other operation that writes i prior to the end of its interval, namely $wi^{(1)}$. Hence $wc_{i_1}^{(2)}$ precedes $wi^{(1)}$ precedes $ri^{(1)}$ precedes rc_{i_1}. But this implies that for $k = 2, \ldots, m_i$, $wc_{i_k}^{(2)}$ precedes $wi^{(k)}$ precedes $ri^{(k)}$ precedes rc_{i_k}. The case where $\mathcal{T}(v_i) = \mathsf{T}$ is symmetric.

Suppose $\mathcal{T}$ does not satisfy a clause C_j. Consider the literals in C_j sorted by their index, and let x be the number of literals in C_j. For $k = 1, \ldots, x$, define $j_k \in \{1, \bar{1}, 2, \bar{2}, \ldots, n, \bar{n}\}$ such that $j_k = i$ (or $\bar{\imath}$) if and only if v_i ($\overline{v}_i$, respectively) is the kth literal in C_j. We claim that all writes of c_j are scheduled in $\mathcal{S}$ after Δ_1. The proof is by induction on decreasing k. Consider the last rc_j in $\mathcal{S}$, in group j_x. Due to the argument given in the preceding paragraph, there is only one write of c_j that could have been scheduled so as to be read by the rc_j: the sole write of c_j in group j_x whose interval is not strictly after Δ_{j_x}. Assume inductively that all writes of c_j from groups j_{k+1} to j_x are scheduled after $\Delta_{j_{k+1}}$. Thus due to the argument given in the preceding paragraph, there is only one write of c_j that could have been scheduled so

as to be read by the rc_j: the sole write of c_j in group j_k whose interval is not strictly after Δ_{j_k}. The claim follows by induction.

Since there are no writes of c_j in $\mathcal{S}$ until after $\Delta_1 = 3n + 3$, but the clause read $read(a, c_j, 1, 3n + 1)$ must be scheduled before Δ_1, $\mathcal{S}$ is not a legal schedule, a contradiction.

Hence $\mathcal{T}$ is a satisfying assignment for $\mathcal{F}$. □

Since the above transformation can be done in polynomial time, Theorem 3.5 follows. □

3.3. The VL problem with few processors. In this section, we describe a polynomial-time algorithm for the VL problem when the number of processors (i.e., sequences) is bounded. Recall that if two operations have nonoverlapping intervals, their order in any legal schedule is predetermined. Thus since we can consider each address in isolation, the number of possible schedules grows with the number of overlapping intervals for operations on that address. The specific result we obtain is the following.

THEOREM 3.7. *The VL problem restricted to instances such that at any time t, there are at most k operations on the same address whose intervals contain t, can be solved in $O(n2^{O(k)} + n \log n)$ time.*

Proof. We consider each address x in isolation. Let S_x be the set of operations on x. Let $t_0 > 0$ be the earliest start-of-interval time and t_∞ be the latest end-of-interval time for an operation in S_x. To simplify the discussion that follows, we augment S_x with two additional writes to x: Let $S'_x = S_x \cup \{write(x, \delta, t_0/3, t_0/2), write(x, \delta, t_\infty + 1, t_\infty + 2)\}$, where δ is a value not appearing in S_x. For each address, we may assume without loss of generality that there are only k processors.

Let G_x be a leveled acyclic digraph, with one level for each operation in S'_x, in order of increasing end-of-interval times, and at most $k2^{k-1}$ vertices per level, as follows. We identify each level by the finishing time t of its corresponding operation α_t. There are at most $k - 1$ additional operations in S'_x with intervals containing t; these may or may not be assigned times greater than t. Consider all possible subsets of these operations; there are at most 2^{k-1} of them. Each vertex at level i specifies one of these subsets, plus a last write to x. The vertex represents a prefix of a schedule of the operations in S'_x, namely, a schedule in which the operations in the subset as well as all operations with start-of-interval times after t are assigned times later than t, and the last write assigned a time no later than t is indicated. Since for a given subset there are at most k possible choices of a last write, we have at most $k2^{k-1}$ nodes. The first level consists of a single node, with last write $write(x, \delta, t_0/3, t_0/2)$; the last level consists of a single node, with last write $write(x, \delta, t_\infty + 1, t_\infty + 2)$.

The edges of G_x are defined as follows. Consider two consecutive levels t and t', and let α and α' be the operations with end-of-interval times t and t', respectively. All intervals, other than α's, that contain time t must contain t'. Consider two vertices in G_x, v at level t and v' at level t'. There is an edge between v and v' if the following conditions hold:

1. Every interval containing both t and t' that was assigned a time no later than t, according to v, is also assigned a time no later than t', according to v'.
2. Let A be the set of operations with intervals containing t' (including possibly α') that were assigned a time later than t (either they do not contain t or they contain t but were assigned a time later than t, according to v) but were assigned a time no later than t', according to v'. Every read in A must either read the value of the last write in v or has a corresponding write of the same value in A. Moreover, the last write in v' must either be in the set A or, if

there are no writes in A, the same last write as in v.

Note that the operations in A can be safely scheduled without reads-from violations in the interval between the last start-of-interval time in A and t'. It follows that there is a directed edge from v to v' if and only if there is a schedule consistent with both v and v'. This leads to the following claim, whose proof is left to the reader.

CLAIM 3.8. *We have a positive VL instance if and only if there is a source-to-sink path in G_x for each address x.*

To construct the graphs for each address, first we sort the operations by address, and within an address, by their starting and finishing times (each operation appears twice in this sorted sequence). Then we test for each pair of vertices on consecutive levels whether a directed edge should be between them. Next, we test for a source-to-sink path in each G_x using depth-first search. If we have a positive instance, we assign times to the operations based on the information in the vertices visited along the source-to-sink paths. Each G_x can be constructed and searched in $O(n_x 2^{O(k)})$ time, where n_x is the number of operations in S_x. Thus the total running time is $O(n \log n + n2^{O(k)})$. □

Since with k processors there can be at most k overlapping intervals, Theorem 3.7 implies, for instance, the following two corollaries.

COROLLARY 3.9. *There is an $O(n \log n)$-time algorithm for the VL problem with any fixed number of processors.*

COROLLARY 3.10. *There is a polynomial-time algorithm for the VL problem with $O(\log n)$ processors.*

4. Providing additional input information. In the previous sections, we have considered the basic VSC and VL problems, in which the only constraints on legal schedules arise from (1) either the order of operations at a processor or the time intervals, and (2) the fact that some write operation with the same address and value must precede each read operation, with no intervening write with the same address but a different value. In this section, we consider variants of the VSC and VL problems in which the input provides additional information on either the pairing of reads to particular writes (section 4.1), the order of writes to a location (section 4.2), the old value overwritten by each write (section 4.3), or the order of all conflicting operations to a location (section 4.4). As shown in Table 1, this additional information helps in some cases, but not in others. We also show how our algorithmic results can be extended to handle atomic read-modify-write operations, with no asymptotic penalty (section 4.5).

Each of the variants considered in this section is motivated by practical considerations in existing multiprocessors. For further details, we refer the reader to [17, 19].

We require that the additional information provided by the memory system as input in these variants be respected in any legal schedule. What happens if the additional information provided is incorrect? If there are anomalies in the additional information, e.g., a read is paired with a write with a different address, we detect this and report a negative instance. If the additional information is incorrect such that a positive instance (ignoring the information) becomes a negative instance, we report a negative instance: there is clearly something wrong with the memory system. The most important property that we require, however, is that the testing procedure must never be persuaded by incorrect (or even correct) additional information that an execution was sequentially consistent or linearizable when in fact it was not. Conversely, if both the execution and the additional information were correct, the testing procedure must report a positive instance.

4.1. Providing the read-mapping. In the basic VSC and VL problems, whenever there are multiple writes with the same address and value, there may be ambiguity as to which of the writes is to be paired with a given read of the same address and value. The question addressed in this section is as follows: If for each read there is no ambiguity as to with which write it is to be paired, do the VSC and VL problems remain NP-complete?

We define the *VSC-read* and *VL-read* problems, in which for each read operation, it is known precisely which write was responsible for the value read; a legal schedule must respect this relation. (A schedule S respects the relation if and only if, for each read in S, the write to which it is mapped is the last preceding write in S with the same address.) The function mapping each read to the responsible write is called a *read-mapping*. A schedule that does not respect the read-mapping has a *reads-from violation*.

VSC-read. We will show that the VSC-read problem is NP-complete by a reduction from view serializability. Recall that in the *view serializability* problem, we are given a *history* H, i.e., a total order on a set of reads and writes, where each read or write is associated with a particular database transaction, and each read or write contains an address but not a value. Each read is assumed to read from the last preceding write in H to the same address. The task is to determine if there is a total order on the *transactions* that preserves the read-mapping in H.

THEOREM 4.1. *The VSC-read problem restricted to instances in which each sequence contains at most three memory operations is* NP*-complete.*

Proof. We begin by showing a reduction to the VSC-read problem with no restrictions on the number of operations in a sequence. Given a history H, an instance of a view serializability problem, we construct an instance of the VSC-read problem as follows. Let α be an address not in H. Let S_i' be the sequence of operations in H for transaction i, where each write operation in a transaction is assigned a unique value to write, and each read operation is assigned the value of the last previous write in H to the same address. Let $S_i = W(\alpha, i)S_i'R(\alpha, i)$ for all transactions i. This construction ensures that all operations in S_i must be scheduled consecutively in any legal schedule: any schedule that interleaves operations from different S_i's must violate the reads-from mapping for α. It follows that we have a positive VSC-read instance if and only if H is view serializable.

To complete the theorem, we note that Observation 2.3 can be adapted to the VSC-read problem by simply adding the read-mapping. □

VL-read. We now turn to the VL-read problem and show that, in contrast to the VSC-read problem, there is an $O(n \log n)$-time algorithm for this problem.

THEOREM 4.2. *There is an $O(n \log n)$-time algorithm for the VL-read problem.*

Proof. We sort the input by address, and within an address, by start-of-interval and end-of-interval times. We check to see that each read is mapped to a write operation with the same address and value; otherwise, we have a negative instance. Consider each address separately. Define a *cluster* to be a write w and the set R of reads mapped to the write. The write w must be assigned a time earlier than that of any read in R. Thus the start-of-interval time for w must be earlier than the end-of-interval time for any read in R; otherwise, we have a negative instance. Any legal time assignment defines an interval for a cluster from the time assigned to w to the time assigned to the last read in R; only operations in this cluster can be scheduled during this time interval. Define a *zone* for a cluster to be the interval from the earliest end-of-interval time for an operation in the cluster to the latest start-of-interval time

for an operation in the cluster. In the normal scenario, the former is earlier than the latter, and we have a *forward* zone; otherwise, we have a *backward* zone. For any legal time assignment, the interval of time for a cluster with a forward zone must contain that zone. Therefore, if two forward zones overlap, we have a negative instance.

For a cluster with a backward zone, the backward zone is the intersection of the individual operation intervals for operations in the cluster. Thus all operations in a cluster can be safely scheduled in *any* subinterval of the zone of positive length. Moreover, one can see that the interval of time for the cluster in any legal time assignment must intersect its backward zone. It follows that if a backward zone is contained within the forward zone for some other cluster, we have a negative instance.

Finally, if none of the illegal configurations described above occurs, we have a positive instance. Let ϵ_0 be the minimum difference between any pair of times in the instance. Augment each forward zone by $\epsilon_0/3$ before and after the zone. Then the operations in forward zone clusters are safely scheduled within their respective augmented zones, and the operations in backward zone clusters are safely scheduled outside all augmented forward zones using nonoverlapping subintervals: for each cluster, w is scheduled at the start of the augmented zone or subinterval, and then the operations in R are scheduled in order of start-of-interval times. The reader may verify that this is a legal schedule.

The final legal schedule is obtained by merging the individual schedules for each address. After the initial sorting, the algorithm runs in linear time. □

4.2. Providing the write-order. A second source of ambiguity in the basic VSC and VL problems is that there may be multiple writes to the same location by different processors. The question addressed in this section is as follows: If for each location there is no ambiguity as to the order of writes to the location, do the VSC and VL problems remain NP-complete?

We define the *VSC-write* and *VL-write* problems, in which for each shared-memory location, a total order on the write operations to the location is known; a legal schedule must respect this *write-order* relation.

VSC-write. We show that the VSC-write problem is NP-complete. In fact, we prove that the VSC problem is NP-complete even when each location is written to by only a single processor, yielding the following stronger result.

THEOREM 4.3. *The VSC and VSC-write problems restricted to instances in which each location is written to by only a single processor are* NP*-complete.*

Proof. Our reduction from 3SAT is depicted in Figure 9. The construction consists of operations used to select a truth assignment and then to test each clause. We use the $W(ok)$ operation in the first sequence to signal when all clauses have been tested and it is time to clean up so that the set of operations not used in testing the particular assignment may be safely scheduled if and only if the assignment satisfied the 3SAT instance. Note that for each location, all writes to that location appear in the same sequence. Thus for the VSC-write problem, the write-order is implied by the order of the individual sequences.

In all three NP-completeness proofs of section 2, the constructions have, for each variable, two writes to the same location such that the order between the two writes determines the truth setting for the variable. Here the order on two such writes is predetermined by the order in which they appear in their processor sequence. Hence we use two writes to *different* locations, which must somehow be coupled so that only one setting of the 3SAT variable occurs prior to the cleanup. The first interesting part of the construction, then, is the four sequences for each variable.

First, we have the following three sequences:

$W(a_1, 1)$	$W(b_1, 1)$	$W(c_1, 1)$
$W(a_1, 2)$	$W(b_1, 2)$	$W(c_1, 2)$
$W(a_2, 1)$	$W(b_2, 1)$	$W(c_2, 1)$
$W(a_2, 2)$	$W(b_2, 2)$	$W(c_2, 2)$
:	:	:
$W(a_m, 1)$	$W(b_m, 1)$	$W(c_m, 1)$
$W(a_m, 2)$	$W(b_m, 2)$	$W(c_m, 2)$
$R(f_1)$	$W(f_1)$	$W(f_2)$
$R(f_2)$		
$W(ok)$	$R(ok)$	$R(ok)$
$W(a_1, 1)$	$W(b_1, 1)$	$W(c_1, 1)$
$W(a_2, 1)$	$W(b_2, 1)$	$W(c_2, 1)$
:	:	:
$W(a_m, 1)$	$W(b_m, 1)$	$W(c_m, 1)$

Then for each variable v_i, $i = 1, 2, \ldots, n$, we have the following four sequences:

$W(y_i, 1)$	$W(z_i, 1)$	$R(y_i, 2)$	$R(z_i, 2)$
$W(y_i, 2)$	$W(z_i, 2)$	$R(z_i, 1)$	$R(y_i, 1)$
$W(y_i, 1)$	$W(z_i, 1)$	$W(v_i)$	$W(\overline{v}_i)$
		$R(ok)$	$R(ok)$
		$R(y_i, 2)$	$R(z_i, 2)$

Finally, for each clause $C_j = \lambda_{j,1} \vee \lambda_{j,2} \vee \lambda_{j,3}$, $j = 1, 2, \ldots, m$, where $\lambda_{j,1}$, $\lambda_{j,2}$, and $\lambda_{j,3}$ denote literals from $\{v_1, \ldots, v_n, \overline{v}_1, \ldots, \overline{v}_n\}$, we have the following three sequences:

$R(\lambda_{j,1})$	$R(\lambda_{j,2})$	$R(\lambda_{j,3})$
$R(b_j, 1)$	$R(c_j, 1)$	$R(a_j, 1)$
$R(a_j, 2)$	$R(b_j, 2)$	$R(c_j, 2)$

FIG. 9. *Transforming an instance of* 3SAT *with n variables and m clauses to an instance of VSC with $4n + 3m + 3$ sequences and locations, such that each location is written to by only a single processor. For locations that are written to only once in the entire construction, we omit the data field in the write and the reads for that location.*

We begin by proving the following lemma about these four sequences.

LEMMA 4.4. *Consider the four sequences for a variable v_i, together with an additional sequence comprised solely of a $W(ok)$ operation. Then we have the following:*

1. *There exists a legal schedule for these sequences such that $W(v_i)$ precedes $W(ok)$ precedes $W(\overline{v}_i)$.*
2. *There exists a legal schedule for these sequences such that $W(\overline{v}_i)$ precedes $W(ok)$ precedes $W(v_i)$.*
3. *There exists no legal schedule for these sequences such that both $W(v_i)$ and $W(\overline{v}_i)$ precede $W(ok)$.*

Proof. For claim 1, the reader may verify that the following is a legal schedule: $W(y_i, 1)$, $W(y_i, 2)$, $R(y_i, 2)$, $W(z_i, 1)$, $R(z_i, 1)$, $W(v_i)$, $W(ok)$, $R(ok)$, $R(y_i, 2)$, $W(z_i, 2)$, $R(z_i, 2)$, $W(y_i, 1)$, $R(y_i, 1)$, $W(\overline{v}_i)$, $R(ok)$, $R(z_i, 2)$, $W(z_i, 1)$. Claim 2 follows by symmetry. As for claim 3, suppose both did precede $W(ok)$ in a legal schedule S.

Let $S = S_1S_2$, where S_1 is the prefix of S up to and including the $W(ok)$ operation. Then the first $R(y_i, 2)$ and the first $R(z_i, 2)$ are in S_1. By symmetry, assume without loss of generality that the former precedes the latter in S_1. Then it can be seen that the first $W(y_i, 1)$ precedes $W(y_i, 2)$ which precedes $R(y_i, 1)$ in S_1. So to avoid a reads-from violation, $W(y_i, 2)$ must precede the second $W(y_i, 1)$ which must precede $R(y_i, 1)$ in S_1. But since the second $R(y_i, 2)$ is in S_2 and there are no other $W(y_i, 2)$ operations to schedule after the $R(y_i, 1)$ operation, S is not a legal schedule, and we have a contradiction. □

Likewise, in the previous construction, we simulate an OR using three writes with the same address and value, any one of which signals that a clause has been satisfied by the truth assignment. Here the order on any such writes is predetermined, and hence this approach for simulating an OR does not work. The second interesting part of the construction, then, is how the first three sequences together with all the clause sequences accurately test each clause.

Consider the jth clause C_j and its three sequences, say S_1, S_2, and S_3. There is a row in the first three sequences in which writes to a_j, b_j, and c_j set the values of these locations to 1, followed by a row in which writes to these same locations set the values to 2. These six operations must be scheduled prior to the cleanup. If C_j is satisfied, then for at least one of S_1, S_2, or S_3, the entire sequence can be scheduled prior to the cleanup. On the other hand, if C_j is not satisfied, then all operations in S_1, S_2, and S_3 remain to be scheduled during the cleanup. The second reads in S_1, S_2, and S_3 read the value 1; thus before the first such read, e.g., $R(c_j, 1)$, can be scheduled, we must schedule the appropriate write of 1, $W(c_j, 1)$, during the cleanup. But then for that particular location c_j, the read of 2 remains to be scheduled, and yet the value cannot be reset to 2.

LEMMA 4.5. *Let $\mathcal{F}$ be an instance of the* 3SAT *problem, and let $\mathcal{V}$ be the instance of the VSC-write problem constructed as depicted in Figure* 9. *Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable, and let $\mathcal{T}$ be a satisfying assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$:

1. For each variable v_i set to true (respectively, false) by $\mathcal{T}$, schedule operations in the four sequences for v_i according to claim 1 (respectively, claim 2) of Lemma 4.4, up to but not including the $W(ok)$ added by the lemma.
2. For each clause C_j in turn, schedule as follows: First, schedule one of the reads that agrees with $\mathcal{T}$. This leaves two reads in the same sequence, say $R(c_j, 1)$ and $R(b_j, 2)$, if the second of the sequences for C_j agrees with $\mathcal{T}$. Schedule the first $W(a_j, 1)$, the first $W(b_j, 1)$, and the first $W(c_j, 1)$ (in a row in the figure), then the read $R(c_j, 1)$. Then schedule $W(a_j, 2)$, $W(b_j, 2)$, $W(c_j, 2)$, and $R(b_j, 2)$.
3. Schedule $W(f_1)$, $R(f_1)$, $W(f_2)$, $R(f_2)$, and $W(ok)$. Then schedule the two $R(ok)$ operations (from the same sequences as the $W(f_1)$ and $W(f_2)$).
4. For each variable v_i set to true (respectively, false) by $\mathcal{T}$, schedule all remaining operations in the four sequences for v_i according to claim 1 (respectively, claim 2) of Lemma 4.4 after the $W(ok)$ added by the lemma.
5. At this point, both $W(v_i)$ and $W(\overline{v}_i)$ have been scheduled, so all remaining $R(v_i)$ and $R(\overline{v}_i)$ operations can safely be scheduled next.
6. For each clause C_j in turn, schedule as follows: There are two pairs of unscheduled reads in the sequences for C_j, say $R(b_j, 1)$ followed by $R(a_j, 2)$ and $R(a_j, 1)$ followed by $R(c_j, 2)$ (if the pair $R(c_j, 1)$ and $R(b_j, 2)$ were the ones

scheduled above). Note that the last writes to a_j, b_j, and c_j scheduled each wrote the value 2. Schedule the second $W(b_j, 1)$, then $R(b_j, 1)$, then $R(a_j, 2)$. Then schedule the second $W(a_j, 1)$, then $R(a_j, 1)$, then $R(c_j, 2)$, and finally the second $W(c_j, 1)$.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance, and let S be a legal schedule for $\mathcal{V}$. Let $S = S_1 S_2$, where S_1 is the prefix of S up to and including $W(ok)$ and S_2 is the remaining suffix of S. We observe that the claims of Lemma 4.4 apply to S since, outside of $W(v_i)$, $W(\overline{v}_i)$, and $R(ok)$, the four sequences for each v_i contain addresses appearing only in these four sequences. Let $\mathcal{T}$ be the partial truth assignment such that for each v_i, v_i is set to true if $W(v_i)$ is in S_1, v_i is set to false if $W(\overline{v}_i)$ is in S_1, and v_i is not set otherwise. (It follows from claim 3 of Lemma 4.4 that no variable is set to both true and false.) We claim that the partial truth assignment $\mathcal{T}$ satisfies $\mathcal{F}$. Suppose not, and let C_j be an unsatisfied clause. Since for each of the three sequences for C_j, the first read cannot be in S_1, then the last two reads in each such sequence are in S_2. Moreover, the last write to a_j, b_j, and c_j in S_1 wrote the value 2. By symmetry, assume without loss of generality that $R(a_j, 1)$ precedes both $R(b_j, 1)$ and $R(c_j, 1)$ in S_2. Then the second $W(a_j, 1)$ precedes $R(a_j, 1)$, to avoid a reads-from violation, which precedes $R(b_j, 1)$ which precedes $R(a_j, 2)$ in S_2. Since there is no $W(a_j, 2)$ in between the $W(a_j, 1)$ and the $R(a_j, 2)$ in S_2, S is not a legal schedule, a contradiction. Therefore, all clauses are satisfied by T. □

Since the above transformation can be done in polynomial time, Theorem 4.3 follows. □

VL-write. We now turn to the VL-write problem, and show that, in contrast to the VSC-write problem, there is an $O(n \log n)$-time algorithm for this problem.

THEOREM 4.6. *There is an $O(n \log n)$-time algorithm for the VL-write problem.*

Proof. Let $\mathcal{V}$ be an instance of the VL-write problem. We sort the instance by address and, within each address, by start-of-interval and end-of-interval times. We check to see that each write is included only in the write-order for its address; otherwise, we have a negative instance.

Consider the set of operations S on an address a in $\mathcal{V}$. Let $w_1, w_2, \ldots, w_m$ be the sequence of writes to address a as ordered by the write-order. The algorithm proceeds in rounds. We begin with all operations in S unscheduled and maintain the invariant that all the start-of-interval times for scheduled operations are less than all the end-of-interval times of unscheduled operations. At round i, if the start-of-interval time for w_i is greater than the end-of-interval time for an unscheduled operation, then we have a negative instance and halt. Otherwise, schedule w_i. Then greedily schedule, in order of increasing start-of-interval times, all unscheduled reads of the same value whose start-of-interval times are less than the end-of-interval time for all unscheduled operations. Continue on to the next round.

If any reads remain unscheduled after round m, then we have a negative instance and halt. Otherwise, a legal assignment of times for the operations in S is obtained inductively as follows. Let ϵ_0 be the minimum difference between any pair of times (start-of-interval or end-of-interval) in $\mathcal{V}$; let $\epsilon = \epsilon_0/n$. The first operation scheduled, w_1, is assigned a time equal to its start-of-interval time. Each subsequent operation is assigned a time equal to the maximum of its start-of-interval time and ϵ greater than the time assigned to the previous scheduled operation.

We now show that this greedy algorithm finds a legal schedule of S if and only if one exists. The schedule produced by the algorithm is legal since it contains all of

the operations in S, the write-order is respected, the last write scheduled prior to a read has the same value, and all operations are assigned times within their intervals. To see that this last condition holds, consider an operation π that is assigned a time greater than its start-of-interval time. Let π' be the last operation prior to π that is assigned a time equal to its respective start-of-interval time t'. Since π' is scheduled before π, t' is less than the end-of-interval time for π. Thus π is assigned a time less than $t' + \epsilon_0$ and hence within its interval.

Conversely, assume that there is a legal schedule of S. Suppose that the greedy algorithm completes $j \leq m$ rounds. For $i = 1, \ldots, j$, we claim that the set of operations S_i in the greedy schedule prior to w_i is a superset of the set of operations scheduled prior to w_i in any legal schedule of S. The proof is by induction, with a trivial basis since w_1 is the first operation in any legal schedule. Assume that the claim is true for $i-1$. Consider a legal schedule of S, and let S'_i be the set of operations prior to w_i in the schedule. Suppose there is an operation α in S'_i that is not in S_i. Since the order on writes is fixed, S'_i and S_i contain the same writes, as do S'_{i-1} and S_{i-1}. It follows from the inductive assumption that α is a read of the value written by w_{i-1}. Since α is not in S_i, there must be an operation not in S_i whose end-of-interval time is less than the start-of-interval time for α such that the operation is a read of a different value. Since α is in S'_i but reads a different value, it must be in S'_{i-1}. But then by the inductive assumption, it is in S_{i-1}, a contradiction.

From this claim, we see that any unscheduled operation in S_j will be scheduled after w_j in any legal schedule, and hence its end-of-interval time is greater than the start-of-interval time for w_j. Moreover, the set of reads not in S_m is a subset of the set of reads scheduled after w_m in any legal schedule. It follows that the algorithm successfully completes all m rounds and finds a legal schedule of S.

If for all addresses, the algorithm succeeds in finding a legal schedule, we have a positive instance. The final legal schedule for $\mathcal{V}$ is obtained by merging the individual schedules for each address. □

4.3. Read&write only. In the VSC-write and VL-write problems, the input provides a mapping from each write to the previous write on the same address (if any). In this section, we consider the complexity of the VSC and VL problems when for each write, the input provides not the identity of the previous write but only its value.

We view the memory operations as atomic read–modify–write operations. Accordingly, we define a *read&write*$(a, d_{old} : d_{new}, t_1, t_2)$ operation, where $a \in A$ is the address, $d_{old} \in D$ is the old value (returned by the read), and $d_{new} \in D$ is the new value written. A legal schedule must respect this pairing of old and new values: each *read&write*$(a, d_{old} : d_{new}, t_1, t_2)$ operation must be preceded by an operation that writes d_{old} to a with no intervening operation that writes a different value to a. As before, the start-of-interval time t_1 and end-of-interval time t_2 are needed for the VL problem but not the VSC problem. In the VSC and VL problems with *read&write only*, all operations are *read&write* operations; in this context, a *read* is simply a *read&write* that does not alter the value.

Note that the relationship between the VSC-write/VL-write problems and the VSC/VL problems with read&write only is analogous to the relationship between the VSC-read/VL-read problems and the basic VSC/VL problems, namely, the distinction between providing the identity of the previous write versus providing only the value of the previous write.

VSC with read&write only. We show that the VSC problem with read&write only is NP-complete, even under two restrictive scenarios.

By Theorem 4.3, the VSC problem restricted to instances in which each location is written to by only a single processor is NP-complete. In such instances, each write can be replaced with the appropriate read&write operation, yielding the following corollary.

COROLLARY 4.7. *The VSC problem with read&write only restricted to instances in which each location is written to by only a single processor is* NP*-complete.*

We next observe that instances with long processor sequences can always be transformed to equivalent instances with at most two memory operations per processor.

OBSERVATION 4.8. *There is a linear-time reduction from the VSC problem with read&write only with p sequences, n operations, and k variables to the VSC problem with read&write only with at most n sequences, $O(n)$ operations, and k variables such that each sequence contains at most two operations.*

Proof. For $i = 1, 2, \ldots, p$, consider the ith sequence in the instance, $S_i = rw(a_1, d_1 : d'_1)$, $rw(a_2, d_2 : d'_2)$, ..., $rw(a_{m_i}, d_{m_i} : d'_{m_i})$, where $a_1, \ldots, a_{m_i}$, $d_1, \ldots, d_{m_i}$, $d'_1, \ldots, d'_{m_i}$ are not necessarily distinct. When S_i has more than two memory operations (i.e., $m_i > 2$), the construction splits each operation $rw(a_j, d_j : d'_j)$ in S_i, other than the first and the last operation, into a pair of operations to the same address: $rw(a_j, d_j : x^{(i)}_{j-1})$ and $rw(a_j, x^{(i)}_{j-1} : d'_j)$. In particular, we replace S_i with the following $m_i - 1$ replacement sequences for S_i:

$$\begin{array}{cccccc} rw(a_1, d_1 : d'_1) & rw(a_2, x^{(i)}_1 : d'_2) & rw(a_3, x^{(i)}_2 : d'_3) & \cdots & rw(a_{m_i-1}, x^{(i)}_{m_i-2} : d'_{m_i-1}) \\ rw(a_2, d_2 : x^{(i)}_1) & rw(a_3, d_3 : x^{(i)}_2) & rw(a_4, d_4 : x^{(i)}_3) & & rw(a_{m_i}, d_{m_i} : d'_{m_i}) \end{array},$$

where $\forall i, j, i', j'$, $x^{(i)}_j = x^{(i')}_{j'}$ if and only if $i = i'$ and $j = j'$.

The idea behind this construction is as follows. Consider the pairs of operations in the constructed instance. Since each new data value is unique, then in any legal schedule of the constructed instance, no operation to the same address can be scheduled between a pair, and no operation from the same replacement sequences can be scheduled between a pair. This in turn will imply that any legal schedule can be reordered to obtain a new legal schedule in which the two operations in any pair are consecutive. Then considering each pair as being replaced by its original operation, we have a legal schedule of the original instance.

The reader may verify that this constructed instance is a positive instance if and only if the original instance is a positive instance. □

We now show that the VSC problem with read&write only is NP-complete even when there is only a single variable and at most two operations per sequence.

THEOREM 4.9. *The VSC problem with read&write only restricted to instances with one variable and at most two memory operations per sequence is* NP*-complete.*

Proof. We show the reduction for sequences with many operations; this can be transformed to sequences with at most two operations each by applying Observation 4.8. Given a 3SAT instance $\mathcal{F}$ with n variables and m clauses, we construct the following VSC instance $\mathcal{V}$. First, we have the following $2n + 1$ sequences, A, $V_1, V'_1, \ldots, V_n, V'_n$, where $\perp$ is the initial value of a, the literal v_1 is in clauses $\{c_2, c_3, \ldots, c_m\}$, the literal $\overline{v}_1$ is in clauses $\{c_4, c_6, \ldots, c_9\}$,..., the literal v_n is in

clauses $\{c_4, c_5, \ldots, c_8\}$, and the literal $\overline{v}_n$ is in clauses $\{c_1, c_6, \ldots, c_m\}$:

$\underline{A}$	$\underline{V_1}$	$\underline{V_1'}$	$\cdots$	$\underline{V_n}$	$\underline{V_n'}$
$rw(a, \perp : 1)$	$rw(a, 1 : c_2)$	$rw(a, 1 : c_4)$		$rw(a, n : c_4)$	$rw(a, n : c_1)$
$rw(a, 1' : 2)$	$rw(a, c_2 : c_3)$	$rw(a, c_4 : c_6)$		$rw(a, c_4 : c_5)$	$rw(a, c_1 : c_6)$
$rw(a, 2' : 3)$	:	:		:	:
:	$rw(a, c_m : 1')$	$rw(a, c_9 : 1')$		$rw(a, c_8 : n')$	$rw(a, c_m : n')$
$rw(a, (n-1)' : n)$					

In addition, we have the following $m + 1$ sequences, A', $C_1, C_2, \ldots, C_m$:

$\underline{A'}$	$\underline{C_1}$	$\underline{C_2}$	$\cdots$	$\underline{C_m}$
$rw(a, c_m' : 1)$	$rw(a, c_1 : c_1)$	$rw(a, c_2 : c_2)$		$rw(a, c_m : c_m)$
$rw(a, 1' : 2)$	$rw(a, n' : c_1')$	$rw(a, c_1' : c_2')$		$rw(a, c_{m-1}' : c_m')$
$rw(a, 2' : 3)$				
:				
$rw(a, (n-1)' : n)$				

LEMMA 4.10. *Let $\mathcal{F}$ be an instance of the* 3SAT *problem, and let $\mathcal{V}$ be the instance of the VSC problem with read&write only constructed as defined above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. Let $T(v_1), \ldots, T(v_n)$ be a satisfying assignment for $\mathcal{F}$, where $T(v_i) \in \{\mathsf{T}, \mathsf{F}\}$. We construct the following schedule for $\mathcal{V}$:

1. first, $rw(a, \perp : 1)$ (or $rw(\perp : 1)$; from now on we shall omit the variable "a"); this is the first operation in A;
2. then if $T(v_1) = \mathsf{T}$, all of the operations in V_1, interleaved with *clause read* operations, as explained below, and followed by the second operation in A, $rw(1' : 2)$; otherwise, if $T(v_1) = \mathsf{F}$, all of the operations in V_1', interleaved with *clause read* operations, as explained below, and followed by the second operation in A.
3. For $i = 2, 3, \ldots, n$, repeat the previous step: If $T(v_i) = \mathsf{T}$ ($T(v_i) = \mathsf{F}$), schedule all of the operations in V_i (respectively, V_i'), interleaved with *clause read* operations, as explained below, and followed by, for $i < n$, the $(i+1)$th operation in A, $rw(i' : i + 1)$.
4. The first operation in each sequence C_j is denoted the *clause read* for clause c_j. Since T satisfies $\mathcal{F}$, then each clause c_j is satisfied by some $T(v_i)$, and therefore when the corresponding V_i or V_i' was scheduled, an operation of the form $rw(x : c_j)$ for some value x was in that sequence; the clause read for c_j is scheduled immediately after the first such operation $rw(x : c_j)$.
5. Next, we schedule $rw(n' : c_1')$, $rw(c_1' : c_2'), \ldots, rw(c_{m-1}' : c_m')$, followed by $rw(c_{m'} : 1)$ (the first operation in A').
6. Finally, repeat the following for $i = 1, 2, \ldots, n$: If $T(v_i) = \mathsf{F}$ ($T(v_i) = \mathsf{T}$), schedule all of the operations in V_i (respectively, V_i'), followed by, for $i < n$, the $(i+1)$th operation in A': $rw(i' : i + 1)$.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive instance and let S be a legal schedule for $\mathcal{V}$. For $i = 1, \ldots, n$, let the first operation in S from either V_i or V_i' be denoted the *variable read* operation for v_i and the sequence V_i or V_i' containing v_i be denoted the *variable read sequence*. Let the sequence V_i or V_i' that is not the variable read sequence for v_i be denoted the *cleanup read sequence* and its first operation be denoted the *cleanup read*. Consider the truth assignment T defined as follows: For each $i = 1, \ldots, n$,

$T(v_i) = \mathsf{T}$ if V_i is the variable read sequence for v_i and $T(v_i) = \mathsf{F}$ otherwise. Note that by the construction, operations in a variable read sequence only write values c_j for clauses satisfied by T. We will show that T is a satisfying assignment for $\mathcal{F}$.

Suppose there is a clause c_j not satisfied by T, and consider the clause read for c_j: $rw(c_j : c_j)$. Let $S = \sigma_1 rw(c_j : c_j)\sigma_2$. We claim that $rw(c'_m : 1)$ and hence all of A' is in σ_2. To see this, first observe that since the values $c'_1, c'_2, \ldots, c'_m$ are read by exactly one operation and written by exactly one operation, the sequence $rw(n' : c'_1)$, $rw(c'_1 : c'_2), \ldots, rw(c'_{m-1} : c'_m)$, $rw(c'_m : 1)$ is a consecutive subsequence of S. Since the second operation in C_j is in this subsequence and in σ_2, then $rw(c'_m : 1)$ is in σ_2 as well. Thus all of A' is in σ_2, and hence at most one operation that writes i, $i = 1, \ldots, n$, is in σ_1.

Since among the two operations that read i, the variable read precedes the cleanup read, it follows that the cleanup read and hence the cleanup read sequence is in σ_2 for all v_i. The last operation in σ_1 must write c_j and thus must be in a variable read sequence. Thus, as observed above, c_j is satisfied by T, a contradiction.

The lemma follows. □

Since the above transformation can be done in polynomial time, Theorem 4.9 follows. □

VL with read&write only. We show that the VL problem with read&write only is NP-complete. This contrasts with the $O(n \log n)$-time algorithm for the VL-write problem.

THEOREM 4.11. *The VL problem with read&write only is* NP*-complete.*

Proof. With only read&write operations, a more careful construction is needed than the one shown in Figure 8 that relies on writes whose old values are not predetermined. Each new value to be written must serve as the old value for the next write (recall that there is but a single location), giving us less flexibility in the construction. Nevertheless, we show below how to overcome this difficulty to obtain a construction with the desired properties.

Our reduction is again from SAT. Consider an instance $\mathcal{F}$ of SAT with n variables, $v_1, v_2, \ldots, v_n$, and m clauses, $C_1, C_2, \ldots, C_m$. Without loss of generality, assume that each variable and its negation appear in at least one clause, but not the same clause, and that there are no repeated variables in a clause. We construct an instance of the VL problem with at most $5nm + 10n + m + 1$ operations, corresponding to $\mathcal{F}$. To simplify the description, we have multiple intervals with common start times or end times; these ties can be broken arbitrarily to ensure unique time values.

Figure 10 depicts an example construction. First, there is an initial read&write operation: *read&write*$(a, \perp : 0, 1, 2)$. Then for each clause C_j, $j = 1, \ldots, m$, we have a *read&write*$(a, c_j : c_j, 3, 5n + 4)$ operation, denoted the *clause read* for C_j.

For $i = 1, \ldots, n$, let m_i (respectively, $m_{\bar{i}}$) be the number of clauses containing the literal v_i (respectively, $\overline{v_i}$). By assumption, $m_i > 0$, $m_{\bar{i}} > 0$, and $m_i + m_{\bar{i}} \leq m$. For $i = 1, \ldots, n+1$, let $\Delta_i = (7m+4)(i-1)+5n+4$. For $i = 1, \ldots, n$, let $\Delta_{\bar{i}} = \Delta_i + 7m_i + 2$.

For each variable v_i, assignment to v_i is simulated using six operations:

- *read&write*$(a, 0 : i, 5i, 5i + 1)$,
- *read&write*$(a, 0 : \bar{i}, 5i, 5i + 1)$,
- *read&write*$(a, \hat{i} : 0, 5i, 5i + 1)$,
- *read&write*$(a, \hat{i} : 0, 5i + 3, 5i + 4)$,
- *read&write*$(a, i : \hat{i}, 5i, \Delta_i - 1)$,
- *read&write*$(a, \bar{i} : \hat{i}, 5i, \Delta_{\bar{i}} - 1)$.

There is a set of operations for each literal v_i, denoted the *group* of operations

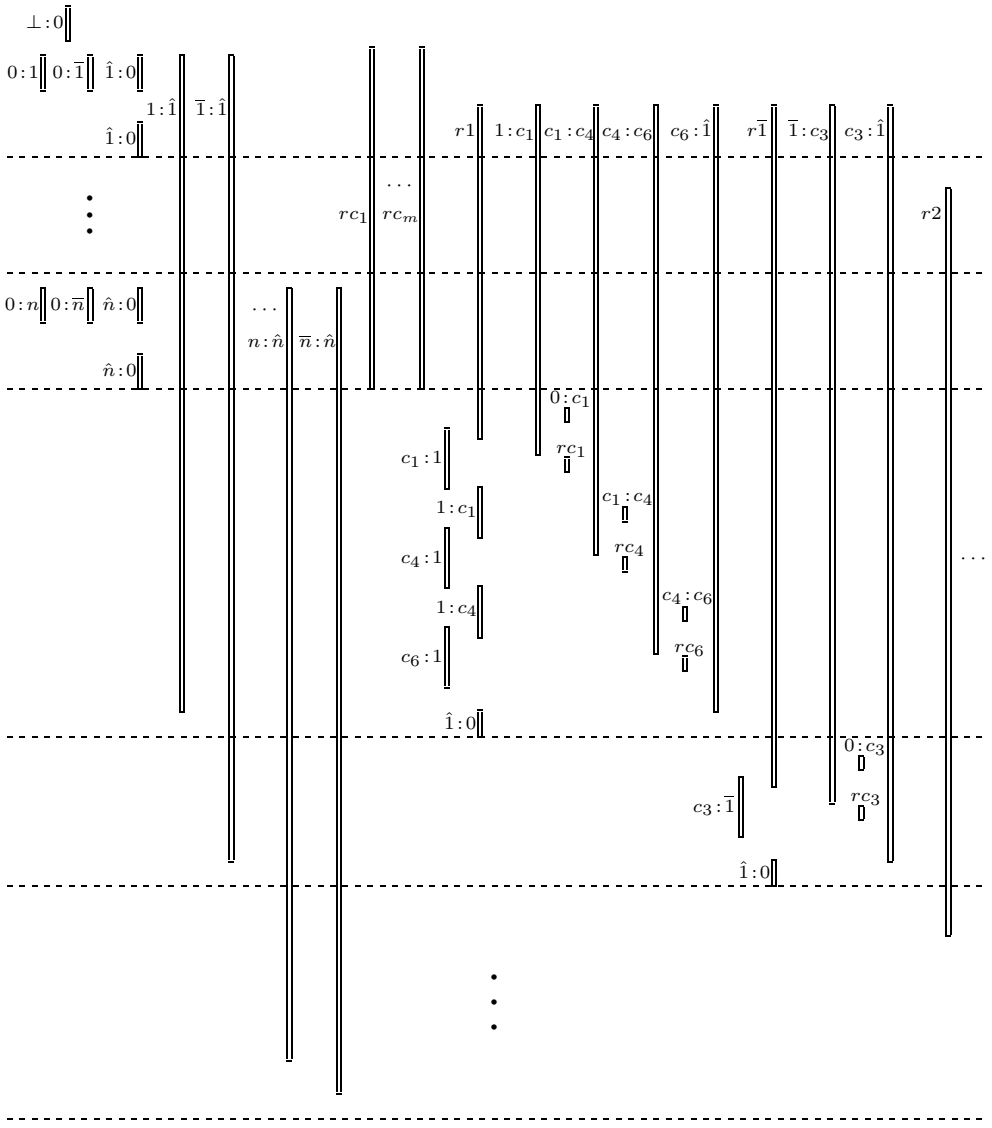


FIG. 10. *Transforming an instance of* SAT *to an instance of VL with a single location such that all memory operations are read&write operations. There are n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. The full construction has at most $5nm + 10n + m + 1$ operations. Here the literal v_1 appears in exactly clauses C_1, C_4, and C_6, the literal $\overline{v}_1$ appears in clause C_3 only, and so forth. Every column corresponds to a processor. Vertical boxes depict the intervals of time for the respective read&write operations to the single location; time progresses from top to bottom in the figure. A read&write operation with old value d and new value d' is denoted $d:d'$; the case where $d = d'$ is denoted simply rd. The set of values used is $\{0, 1, \overline{1}, \hat{1}, 2, \overline{2}, \hat{2}, \ldots, n, \overline{n}, \hat{n}, c_1, c_2, \ldots, c_m\}$. The first read&write necessarily reads the initial value in memory, which we denote $\perp$, before writing a new value (i.e., 0).*

for i. For $i = 1, \ldots, n$, if $C_{i_1}, C_{i_2}, \ldots, C_{i_{m_i}}$ are the clauses containing the literal v_i, we have the following $5m_i + 2$ intervals. First, for C_{i_1}, we have

- *read&write*$(a, i\!:\!i, 5i+2, \Delta_i+4)$,
- *read&write*$(a, i\!:\!c_{i_1}, 5i+2, \Delta_i+5)$,
- *read&write*$(a, 0\!:\!c_{i_1}, \Delta_i+1, \Delta_i+2)$,
- *read&write*$(a, c_{i_1}\!:\!c_{i_1}, \Delta_i+5, \Delta_i+6)$,
- *read&write*$(a, c_{i_1}\!:\!i, \Delta_i+3, \Delta_i+7)$.

Then for C_{i_k}, $k = 2, \ldots, m_i$, we have

- *read&write*$(a, i\!:\!c_{i_{k-1}}, \Delta_i+7(k-1), \Delta_i+7(k-1)+4)$,
- *read&write*$(a, c_{i_{k-1}}\!:\!c_{i_k}, 5i+2, \Delta_i+7(k-1)+5)$,
- *read&write*$(a, c_{i_{k-1}}\!:\!c_{i_k}, \Delta_i+7(k-1)+1, \Delta_i+7(k-1)+2)$,
- *read&write*$(a, c_{i_k}\!:\!c_{i_k}, \Delta_i+7(k-1)+5, \Delta_i+7(k-1)+6)$,
- *read&write*$(a, c_{i_k}\!:\!i, \Delta_i+7(k-1)+3, \Delta_i+7k)$.

Finally, we have *read&write*$(a, c_{i_{m_i}}\!:\!\hat{\imath}, 5i+2, \Delta_{\bar{\imath}}-1)$ and *read&write*$(a, \hat{\imath}\!:\!0, \Delta_{\bar{\imath}}-1, \Delta_{\bar{\imath}})$.

Likewise, there is a set of operations for each literal $\overline{v_i}$, denoted the *group* of operations for $\bar{\imath}$. For $i = 1, \ldots, n$, if $C_{\bar{i}_1}, C_{\bar{i}_2}, \ldots, C_{\bar{i}_{m_{\bar{i}}}}$ are the clauses containing the literal $\overline{v_i}$, we have the $5m_{\bar{\imath}} + 2$ intervals obtained from the previous definition by replacing $\Delta_{\bar{\imath}}$ with Δ_{i+1} and leaving "$5i+2$" unchanged but otherwise replacing i with $\bar{\imath}$ throughout.

This completes the construction.

We show below that for any satisfying truth assignment, there is a legal schedule for the instance constructed. As an example, consider a satisfying assignment that sets v_1 to true, and refer to Figure 10. In this case, a legal schedule begins: $\perp : 0$, then $0 : \overline{1}$, then $\overline{1} : \hat{1}$, then the first $\hat{1} : 0$, then $0 : 1$, then $r1$, then $1 : c_1$ (immediately to the right of $r1$ in the figure), then the clause read rc_1, then $c_1 : c_4$, then the clause read rc_4 (not shown), then $c_4 : c_6$, then the clause read rc_6 (not shown), then $c_6 : \hat{1}$, and then the second $\hat{1} : 0$. The schedule continues with operations for $v_2, v_3, \ldots, v_n$ until the second $\hat{n} : 0$ is scheduled. Then the remaining (cleanup) operations for v_1 are scheduled as follows: $0 : c_1$ (in the figure, just below the third dashed line from the top), then rc_1, then $c_1 : 1$, then the $1 : c_1$ to its right, then the $c_1 : c_4$ to its right, then the rc_4 below it, then $c_4 : 1$, then $1 : c_4$, then the $c_4 : c_6$ to its right, then the rc_6 below it, then $c_6 : 1$, then $1 : \hat{1}$, and then the $\hat{1} : 0$ just above the fourth dashed line; then $0 : c_3$, then $c_3 : \overline{1}$, then $r\overline{1}$, then $\overline{1} : c_3$, then rc_3, then $c_3 : \hat{1}$, and then the $\hat{1} : 0$ just above the fifth dashed line. The schedule continues with cleanup operations for $v_2, v_3, \ldots, v_n$ until all operations have been scheduled.

LEMMA 4.12. *Let $\mathcal{F}$ be an instance of a* SAT *problem, and let $\mathcal{V}$ be the instance of the VL problem with read&write operations constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable, and let $\mathcal{T}$ be a satisfying truth assignment for $\mathcal{F}$. We construct the following schedule for $\mathcal{V}$. When there is no ambiguity, we use the notation "$d_1 : d_2$" to denote a read&write operation to address a with old value d_1 and new value d_2; the case where $d_1 = d_2$ is denoted simply "rd_1." We show the order in which events are scheduled; an assignment of distinct, valid times to operations is left to the reader. The schedule is as follows:

1. First, schedule $\perp : 0$.
2. Repeat the following for $i = 1, 2, \ldots, n$:
   If v_i is set to true by $\mathcal{T}$, schedule $0 : \bar{\imath}$, then $\bar{\imath} : \hat{\imath}$, then *read&write*$(a, \hat{\imath} : 0, 5i, 5i+1)$, then $0 : i$, and then ri. Schedule $i : c_{i_1}$, followed by—if it has not already been scheduled—the clause read rc_j, where $j = i_1$. Then repeat

for $k = 2, \ldots, m_i$: schedule *read&write*$(a, c_{i_{k-1}} : c_{i_k}, 5i + 2, \Delta_i + 7(k-1) + 5)$, followed by—if it has not already been scheduled—the clause read rc_j, where $j = i_k$. Finally, schedule $c_{i_{m_i}} : \hat{\imath}$, then *read&write*$(a, \hat{\imath} : 0, 5i + 3, 5i + 4)$.

The case where v_i is set to false by $\mathcal{T}$ is symmetric and left to the reader.

3. Since $\mathcal{T}$ is a satisfying assignment for $\mathcal{F}$, all clause reads have been scheduled by this point. Note also that all the above operations can be scheduled prior to time Δ_1.

   Repeat the following for $i = 1, 2, \ldots, n$:

   If v_i is set to true by $\mathcal{T}$, consider the $4m_i$ unscheduled operations in group i, together with the unscheduled $i : \hat{\imath}$, and finally the $5m_{\bar{\imath}} + 2$ (unscheduled) operations in group $\bar{\imath}$:

   (a) Schedule $0 : c_{i_1}$, then rc_{i_1}, then $c_{i_1} : i$.

   (b) Then repeat for $k = 2, \ldots, m_i$: Schedule $i : c_{i_{k-1}}$, then the unscheduled $c_{i_{k-1}} : c_{i_k}$, then rc_{i_k}, and then $c_{i_k} : i$.

   (c) Schedule the unscheduled $i : \hat{\imath}$, then schedule *read&write*$(a, \hat{\imath} : 0, \Delta_{\bar{\imath}} - 1, \Delta_{\bar{\imath}})$ to complete group i.

   (d) Next, schedule $0 : c_{\bar{\imath}_1}$, then $c_{\bar{\imath}_1} : \bar{\imath}$, then $r\bar{\imath}$, then $\bar{\imath} : c_{\bar{\imath}_1}$, and then $rc_{\bar{\imath}_1}$.

   (e) Then repeat for $k = 2, \ldots, m_{\bar{\imath}}$: Schedule *read&write*$(a, c_{\bar{\imath}_{k-1}} : c_{\bar{\imath}_k}, \Delta_{\bar{\imath}} + 7(k-1) + 1, \Delta_{\bar{\imath}} + 7(k-1) + 2)$, then $c_{\bar{\imath}_k} : \bar{\imath}$, then $\bar{\imath} : c_{\bar{\imath}_{k-1}}$, then the unscheduled $c_{\bar{\imath}_{k-1}} : c_{\bar{\imath}_k}$, and then $rc_{\bar{\imath}_k}$.

   (f) Finally, to complete group $\bar{\imath}$, schedule $c_{\bar{\imath}_{m_{\bar{\imath}}}} : \hat{\imath}$ and then *read&write*$(a, \hat{\imath} : 0, \Delta_{i+1} - 1, \Delta_{i+1})$.

   The case where v_i is set to false by $\mathcal{T}$ is symmetric and left to the reader.

The reader may verify that this is a legal schedule.

The proof of the converse, i.e., if $\mathcal{V}$ is a positive instance, then $\mathcal{F}$ is satisfiable, parallels the proof given in Lemma 3.6 and is left to the reader. □

Since the above transformation can be done in polynomial time, Theorem 4.11 follows. □

4.4. Providing the conflict-order. Two operations on the same location *conflict* if at least one is a write. In the *VSC-conflict* and *VL-conflict* problems, both a read-mapping and a write-order are known, implying that all conflicting operations are ordered; a legal schedule must respect this *conflict-order* relation.

VSC-conflict. There is a simple $O(n \log n)$-time algorithm for the VSC-conflict problem. Thus although providing either the read-mapping or the write-order is NP-complete, providing both yields a fast algorithm.

THEOREM 4.13. *There is an $O(n \log n)$-time algorithm for the VSC-conflict problem.*

Proof. Sort the VSC-conflict instance by address. We check to see that each read is mapped to a write operation with the same address and value and that each write is included only in the write-order for its address; otherwise, we have a negative instance. Construct a graph G with a vertex for each operation in the instance. Add an edge between a vertex u and a vertex v if u immediately precedes v in some processor sequence. In addition, add edges from each write to all reads mapped to it and from the write and these reads to the next write in its write-order. The graph G has n vertices and $O(n)$ edges.

The reader may verify that if G has a directed cycle, we have a negative instance. Otherwise, we have a positive instance, and any topological sort of G yields a legal schedule. □

VL-conflict. An $O(n \log n)$-time algorithm for the VL-conflict problem follows as a corollary to Theorems 3.2 and 4.13.

COROLLARY 4.14. *There is an $O(n \log n)$-time algorithm for the VL-conflict problem.*

4.5. Atomic read–modify–write. We conclude this section by showing how the algorithmic results of this paper can be extended to handle atomic read–modify–write operations at no asymptotic penalty. The input consists of three types of memory operations: reads, writes, and read&writes. The read&write operations are included in both the domain and range of any read-mapping, as well as ordered by any write-order (in which case the value read must match the value written by the previous write).

Few processors. We first consider the scenario where the number of processors is bounded.

THEOREM 4.15. *The VL problem with read, write, and read&write operations, restricted to instances such that at any time t, there are at most k operations on the same address whose intervals contain t, can be solved in $O(n2^{O(k)} + n \log n)$ time.*

Proof. The proof parallels the proof of Theorem 3.7, with the following modification of the algorithm to handle the possible presence of read&write operations. Namely, condition 2 for including an edge between a vertex v and a vertex v' on consecutive levels of the graph defines a set A of operations that must be assigned times after the last write ω designated by v and before any writes not in A. With only reads and writes, it sufficed to test whether each read in A read either the value of the last write designated by v or the value of some write in A. With read&write operations, however, we cannot have, for example, the set A consist of two operations that each read the designated last write and write a new value. We use the following test instead. Consider a multigraph H consisting of a vertex for every value occurring in $A \cup \{\omega\}$, plus an extra dummy vertex z. Each read&write operation in A that reads a value d and writes a value d' defines a directed edge in H from the vertex for d to the vertex for d'. Each write operation defines a directed edge from z to the vertex for the value written. Each read operation defines a self-loop at the vertex for the value read. There is a directed edge from z to the vertex for the value written by ω. Then for each vertex y in H whose in-degree exceeds its out-degree by $j > 0$, we have j directed dummy edges in H from y to z.

LEMMA 4.16. *The set of operations in A can be safely scheduled if and only if H has an Eulerian circuit.*

Proof. If H has an Eulerian circuit, then consider the sequence S of operations defined by an Eulerian circuit that begins with the edge for ω. Then the schedule obtained from S by removing ω and all dummy edges contains all of A and has no reads-from violations. Conversely, if there is no Eulerian circuit, then either H has a vertex x whose out-degree exceeds its in-degree or it has a nonempty set of vertices not reachable from z. In the former case, note that $x \neq z$ since by construction the in-degree of z is necessarily at least as large as its out-degree. Moreover, since any vertex with an outgoing dummy edge has the same in-degree as out-degree, all edges incident to x correspond to operations in $A \cup \{\omega\}$. Hence there are more operations that read and modify x's value than there are operations that write it. It follows that any schedule of A will have a reads-from violation. In the latter case, any schedule of A will have a reads-from violation at the first scheduled operation from the unreachable set. □

We construct H and check to see that all vertices are reachable from z and have the same in-degree as out-degree in time linear in the size of A. Finally, consider the last write, ω', designated by v', which like ω may be either a write or a read&write. The final condition required for adding an edge from v to v' is that either ω' is in A and writes a value whose vertex in H has an outgoing dummy edge or, if there are no writes or read&writes in A, $\omega' = \omega$. □

Theorem 4.15 implies, for instance, the following corollary.

COROLLARY 4.17. *There is an $O(n \log n)$-time algorithm for the VL problem with read, write, and read&write operations and any fixed number of processors.*

Read-mapping. If a read-mapping is provided, then we have the following corollary to Theorem 4.2.

COROLLARY 4.18. *There is an $O(n \log n)$-time algorithm for the VL-read problem with read, write, and read&write operations.*

Proof. We modify the algorithm in Theorem 4.2 to handle read&write operations by simply replacing the use of *clusters* in that algorithm with *cluster sequences*, as defined below. Observe that at most one read&write operation can be mapped by the read-mapping to a single write or read&write; otherwise, we have a negative instance. Define a *cluster sequence*, $S = w_1, R_1, w_2, R_2, \ldots, w_k, R_k$, $k \geq 1$, to be a sequence of alternating operations and sets such that w_1 is an ordinary write operation, $w_2, \ldots, w_k$ are read&write operations with w_i mapped to w_{i-1} for $i = 2, \ldots, k$, there is no read&write operation mapped to w_k, and R_i is the set of ordinary reads mapped to w_i for $i = 1, \ldots, k$. Any legal schedule assigns w_1 an earlier time than any read in R_1, which is assigned an earlier time than w_2, and so on. Thus the start-of-interval time for an operation must be earlier than the end-of-interval time for any operation that is ordered after it in its cluster sequence; otherwise, we have a negative instance. A legal assignment of times defines an interval for a cluster sequence, from the time assigned to w_1 to the time assigned to the last read in R_k; only operations in this cluster sequence can be scheduled during this time interval. The algorithm and proof continue as in Theorem 4.2. □

Write-order and conflict-order. Since the old value on writes provides no additional information once a write-order is provided, we have the following corollary to Theorems 4.6 and 4.13 and Corollary 4.14.

COROLLARY 4.19. *There are $O(n \log n)$-time algorithms for the VL-write, VSC-conflict, and VL-conflict problems with read, write, and read&write operations.*

5. Conclusions. This paper provides the first formal and systematic study of the complexity of testing the correctness of an execution of a shared memory, based on the reads and writes observed by the individual processors. We define two combinatorial problems: the *verifying sequential consistency of shared-memory executions* (VSC) problem for testing for sequential consistency, and the *verifying linearizability of shared-memory executions* (VL) problem for testing for linearizability. We show that the VSC problem is NP-complete even when the number of processors, locations, or operations per processor is bounded. Moreover, the problem remains NP-complete even when additional information is provided, such as the pairing of reads to particular writes, the order of writes to a location, or the old value overwritten by each write. However, if the order of all conflicting operations to a location is provided, we obtain an $O(n \log n)$-time algorithm. Linearizability, in contrast, is more restrictive than sequential consistency, and our results show that these additional restrictions can be quite useful in testing for linearizability. In particular, we present $O(n \log n)$-time algorithms for several variants of the VL problem whose corresponding VSC variants

are NP-complete. On the other hand, we show that the VL problem is NP-complete even when given the old value overwritten by each write operation.

Efficient testing algorithms can be used by machine designers, system software writers, application programmers, and naïve users whenever doubt arises as to whether all aspects of the memory system are functioning properly. In [17, 19], we present approaches to implementing testing procedures on real multiprocessors or their simulators. In particular, we devise schemes for collecting the additional information needed for the read-mapping, write-order, old value on writes, or conflict-order. We also describe algorithms for heuristic testing of several processors at a time. The results in [17, 19] demonstrate interesting tradeoffs between the speed, accuracy, and obtrusiveness of various testing procedures.

We have presented several $O(n \log n)$-time algorithms; after sorting, these run in linear time. Depending on the assumptions made on the form of the input, one could apply integer sorting to obtain linear time bounds for these algorithms.

In this paper, we consider read, write, and read–modify–write operations. This is extended in [17, 19], where we also consider the load-reserved (a.k.a. load-linked, load-locked) and store-conditional operations [24] appearing in many recent architectures. More generally, one can consider testing shared memories that support various data structures, such as priority queues.

Finally, correctness conditions other than linearizability and sequential consistency can be considered. A number of correctness conditions from the domain of database transactions have been previously studied (e.g., [6, 25, 30]). It would be interesting to develop a general theory encompassing a wide range of correctness conditions so that tradeoffs between the generality of a correctness condition and its complexity may be better understood.

Acknowledgments. We thank Michael Merritt and Robert Cypher for discussions related to this work. We acknowledge the helpful comments of an anonymous referee.

REFERENCES

[1] S. V. Adve, *Designing memory consistency models for shared-memory multiprocessors*, Ph.D. thesis, University of Wisconsin, Madison, WI, 1993.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer, *Detecting data races on weak memory systems*, in Proc. 18th International Symposium on Computer Architecture, ACM, New York, 1991, pp. 234–243.

[3] Y. Afek, G. M. Brown, and M. Merritt, *Lazy caching*, ACM Trans. Programming Lang. Systems, 15 (1993), pp. 182–205.

[4] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld, *Computing with faulty shared objects*, J. Assoc. Comput. Mach., 42 (1995), pp. 1231–1274.

[5] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung, *The MIT Alewife machine: A large-scale distributed-memory multiprocessor*, in Proc. Workshop on Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers, Norwell, MA, 1991.

[6] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy, *Relative serializability: An approach for relaxing the atomicity of transactions*, in Proc. 13th ACM Symposium on Principles of Database Systems, ACM, New York, 1994, pp. 139–149.

[7] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, *The Tera computer system*, in Proc. 1990 International Conference on Supercomputing, ACM, New York, 1990, pp. 1–6.

[8] H. Attiya and J. L. Welch, *Sequential consistency versus linearizability*, ACM Trans. Comput. Systems, 12 (1994), pp. 91–122.

[9] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, *Checking the correctness of memories*, Algorithmica, 12 (1994), pp. 225–244.

[10] J. Bright and G. Sullivan, *Checking mergeable priority queues*, in Proc. 24th IEEE Fault-Tolerant Computing Symposium, IEEE, Piscataway, NJ, 1994, pp. 144–153.
[11] W. W. Collier, *Reasoning About Parallel Architectures*, Prentice–Hall, Englewood Cliffs, NJ, 1992.
[12] S. Frank, H. Burkhardt III, and J. Rothnie, *The KSR1: Bridging the gap between shared memory and MPPs*, in Proc. 1993 IEEE Compcon Spring, IEEE, Piscataway, NJ, 1993, pp. 285–294.
[13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.
[14] K. Gharachorloo and P. B. Gibbons, *Detecting violations of sequential consistency*, in Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1991, pp. 316–326.
[15] K. Gharachorloo, A. Gupta, and J. Hennessy, *Two techniques to enhance the performance of memory consistency models*, in Proc. 1991 International Conference on Parallel Processing, CRC Press, Boston, MA, 1991, pp. I:355–364.
[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, in Proc. 17th International Symposium on Computer Architecture, IEEE, Piscataway, NJ, 1990, pp. 15–26.
[17] P. B. Gibbons and E. Korach, *Testing for sequential consistency*, in preparation.
[18] P. B. Gibbons and E. Korach, *The complexity of sequential consistency*, in Proc. 4th IEEE Symposium on Parallel and Distributed Processing, IEEE, Piscataway, NJ, 1992, pp. 317–325.
[19] P. B. Gibbons and E. Korach, *On testing cache-coherent shared memories*, in Proc. 6th ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1994, pp. 177–188.
[20] P. B. Gibbons and M. Merritt, *Specifying nonblocking shared memories*, in Proc. 4th ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1992, pp. 306–315.
[21] P. B. Gibbons, M. Merritt, and K. Gharachorloo, *Proving sequential consistency of high-performance shared memories*, in Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1991, pp. 292–303.
[22] M. P. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Trans. Programming Lang. Systems, 12 (1990), pp. 463–492.
[23] R. Hood, K. Kennedy, and J. Mellor-Crummey, *Parallel program debugging with on-the-fly anomaly detection*, in Proc. 1990 International Conference on Supercomputing, ACM, New York, 1990, pp. 74–81.
[24] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, *A new approach to exclusive data access in shared memory multiprocessors*, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, 1987.
[25] V. Krishnaswamy and J. Bruno, *On the complexity of concurrency control using semantic information*, Technical Report TRCS 92-21, Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA, 1992.
[26] L. Lamport, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Trans. Comput., C-28 (1979), pp. 690–691.
[27] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, *The Stanford DASH multiprocessor*, IEEE Comput., 25 (1992), pp. 63–79.
[28] J. Mellor-Crummey, *Compile-time support for efficient data race detection in shared-memory parallel programs*, in Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, ACM, New York, 1993, pp. 129–139.
[29] R. H. B. Netzer and B. P. Miller, *Improving the accuracy of data race detection*, in Proc. 3rd ACM Symposium on Principles and Practice of Parallel Programming, ACM, New York, 1991, pp. 133–144.
[30] C. Papadimitriou, *The serializability of concurrent database updates*, J. Assoc. Comput. Mach., 26 (1979), pp. 631–653.
[31] C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, Rockville, MD, 1986.
[32] F. Pong and M. Dubois, *A new approach for the verification of cache coherence protocols*, IEEE Trans. Parallel Distrib. Systems, 6 (1995), pp. 773–787.
[33] D. Shasha and M. Snir, *Efficient and correct execution of parallel programs that share memory*, ACM Trans. Programming Lang. Systems, 10 (1988), pp. 282–312.
[34] J. M. Wing and C. Gong, *Testing and verifying concurrent objects*, J. Parallel Distrib. Comput., 17 (1993), pp. 164–182.

SIAM J. COMPUT.
Vol. 26, No. 4, pp. 1245–1275, August 1997

015

POLYNOMIAL METHODS FOR SEPARABLE CONVEX OPTIMIZATION IN UNIMODULAR LINEAR SPACES WITH APPLICATIONS*

ALEXANDER V. KARZANOV† AND S. THOMAS MCCORMICK‡

Abstract. We consider the problem of minimizing a separable convex objective function over the linear space given by a system $Mx = 0$ with M a totally unimodular matrix. In particular, this generalizes the usual minimum linear cost circulation and cocirculation problems in a network and the problems of determining the Euclidean distance from a point to the perfect bipartite matching polytope and the feasible flows polyhedron.

We first show that the idea of minimum mean cycle canceling originally worked out for linear cost circulations by Goldberg and Tarjan [*J. Assoc. Comput. Mach.*, 36 (1989), pp. 873–886.] and extended to some other problems [T. R. Ervolina and S. T. McCormick, *Discrete Appl. Math*, 46 (1993), pp. 133–165], [A. Frank and A. V. Karzanov, Technical Report RR 895-M, Laboratoire ARTEMIS IMAG, Université Joseph Fourier, Grenoble, France, 1992], [T. Ibaraki, A. V. Karzanov, and H. Nagamochi, private communication, 1993], [M. Hadjiat, Technical Report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, France, 1994] can be generalized to give a combinatorial method with geometric convergence for our problem. We also generalize the computationally more efficient cancel-and-tighten method.

We then consider objective functions that are piecewise linear, pure and piecewise quadratic, or piecewise mixed linear and quadratic, and we show how both methods can be implemented to find exact solutions in polynomial time (strongly polynomial in the piecewise linear case). These implementations are then further specialized for finding circulations and cocirculations in a network.

We finish by showing how to extend our methods to find optimal integer solutions, to linear spaces of larger fractionality, and to the case when the objective functions are given by approximate oracles.

Key words. separable convex optimization, unimodular linear spaces, min mean canceling, network flows

AMS subject classifications. 90C25, 68Q25, 90B10

PII. S0097539794263695

1. Introduction.

A *circulation* in a digraph $G = (V, E)$ is a vector $x \in \mathbb{R}^E$ satisfying the *conservation* condition

$$\sum_{k:(k,i)\in E} x_{ki} - \sum_{j:(i,j)\in E} x_{ij} = 0 \quad \text{for all } i \in V.$$

A vector $\pi \in \mathbb{R}^V$ of *node potentials* induces a *cocirculation* (or potential difference, or tension) $\Delta\pi \in \mathbb{R}^E$ in G defined by

$$\Delta\pi(i,j) = \pi_j - \pi_i \quad \text{for } (i,j) \in E.$$

* Received by the editors February 25, 1994; accepted for publication (in revised form) September 16, 1995. This research was performed while both authors were visiting Laboratoire ARTEMIS IMAG at Université Joseph Fourier de Grenoble, Grenoble, France.

http://www.siam.org/journals/sicomp/26-4/26369.html

† Institute for System Analysis, 9 Prospect 60 Let Oktyabrya, 117312 Moscow, Russia (karzanov@cs.vniisi.msk.ru). The research of this author was supported by the "Chaire municipale," Mairie de Grenoble, Grenoble, France.

‡ Faculty of Commerce and Business Administration, University of British Columbia, Vancouver, BC V6T 1Z2, Canada (stmv@adk.commerce.ubc.ca). The research of this author was supported by an NSERC Operating Grant, an NSERC Grant for Research Abroad, and a UBC Killam Faculty Study Leave Fellowship.

Suppose that for each $e \in E$ we are given an *objective* function $w_e : \mathbb{R} \to \mathbb{R}$ which is convex. Consider the problem of minimizing the separable *total objective* function, defined as

$$w(x) = \sum_{e \in E} w_e(x_e), \tag{1}$$

over the set $\mathcal{C}$ of circulations in G and consider a similar problem for the set $\mathcal{C}^\perp$ of cocirculations in G. This framework includes many special problems on circulations and cocirculations. The most popular is the classical *minimum linear cost circulation problem* (briefly, the *linear circulation problem*): given lower and upper arc bounds $l, u \in \mathbb{R}^E$ with $l \le u$ and arc costs $c \in \mathbb{R}^E$, find $x \in \mathcal{C}$ that minimizes $\sum_{e \in E} c_e x_e$ subject to $l_e \le x_e \le u_e$, $e \in E$. This is reduced to the above "unconstrained" problem by defining for $e \in E$

$$w_e(r) = \begin{cases} c_e u_e + \alpha(r - u_e) & \text{if } r > u_e, \\ c_e r & \text{if } l_e \le r \le u_e, \\ c_e l_e + \alpha(l_e - r) & \text{if } r < l_e, \end{cases} \tag{2}$$

where α is a sufficiently large positive number. (We could also just set $w_e(x_e) = +\infty$ when x_e is outside the bounds.) Thus we obtain a special case of separable piecewise linear objective function w; see Figure 1. In the case of the *minimum linear cost cocirculation problem* (briefly, *linear cocirculation problem*), let a^+ (a^-) denote the vector of positive parts (negative parts) of vector a, i.e., the vector with ith component $\max\{a_i, 0\}$ ($\max\{-a_i, 0\}$). Then the dual objective function can be written as

$$\max_\pi \{l^T(\bar{c}^\pi)^+ - u^T(\bar{c}^\pi)^-\} \tag{3}$$

(see [8]), where $\bar{c}^\pi_e = c_e - \Delta\pi_e$ is the vector of reduced costs. Equation (3) shows that the classic dual circulation (cocirculation) objective is the piecewise linear function shown in Figure 2.

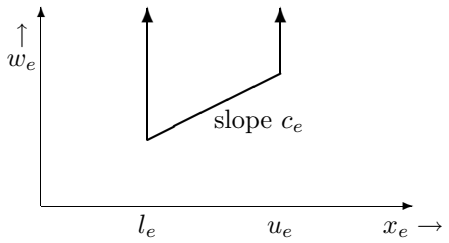


FIG. 1. *The arc objective function for the classic linear circulation problem with lower bound l_e, upper bound u_e, and cost c_e.*

It is tempting to further generalize this model by allowing demands at nodes, by putting convex penalties on violations of conservation, or by putting convex costs on the node potentials, etc. However, all of these generalizations can be handled by adjoining a new node 0 to the network, along with arcs from 0 to every other node. These new arcs can then carry demands, conservation violations, or node potentials. Thus we do not really lose any generality by considering only the simpler model.

Among the many approaches to solving the linear circulation problem (see, e.g., Ahuja, Magnanti, and Orlin [1] for a survey), one strongly polynomial method is especially simple, namely, the minimum mean cycle canceling method due to Goldberg

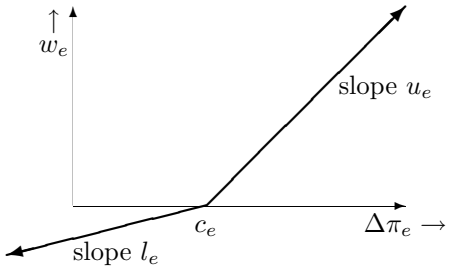


FIG. 2. *The arc objective function for the classic linear cocirculation problem with lower bound l_e, upper bound u_e, and cost c_e.*

and Tarjan [5]. Its ideas inspired a number of polynomial and strongly polynomial algorithms for other problems, such as the linear cocirculation problem [2], determining the Euclidean distances to certain polyhedra [4, 12], and the min-cost cocirculation (tension) problem [7]. (However, it appears that it does not work well for submodular flow problems; see [20].)

The first purpose of the present paper is to show that the "minimum mean canceling method" can be unified and generalized to solve a wide class of problems. We generalize it in two directions at once: to deal with separable convex objectives as in (1); and to arbitrary unimodular linear spaces L (i.e., the solution set of system $Mx = 0$ with a totally unimodular matrix M). The sets of circulations and cocirculations of a digraph are special cases of such an L. The method we develop is combinatorial and has geometric dual convergence in terms of the ℓ_1-distance of the current vector of partial (left and right) derivatives to an optimal dual vector.

There is a faster version of the minimum mean cycle canceling method for the linear circulation problem, the "cancel-and-tighten method" [5, 2]. We show that this also can be generalized to our generic case to give a method whose dual convergence is at essentially the same geometric rate but which has much faster iterations than our first method.

This paper considers several specializations and two generalizations of our general model by varying the form of the objective functions and the type of linear space. An overview of the various cases and their running times is given in the next section, with details in sections 3–12. Two generalizations go beyond our model: our methods can solve problems where the variables are restricted to be integer (see section 12) and problems whose constraint matrices are not totally unimodular but have "bounded fractionality" (see section 10). We also consider practical implications of duality for these problems in section 11 and the accuracy attainable by our methods when the objective functions are represented by approximate oracles in section 13.

Polynomial algorithms for the circulation problem with nonlinear separable costs based on capacity scaling were designed by Florian [3] and Minoux [23, 24]. In particular, [24] gives a polynomial algorithm for finding an optimal integer-valued circulation in a network with quadratic costs; this algorithm was speeded up and extended to general convex costs in [1, Chapter 14]. Our methods do not require any scaling, are as fast, and work with the problem in a more combinatorial way.

Hochbaum and Shanthikumar [11] gave a polynomial method for separable convex minimization over a linear space which is also based on capacity scaling. Their method involves solving linear programs where each original variable gets replicated $4n$ times (n is the number of original variables). By contrast, our methods work in the space

of original variables at all times and appear to be faster in general. For the case of separable convex circulation problems on networks, the Hochbaum–Shanthikumar algorithm can be specialized into a practical algorithm [10, 1], with about the same running time as our method (see section 2).

On the other hand, our methods assume the existence of a stronger oracle than what is assumed in [11]. That is, our oracle (O) (ii) in section 3 essentially assumes that we can do root-finding for the derivatives of the objective functions, whereas [11] assumes only an evaluation oracle. There are well-developed methods for root-finding, so our oracle is reasonable in practice. However, in theory, as [10] points out, there is a lower bound on root-finding even for square roots with the floor operation that is not strongly polynomial [17], and this shows that our oracle cannot be simulated with a strongly polynomial number of calls to an evaluation oracle.

In the general case, our methods depend on having an efficient procedure for solving various linear programs (LPs) over L. Such LPs can be solved in strongly polynomial time, e.g., by Tardos's version [32] of the ellipsoid method. For circulations and cocirculations on networks, we use fast combinatorial algorithms in place of solving LPs.

There are many applications for convex separable cost circulations; see [1, Chapter 14]. Many of these applications involve piecewise quadratic and/or linear objective functions. However, our method should be useful also for more general nonlinear problems. An example of this comes from applications to queuing networks where modelers are interested in the arc congestion function $w_e(x_e) = a_e x_e/(u_e - x_e)$ on the interval $[0, u_e)$ (see, e.g., [16]).

Applications for cocirculations are rarer, but they do arise. The best known example is the dual to the minimum-cost network flow problem, as discussed above. Chapter 7F of [30] contains several other more general applications of cocirculations. A more nonlinear application is to traffic flow equilibrium in congested cities, where the cocirculation represents differences in timings of traffic lights; see Hassin [9]. This application is not convex, but perhaps our methods could be used as a heuristic for quickly finding local optima. Our cocirculation methods should be also useful in PERT networks, where there are often nonlinear penalties for the elapsed time (time difference) for the task represented by an arc.

An example of a problem that is not unimodular is the well-known fractional b-matching problem (see, e.g., [6]). Such problems have the node-edge incidence matrix of an undirected graph as the essential part of their coefficient matrix. This problem turns out to have fractionality 2, which is certainly bounded, so the extension of our methods in section 10 will give strongly polynomial algorithms for this problem.

A more significant nonunimodular application arises when we add a new set of linear constraints $d^i x = 0$, $i = 1, \dots, p$, to the unimodular system $Mx = 0$, where all entries d^i_j are integers and the sum of their absolute values is at most a fixed number k. For example, in a network, we could add the equalities $x_e = x_{e'}$ for $k/2$ pairs of arcs $\{e, e'\}$ in the network. Then 2^k (which is constant because k is fixed) is an upper bound on any subdeterminant of the resulting matrix and so also on the fractionality of the problem. Thus our methods are again polynomial.

2. An overview of the results. Let L be a subspace of the Euclidean space $\mathbb{R}^E$ with coordinates indexed by the elements of a set E, and let $L^\perp$ be the orthogonal complement of L. We study the unconstrained (within L) optimization problem

(4) find $x \in L$ such that $w(x) = \sum_{e \in E} w_e(x_e)$ is as small as possible,

where each w_e is convex. We may assume that L contains a nonzero point; otherwise, the problem is trivial. We also assume that (4) has a *finite* optimal solution x. This assumption can be checked by solving two linear programs, one which computes an $x \in L$ satisfying any bounds implicit in the w_e's and another which computes an $h \in L^\perp$ that satisfies any dual bounds (see [30]).

We now impose the important restriction that L is a *unimodular space* given by a totally unimodular matrix M with n rows and m columns (the columns correspond to the elements of E, so $|E| = m$ also). We assume w.l.o.g. that M has full row rank n, so we can permute and partition the columns of M as $(B \quad N)$, where B is $n \times n$ and nonsingular. Since pivoting preserves total unimodularity, the matrix $(I \quad B^{-1}N)$ gives an alternative unimodular representation of L.

Now the $(m-n) \times m$ matrix $K = ((B^{-1}N)^T \quad -I)$ clearly has full row rank and is totally unimodular, and it is easy to check that $\{h \in \mathbb{R}^E : Kh = 0\}$ is exactly $L^\perp$. Thus $L^\perp$ is a unimodular linear space which is represented by matrix K. For convenience, we will often call the vectors in L ($L^\perp$) *circulations* (*cocirculations*).

A nonzero vector $\xi \in L$ ($\xi \in L^\perp$) whose support $\text{supp}(\xi) = \{e \in E : \xi_e \neq 0\}$ is inclusionwise minimal is said to be a *cycle* (resp. *cocycle*) of L. Since a positive multiple of a cycle essentially defines the same cycle, we will henceforth assume that a cycle is the canonical representative of this class that is integral with no common divisors. Then a unimodular linear system is distinguished by the fact that all of its cycles (and cocycles) are $(0, \pm 1)$ vectors. The *size* of a cycle or cocycle is its number of nonzero components. Key parameters for us will be ϕ ($\phi^\perp$), the maximum size of a cycle (resp. cocycle), and ρ ($\rho^\perp$), one plus the rank of M (resp. K); clearly, $\phi \leq \rho = n+1$ and $\phi^\perp \leq \rho^\perp = m-n+1$.

Example. Let $\mathcal{D} = (V, E)$ be a digraph with node-arc incidence matrix M. Then M is totally unimodular and $L = \{x \in \mathbb{R}^E : Mx = 0\}$ is the set of circulations for $\mathcal{D}$. The set of cycles for this L is exactly the set of simple circuits (with possible forward and backward arcs) in $\mathcal{D}$, and the set of co-cycles is exactly the set of simple cuts (with possible forward and backward arcs) in $\mathcal{D}$. The dual space $L^\perp$ is the space of cocirculations. Conversely, if M is instead a matrix whose rows are the incidence vectors of a circuit basis, then M is again totally unimodular and $L = \{x \in \mathbb{R}^E : Mx = 0\}$ is the set of cocirculations for $\mathcal{D}$. The set of cycles for this L is exactly the set of simple cuts in $\mathcal{D}$, and the set of co-cycles is exactly the set of simple circuits in $\mathcal{D}$. The dual space $L^\perp$ is the space of circulations.

Our first algorithm will find a cycle ξ such that updating the current x along ξ by step length α via $x := x + \alpha\xi$ will decrease the objective value. The cost of cycle ξ w.r.t. x is the local change in objective value from this change. A minimum mean cycle is a cycle whose cost per nonzero component is minimum. The min mean canceling method (MMCM) chooses a min mean cycle with an appropriate step length at each iteration. Section 3 will show that MMCM is a polynomial-time algorithm, but its time bounds are not impressive. One reason is that it is slow (but polynomial) to compute min mean cycles. Another is that it takes $O(\rho^\perp)$ MMCM iterations to achieve a relatively modest reduction in its measure of convergence.

We propose a second method, called *cancel-and-tighten*, that overcomes both of these problems. Section 5 shows that a single cancel-and-tighten iteration achieves essentially the same reduction in the convergence parameter that takes $O(\rho^\perp)$ MMCM iterations, without the need to compute any min mean cycles.

We shall consider various special and general cases of problem (4). First, consider the objective functions w_e. In the piecewise linear and/or quadratic case, we use C

to represent the largest number among the data. More precisely, when the w_e's are all piecewise linear with integer slopes, then C denotes the absolutely largest finite slope. When the w_e's are piecewise mixed linear and quadratic with all coefficients of all pieces and all breakpoints integer, then C denotes the absolutely largest finite number among all these data. We look at the cases where for all $e \in E$, we have the following:

1. The w_e's are *general*, i.e., they have no special properties other than convexity. In this case, we shall assume that we have some initial bound β on the "deviation" of an initial solution from optimality and that we want to get a final solution whose "deviation" is at most ε, in a sense to be made precise in section 3.

2. The w_e's are piecewise linear, with all slopes integral.

3. The w_e's are again piecewise linear, but the slopes may be general. Define B as the number of linear pieces in all of the w_e's; note that $O(B)$ pieces of data are needed to represent such a set of w_e's, so B can legitimately appear in a strongly polynomial bound. Cases 2 and 3 are dealt with in section 6.

4. The w_e's are piecewise quadratic with all data integral.

5. The w_e are piecewise mixed linear and quadratic with all data integral. Cases 4 and 5 are dealt with in section 7.

6. The w_e's are general, but we are looking for an optimal *integral* solution. This case actually concerns constraints, not objectives, but it is dealt with in section 12 by transforming it to the case of piecewise linear objectives.

Now consider a generalization and some specializations of the space L:

a. We generalize by allowing L to be any linear space represented by a rational matrix. We measure the deviation of L ($L^\perp$) from unimodularity by its *fractionality* q ($q^\perp$), which is the absolutely largest component of any cycle of L ($L^\perp$) when it is scaled to have all integral components with no common divisors. Thus unimodular spaces have fractionality $q = q^\perp = 1$. Case a is dealt with in section 10.

b. This is the standard unimodular case, which is dealt with in section 3.

c. Here we specialize to the case where L is the space of circulations of a network.

d. Here we specialize to the case where L is the space of cocirculations of a network. Cases c and d are dealt with in section 8.

The running-time bounds of our methods will involve the parameters ϕ, $\phi^\perp$, ρ, and $\rho^\perp$. In most instances, we will use the upper bounds $\phi \le \rho \le O(n)$ and $\phi^\perp \le \rho^\perp \le O(m-n) \le O(m)$. However, there are some situations where our sharper bounds are useful. For example, if $\mathcal{D}$ is a bipartite network with n_1 left nodes and n_2 right nodes, where $n_1 \ll n_2$, then $\phi = O(n_1)$. As another example, if $m = n + O(\sqrt{n})$, then $\rho^\perp = O(\sqrt{n})$.

In order to express running time of algorithms for computing min mean cycles below, we need two more definitions. Let $LP_q(n, m)$ denote the time to solve an $n \times m$ linear program over a space of fractionality q, and let $MC(n, m)$ denote the time to find a min cut separating two given nodes in a network with n nodes and m arcs.

THEOREM 2.1. *The numbers of iterations and running times per iteration for MMCM and cancel-and-tighten under the above assumptions about the objective functions and constraints are as given in Table* 1 *below. The bounds in row* 1 *are for approximate solutions, and the bounds in the other rows are for finding exact solutions.*

We now give a "proof" of Theorem 2.1 that outlines the main ideas behind its bounds, while leaving the harder technical details for later sections.

TABLE 1

Table of main results. Abbreviations PWL, PWQ, and PWLQ stand for "piecewise linear," "piecewise quadratic," and "piecewise mixed linear and quadratic," respectively. C+T stands for "cancel-and-tighten."

Type of Objective	Type of Constraints			
	(a) Fractionality q	(b) Unimodular	(c) Network Circulation	(d) Network Cocirculation
Number of MMCM Iterations				
(1) Gen'l	$O(\rho^\perp \phi q \log(\beta/\varepsilon))$	$O(\rho^\perp \phi \log(\beta/\varepsilon))$	$O(mn \log(\beta/\varepsilon))$	$O(mn \log(\beta/\varepsilon))$
(2) PWL, Int. Slopes	$O(\rho^\perp \phi q \log(\phi C))$	$O(\rho^\perp \phi \log(\phi C))$	$O(mn \log(nC))$	$O(mn \log(nC))$
(3) PWL, Gen'l Slopes	$O(Bq\rho^\perp \phi)$	$O(B\rho^\perp \phi)$	$O(Bmn)$	$O(Bmn)$
(4) PWQ, Int. Data	$O(\rho^\perp \phi qm \log C)$	$O(\rho^\perp \phi m \log C)$	$O(m^2 n \log C)$	$O(m^2 n \log C)$
(5) PWLQ, Int. Data	$O(\rho^\perp \phi qm \log C)$	$O(\rho^\perp \phi m \log C)$	$O(m^2 n \log C)$	$O(m^2 n \log C)$
(6) Int. Solution	NP hard	$O(\rho\phi^\perp \log(\phi^\perp C))$	$O(mn \log(nC))$	$O(mn \log(nC))$
(7) MMCM Time per Iteration	$O(\rho\phi q^2 LP_q(n,m))$	$O(\rho\phi LP_1(n,m))$	$O(mn)$ $O(\sqrt{n}m \log(nC))$	$O(nMC(n,m))$ $O(mn \log(nC))$
Number of Cancel-and-Tighten Iterations				
(1) Gen'l	$O(\rho q \log(\beta/\varepsilon))$	$O(\rho \log(\beta/\varepsilon))$	$O(n \log(\beta/\varepsilon))$	$O(m \log(\beta/\varepsilon))$
(2) PWL, Int. Slopes	$O(\rho q \log(\phi C))$	$O(\rho \log(\phi C))$	$O(n \log(nC))$	$O(m \log(nC))$
(3) PWL, Gen'l Slopes	$O(Bq\rho \log \phi)$	$O(B\rho \log \phi)$	$O(Bn \log n)$	$O(Bm \log n)$
(4) PWQ, Int. Data	$O(\rho qm \log C)$	$O(\rho m \log C)$	$O(mn \log C)$	$O(mn \log C)$
(5) PWLQ, Int. Data	$O(\rho qm \log(\phi C))$	$O(\rho m \log(\phi C))$	$O(mn \log(nC))$	$O(mn \log(nC))$
(6) Int. Solution	NP hard	$O(\rho^\perp \log(\phi^\perp C))$	$O(m \log(nC))$	$O(n \log(nC))$
(7) C+T (Amortized) Time per Iteration	$O(\rho^\perp LP_q(m,n))$	$O(\rho^\perp LP_1(m,n))$	$O(m \log n)$	$O(m + n \log n)$

Proof. The convergence proof for MMCM depends on analyzing $\lambda(x)$, the negative of the min mean cycle value w.r.t. the current point x. Lemma 3.4 shows that $\rho^\perp$ iterations of MMCM suffice to reduce $\lambda(x)$ by a factor of at most $(1 - 1/2\phi)$. Since

$$(1 - 1/2\phi)^{2\phi \log(\beta/\varepsilon)} < \varepsilon/\beta,$$

$O(\rho^\perp \phi \log(\beta/\varepsilon))$ MMCM iteration reduce $\lambda(x)$ from its initial value of at most β to a final value below ε. This is the result in box 1b of Table 1, and all other MMCM results follow from this.

To get the results in row 2 for MMCM with piecewise linear objectives with integral slopes, section 6 shows that $\lambda(x) \leq C$ for any feasible initial x, and Lemma 6.1 shows that if $\lambda(x) < 1/\phi$, then x is optimal. Thus the bounds in row 2 result from substituting C for β and ϕ for $1/\varepsilon$ in the row 1 bounds.

The results in row 4 for MMCM with piecewise quadratic objectives and integral data are derived similarly. Section 7 shows that we can choose an initial x with $\lambda(x) = O(C^{m+1})$ and that if we reduce $\lambda(x)$ below $1/(2C^{2m}\phi)$, then we can round x

to an optimal x using continued fractions (see [31]). Thus here $\log(\beta/\varepsilon) = O(m \log C)$.

The results in row 3 for MMCM with piecewise linear objectives and general slopes are slightly more complicated. Section 4 defines *reduced costs*, which are closely related to $\lambda(x)$; in fact, Corollary 4.2 and Lemma 4.3 show that $-\lambda(x)$ and the most negative reduced cost differ by a factor of at most ϕ. For piecewise linear objectives, there are at most B different values that reduced costs can take. By the same reasoning as for row 1, $O(\rho^{\perp}\phi \log \phi)$ MMCM iterations suffice to reduce $\lambda(x)$ by a factor of at least ϕ. By Corollary 4.2 and Lemma 4.3, this must cause a decrease in the most negative reduced cost. But this can happen at most B times. Finally, Lemma 11 from [28] applies here to remove the factor of $\log \phi$ from the iteration bounds.

The results in row 5 for MMCM with piecewise mixed linear and quadratic objectives follow the ideas for the piecewise quadratic case. Using the same ideas and number of iterations, we can compute the exact values of optimal x_e's that are located in "quadratic parts" of the objective functions. We are left with a piecewise linear problem with at most two pieces per element, which we can solve in the same number of iterations as in row 3 where $B = O(m)$. Thus these bounds are the sums of the bounds in row 3 and row 4.

The results in row 6 for MMCM (aside from column a) restricted to integral solutions start by noting that an equivalent problem results if we replace the w_e's with their piecewise linear approximations with breakpoints at all integers. Thus we effectively have instances with piecewise linear objectives (with possibly an unbounded number of pieces) with integral breakpoints and general slopes. We now apply the duality results from section 11, which say that when we dualize, the objective functions for the dual are piecewise linear, now with integral slopes and general breakpoints. Then the bounds from row 2 applied to the dual can be used with the parameters for L and $L^{\perp}$ interchanged.

The convergence proof for cancel-and-tighten in section 5 depends on analyzing $\overline{\lambda}$, an upper bound on $\lambda(x)$. Lemma 5.2 shows that one iteration of cancel-and-tighten reduces $\overline{\lambda}$ by a factor of at most $(1 - 1/2\rho)$. Then the same reasoning as in row 1 of MMCM gives the iteration bound in box 1b for cancel-and-tighten.

Rows 2–6 in Table 1 for cancel-and-tighten are obtained in the same way as the corresponding rows for MMCM, except that the trick from [28] does not apply to cancel-and-tighten in row 3.

We turn now to the columns of Table 1. Column a deals with linear spaces of fractionality q. Section 10 shows that essentially the same proofs go through except that the factor $(1 - 1/2\phi)$ of Lemma 3.4 is replaced by the factor $(1 - 1/(2q\phi))$ in Lemma 10.1. This leads to an extra factor of $O(q)$ in all of the bounds in column (a), except for the subcase asking for integral optimal solutions (box 6a). This problem is NP hard even for $q = 2$ since a special case of this is finding a maximum independent set (stable set) in a graph. It can be shown that there exists an optimal solution to the continuous relaxation of the linearized problem here with the denominators of all components bounded by Q, the least common multiple of all cycle vectors. For example, for the independent set problem, $q = Q = 2$. This would then lead to $O(\rho^{\perp} q^{\perp} \log(\phi^{\perp} QC))$ iterations to solve the continuous relaxation for MMCM and $O(\rho\phi^{\perp} q^{\perp} \log(\phi^{\perp} QC))$ iterations for cancel-and-tighten.

Columns c and d deal with linear spaces coming from networks with n nodes and m arcs. Column c deals with network circulation problems, and its bounds come from replacing ϕ and ρ by n and $\rho^{\perp}$ by m. Column d deals with network cocirculation problems, and its bounds come from replacing ϕ and ρ by m and $\rho^{\perp}$ by n.

Finally, we consider the two row 7's, the time per iteration rows. Section 9 gives a general algorithm for computing min mean cycles by solving linear programs, which is a refinement of a general method in [15]. The bound in MMCM box 7a comes from Theorem 9.2, and MMCM box 7b just sets $q = 1$ in this bound. MMCM box 7c gives the current fastest weakly and strongly polynomial bounds for computing a min mean cycle in a network (which dominates the time per iteration for MMCM); the top bound is from [26, 18] and the bottom one from [14]. MMCM box 7d does the same for computing a max mean cut in a network; the top bound is from [27], and the bottom bound is from [13, 18, 27].

The bounds in the cancel-and-tighten time-per-iteration row are implicit in section 5 in columns a and b and are covered explicitly in section 8 in columns c and d. To get the strongly polynomial bounds in cancel-and-tighten row 3, it is necessary to intersperse an iteration using an exact min mean cycle from time to time. Section 5 shows that it is possible to amortize the time for these (expensive) iterations over the remaining (cheaper) iterations. □

Note that when we specialize our general methods to piecewise linear objective functions on networks (rows 2 and 3, columns c and d times row 7), when we set $B = O(m)$ (as would be the case for classic linear circulation and cocirculation problems), they have the same time bounds as the best bounds in [5, 2]. Thus, despite being considerably generalized, our methods do not sacrifice any efficiency.

Taking the product of cancel-and-tighten box 6c with its time-per-iteration box 7d (we must use box 7d instead of 7c since we dualize when solving integer problems), we get a bound of $O(m \log(nC)(m + n \log n))$ total time for cancel-and-tighten for circulations in networks with general objectives and integer restrictions. This is roughly the same as the bound of $O(m \log C(m + n \log n))$ for an algorithm for the same problem in [1, Chapter 14]; see also [10].

Note that for problems with fractionality q, there is a factor of $O(q)$ more iterations, and each iteration takes a factor of $O(q^2)$ more time. Nevertheless, if q is bounded by a polynomial in $|E|$, the whole algorithm is still polynomial. For the fractional b-matching problem $q = 2$ and for the second example in the introduction, $q \leq 2^k$ for k fixed, so our methods are polynomial in both cases.

3. The minimum mean canceling method. In what follows, we utilize some well-known facts in convex analysis, linear algebra, and optimization; for references, see, e.g., Rockafellar [29] or Schrijver [31].

The convexity of w_e ensures the existence of the *right derivative*, denoted by $c_e^{\vdash}(r)$, and the *left derivative*, $c_e^{\dashv}(r)$, at every $r \in \mathbb{R}$. Note that $c_e^{\dashv}(r) < c_e^{\vdash}(r)$ is possible at a *breakpoint*. The interval $[c_e^{\dashv}(r), c_e^{\vdash}(r)]$ is known as the *subdifferential* of $w_e(r)$, and an $h \in [c_e^{\dashv}(r), c_e^{\vdash}(r)]$ is called a *subgradient* of w_e at r; see [29]. We refer to $c_e^{\vdash}(r)$ and $c_e^{\dashv}(r)$ as the *left (local) cost* and *right (local) cost* of e at r, respectively. Because of the convexity of w_e,

$$\text{if } r < r', \quad \text{then } c_e^{\dashv}(r) \leq c_e^{\vdash}(r) \leq c_e^{\dashv}(r') \leq c_e^{\vdash}(r'); \tag{5}$$

in particular, both $c_e^{\dashv}(r)$ and $c_e^{\vdash}(r)$ are monotone nondecreasing in r. The convexity of w_e implies that

$$\text{for any } r \in \mathbb{R}, \quad c_e^{\vdash} \text{ is continuous at } r \text{ in the set } [r, \infty) \text{ and } c_e^{\dashv} \text{ is continuous at } r \text{ in } (-\infty, r]. \tag{6}$$

It will be enough for us to assume that w_e is given by an *oracle* which

(O) (i) when given a point $r \in \mathbb{R}$ will return the costs $c_e^{\vdash}(r)$ and $c_e^{\dashv}(r)$;
(ii) when given a "slope" $p \in \mathbb{R}$ will return a point r with $c_e^{\dashv}(r) \le p \le c_e^{\vdash}(r)$.

In the case of piecewise linear or quadratic functions, (O) (i) and (ii) reduce to a simple linear or binary search, followed by solving a one-variable linear equation. Even for more complicated objectives such as the application to queuing networks [16], it is usually easy to compute both parts of (O) in practice.

We recall that any nonzero point (resp. any integral point) $x \in L$ is a nonnegative (resp. nonnegative integer) combination $\alpha_1\xi^1 + \cdots + \alpha_k\xi^k$ of $k \le |E|$ cycles $\xi^1, \ldots, \xi^k \in L$; moreover, these ξ^i's may be chosen so that

$$\xi_e^i \xi_e^j \ge 0 \quad \text{for } i, j = 1, \ldots, k \text{ and } e \in E. \tag{7}$$

This is called a *conformal decomposition* of x.

In what follows, we associate a $(0, \pm 1)$-vector ξ in $\mathbb{R}^E$ with the pair $F_\xi = (A_\xi, B_\xi)$, where $A_\xi = \{e : \xi_e = 1\}$ and $B_\xi = \{e : \xi_e = -1\}$. If ξ is a cycle (resp. cocycle) of L, we call F_ξ a *cycle* (resp. *cocycle*) also; the set of such F_ξ's is denoted by $\mathcal{F}$ (resp. $\mathcal{F}^\perp$). Define $|F|$, the *size* of F, as $|A| + |B|$.

For a function $f : S \to \mathbb{R}$ and a subset $S' \subseteq S$, we use $f(S')$ to denote $\sum(f_e : e \in S')$. Also, when it is not confusing, for a fixed point $x \in L$ and a cycle $F = (A, B)$, we denote $\sum_{e \in A} c_e^{\vdash}(x_e)$ by $c^{\vdash}(A)$ and denote $\sum_{e \in B} c_e^{\dashv}(x_e)$ by $c^{\dashv}(B)$.

Suppose that we have a cycle $F = (A, B)$ and a circulation $x \in L$ and we want to "augment x around F," i.e., increase x by an amount $\varepsilon > 0$ on the elements of A and decrease x by ε on the elements of B. Then the local change in objective function value per unit of ε is $c^{\vdash}(A) - c^{\dashv}(B)$, the *cost of* F *at* x, denoted by $c(x, F)$. We say that F is a *negative cycle* at x if $c(x, F) < 0$. The *mean cost* of F at x is $\overline{c}(x, F) := c(x, F)/|F|$. We say that F is a *minimum mean cycle* if $\overline{c}(x, F)$ is as small as possible. Define

$$\lambda(x) = \max\{0, -\min_{F \in \mathcal{F}} \overline{c}(x, F)\}. \tag{8}$$

If $\lambda(x)$ is positive, it is called the *absolute minimum mean value* for x.

The following lemma gives an optimality criterion for (4).

LEMMA 3.1. *Point $x \in L$ is an optimal solution to* (4) *if and only if there are no negative cycles at x or, equivalently, if $\lambda(x) = 0$.*

This lemma can be derived from general optimality theorems in convex programming, but to make our description self-contained, we prove it directly. Let $x \in L$. Suppose that $\overline{c}(x, F) = -\lambda < 0$ for some $F = (A, B) \in \mathcal{F}$. Then push a small amount $\varepsilon > 0$ around F to get $x' \in L$ with $w(x') < w(x)$, thus proving that x is not optimal. More precisely, we set

$$x'_e = \begin{cases} x_e + \varepsilon & \text{for } e \in A, \\ x_e - \varepsilon & \text{for } e \in B, \\ x_e & \text{otherwise,} \end{cases} \tag{9}$$

where $\varepsilon > 0$ is chosen so that $\delta_e(\varepsilon) := c_e^{\vdash}(x_e + \varepsilon) - c_e^{\vdash}(x_e) < \lambda$ for each $e \in A$ and $\delta_e(\varepsilon) := c_e^{\dashv}(x_e) - c_e^{\dashv}(x_e - \varepsilon) < \lambda$ for each $e \in B$. Such an ε exists because of (6). Since $c^{\vdash}$ and $c^{\dashv}$ are monotone, $w_e(x'_e) - w_e(x_e)$ is at most $\varepsilon c_e^{\vdash}(x_e) + \varepsilon\delta_e(\varepsilon)$ for $e \in A$ and at most $-\varepsilon c_e^{\dashv}(x_e) + \varepsilon\delta_e(\varepsilon)$ for $e \in B$, whence

$$w(x') - w(x) \le \varepsilon c(x, F) + \sum_{e \in A \cup B} \varepsilon\delta_e(\varepsilon) < \varepsilon c(x, F) + \varepsilon\lambda|F| = 0.$$

This proves the "only if" part of Lemma 3.1. The "if" part will follow from the next lemma. Given a circulation x and a cocirculation h, the *positive reduced cost* of element e is $c_e^{\vdash}(x_e) - h_e$ and its *negative reduced cost* is $h_e - c_e^{\dashv}(x_e)$. The next lemma characterizes $-\lambda(x)$ as being the largest possible lower bound on reduced costs when h varies over all vectors in $L^{\perp}$.

LEMMA 3.2. *A real λ is an upper bound on $\lambda(x)$ if and only if there exists $h \in L^{\perp}$ such that the reduced costs satisfy*

$$c_e^{\vdash}(x_e) - h_e \geq -\lambda \quad \text{and} \quad h_e - c_e^{\dashv}(x_e) \geq -\lambda \quad \text{for all } e \in E. \tag{10}$$

Moreover, if $\lambda = \lambda(x) > 0$, $F = (A, B)$ is a minimum mean cycle, and $h \in L^{\perp}$ satisfies (10)*, then $c_e^{\vdash}(x_e) - h_e = -\lambda$ for all $e \in A$ and $h_e - c_e^{\dashv}(x_e) = -\lambda$ for all $e \in B$.*

Proof. Let $\mathbb{1}$ denote a vector of ones, and consider the following optimization problem:

$$\begin{aligned} \min(c^{\vdash}x^+ - c^{\dashv}x^-)/(\mathbb{1}x^+ + \mathbb{1}x^-) \\ \text{s.t.} \quad M(x^+ - x^-) = 0, \\ x^+, x^- \geq 0. \end{aligned} \tag{11}$$

If (11) has an optimal solution, then it also has an optimal (x^+, x^-) with minimal support, which must then be a min mean cycle. Since (11) is homogeneous, it does not hurt to normalize by requiring that the denominator equals 1. Therefore, the problem remains the same if we delete the denominator to get the LP

$$\begin{aligned} \min c^{\vdash}x^+ - c^{\dashv}x^- \\ \text{s.t.} \quad M(x^+ - x^-) = 0, \\ \mathbb{1}x^+ + \mathbb{1}x^- = 1, \\ x^+, x^- \geq 0, \end{aligned} \tag{12}$$

with dual

$$\begin{aligned} \max \delta \\ \text{s.t.} \quad \pi M + \delta\mathbb{1} \leq c^{\vdash}, \\ -\pi M + \delta\mathbb{1} \leq -c^{\dashv}, \\ \pi, \delta \text{ free}. \end{aligned} \tag{13}$$

Note that since π is free, πM is an arbitrary vector h in $L^{\perp}$. Thus we can rewrite (13) as

$$\begin{aligned} \max \delta \\ \text{s.t.} \quad \delta\mathbb{1} \leq c^{\vdash} - h, \\ \delta\mathbb{1} \leq h - c^{\dashv}, \\ Kh = 0, \\ \delta \text{ free}. \end{aligned} \tag{14}$$

Dual LP (14) is asking for the largest (least negative) value of δ that satisfies (10). Then the first part of the lemma follows from LP duality.

For the second part, if F is a minimum mean cycle with, say, $c_e^{\vdash} - h_e > -\lambda(x)$ for some element in A, then since (10) is true for all elements of F, we would have $\bar{c}(x, F) > -\lambda$, contradicting that $\lambda = \lambda(x)$. □

From Lemma 3.2, we now derive the "if" part of Lemma 3.1. Suppose that $x^* \in L$ admits no negative cycles, i.e., $\lambda(x^*) = 0$. Then (10) yields

$$c_e^{\dashv}(x_e^*) \le h_e \le c_e^{\vdash}(x_e^*) \quad \text{for each } e \in E. \tag{15}$$

We show that (15) implies the optimality of x^*, i.e., $w(x) \ge w(x^*)$ holds for any $x \in L$. Since w is convex and any point in L is a nonnegative combination of cycles, it suffices to prove this for $x = x^* + \xi$, where ξ is a cycle of L. Let $F_\xi = (A, B)$. If $e \in A$, we have $x_e = x_e^* + 1$ and $c_e^{\vdash}(x_e) \ge c_e^{\vdash}(x_e^*)$, whence $w_e(x_e) - w_e(x_e^*) \ge c_e^{\vdash}(x_e^*)(x_e - x_e^*) = c_e^{\vdash}(x_e^*)$. If $e \in B$, we have $x_e = x_e^* - 1$ and $c_e^{\dashv}(x_e) \le c_e^{\dashv}(x_e^*)$, whence $w_e(x_e) - w_e(x_e^*) \ge -c_e^{\dashv}(x_e^*)$. Therefore,

$$w(x) - w(x^*) \ge \sum_{e \in A} c_e^{\vdash}(x_e^*) - \sum_{e \in B} c_e^{\dashv}(x_e^*) \ge h(A) - h(B) = h\xi = 0,$$

as required.

Lemma 3.2 is crucial for our MMCM. We assume that there is a procedure available to solve the following auxiliary problem (AP); algorithms for solving (AP) are discussed in section 9.

(AP): Given $x \in L$, find $\lambda = \lambda(x)$, and if $\lambda > 0$, find a minimum mean cycle $F = (A, B)$ at x and $h \in L^{\perp}$ satisfying (10).

MMCM starts with any reasonable $x \in L$ and computes $\lambda = \lambda(x)$. If $\lambda = 0$, stop since x is optimal (by Lemma 3.1). Otherwise ($\lambda > 0$), find F and h as in (AP). For each element $e \in F$, use answer (ii) of oracle (O) for w_e to compute the local step length ε_e as an increment Δ s.t.

$$\begin{cases} c_e^{\dashv}(x_e + \Delta) \le c_e^{\vdash}(x_e) + \lambda = h_e \le c_e^{\vdash}(x_e + \Delta) & \text{if } e \in A, \\ c_e^{\dashv}(x_e - \Delta) \le c_e^{\dashv}(x_e) - \lambda = h_e \le c_e^{\vdash}(x_e - \Delta) & \text{if } e \in B \end{cases} \tag{16}$$

(the equalities come from the second part of Lemma 3.2); clearly, $\varepsilon_e > 0$. The overall step length ε is $\min\{\varepsilon_e : e \in A \cup B\}$. That is, we increase ε until the first time a reduced cost (w.r.t. h) in F becomes nonnegative. Now push ε around F forming x' as in (9), and make x' the new current point x in L. Continue until λ is "small enough."

We claim that this method converges geometrically in λ. We show this using two lemmas. The first is similar to a lemma in Goldberg and Tarjan [5], and the second is similar to a lemma in Frank and Karzanov [4] with an improvement generalizing a result in Radzik and Goldberg [28]. Note that (16) and (5) imply that

$$\begin{aligned} &c_e^{\vdash}(x_e) \le c_e^{\dashv}(x'_e) \le c_e^{\vdash}(x_e) + \lambda \quad \text{for } e \in A, \\ &c_e^{\dashv}(x_e) \ge c_e^{\vdash}(x'_e) \ge c_e^{\dashv}(x_e) - \lambda \quad \text{for } e \in B. \end{aligned} \tag{17}$$

LEMMA 3.3. *If x' is obtained from x at an iteration, then $\lambda(x') \le \lambda(x)$.*

Proof. Let $h \in L^{\perp}$ be a cocirculation proving the optimality of $\lambda = \lambda(x)$, i.e., h is as in Lemma 3.2. By Lemma 3.2, it is enough to show that (10) holds for $c^{\vdash}(x')$ and $c^{\dashv}(x')$.

Since x' and x coincide outside F, it suffices to examine only elements in the minimum mean cycle $F = (A, B)$ at x that is chosen at the iteration. For $e \in A$, the inequality

$$c_e^{\vdash}(x'_e) \ge c_e^{\vdash}(x_e)$$

(since $x'_e > x_e$) together with (10) implies

$$c_e^{\vdash}(x'_e) \geq h_e - \lambda,$$

and the inequality

$$c_e^{\dashv}(x'_e) \leq c_e^{\vdash}(x_e) + \lambda$$

(by (17)) together with the equality $c_e^{\vdash}(x_e) = h_e - \lambda$ (by Lemma 3.2) implies

$$-c_e^{\dashv}(x'_e) \geq -c_e^{\vdash}(x_e) - \lambda = -h_e + \lambda - \lambda > -h_e - \lambda,$$

as required. The proof for $e \in B$ is similar. □

For the next lemma, we superscript everything by the iteration number.

LEMMA 3.4. *After every $\rho^{\perp}$ iterations, $\lambda = \lambda(x)$ decreases by a factor of at most $(1 - 1/2\phi)$, i.e., $\lambda^{i+\rho^{\perp}} \leq \lambda^i(1 - 1/2\phi)$.*

Proof. Fix $h := h^i \in L^{\perp}$ proving the optimality of $\lambda = \lambda(x^i)$, and consider iterations $i, i+1, \ldots, i+\rho^{\perp}$. For each $j = i, \ldots, i+\rho^{\perp}$, call $e \in E$ *positively* (*negatively*) *close to* h if $c_e^{\vdash}(x_e^j) - h_e \geq -\lambda/2$ (resp. $h_e - c_e^{\dashv}(x_e^j) \geq -\lambda/2$), and otherwise call it positively (negatively) *far from* h. Note that $c_e^{\vdash}(x_e) - c_e^{\dashv}(x_e) \geq 0$ implies that e cannot be positively and negatively far from h at the same time. The proof rests on the following two claims.

CLAIM 1. *If for all $k = i, \ldots, j-1$ the chosen cycle $F^k = (A^k, B^k)$ is such that each $e \in A^k$ is positively far from h and each $e \in B^k$ is negatively far from h, then $c_e^{\vdash}(x_e^j) - h_e \geq -\lambda$ and $h_e - c_e^{\dashv}(x_e^j) \geq -\lambda$ for all $e \in E$.*

Proof. The proof is by induction on j. Then $c_e^{\vdash}(x_e^j) \geq c_e^{\vdash}(x_e^{j-1}) \geq h_e - \lambda$ for $e \in E - B^{j-1}$ and $-c_e^{\dashv}(x_e^j) \geq -c_e^{\dashv}(x_e^{j-1}) \geq -h_e - \lambda$ for $e \in E - A^{j-1}$. Consider the remaining two cases. For $e \in A^{j-1}$, we have $c_e^{\dashv}(x_e^j) \leq c_e^{\vdash}(x_e^{j-1}) + \lambda^{j-1}$ (by (17)) and $c_e^{\vdash}(x_e^{j-1}) < h_e - \lambda/2$ (since e is positively far from h). Putting these inequalities together yields $c_e^{\dashv}(x_e^j) < h_e - \lambda/2 + \lambda^{j-1} < h_e + \lambda^{j-1}$, whence $-c_e^{\dashv}(x_e^j) > -h_e - \lambda^{j-1} \geq -h_e - \lambda$ (since $\lambda^{j-1} \leq \lambda$ by Lemma 3.3). The case for $e \in B^{j-1}$ is similar. □

We say that an element e in a cycle $F = (A, B) \in \mathcal{F}$ (i.e., $e \in A \cup B$) is close (far) if e is positively close (far) whenever $e \in A$ and negatively close (far) whenever $e \in B$.

CLAIM 2. *There is some iteration j ($i < j \leq i+\rho^{\perp}$) such that either* (i) $\lambda^j < \lambda/2$, *or* (ii) F^j *includes an element close to* h.

Proof. Suppose that for $j = 1, \ldots, i + \rho^{\perp} - 1$, no iteration j satisfies (i), i.e., $\lambda^j \geq \lambda/2$, and suppose that all iterations up to and including j have all elements of their cycles far from h so that Claim 1 applies. Denote by P (resp. N) the current set of elements positively (resp. negatively) close to h, and let $C = P \cap N$. By the *rank* of C, we mean the rank of the submatrix of K induced by the columns corresponding to elements of C (recall that K is a matrix representing $L^{\perp}$, with rank $\rho^{\perp} - 1$). We shall show that each iteration j increases the rank of C by at least one. This implies that after iteration $i + \rho^{\perp} - 1$, C meets the support of every cycle $\xi \in L$. Therefore, the cycle $F^{i+\rho^{\perp}}$ must include some element of C, proving (ii).

First, we show that if $e \in E$ belongs to P (resp. N) before iteration j, then e remains in P (resp. N) after the iteration. Consider a positively close e (the proof is similar for a negatively close e). Now e could become positively far only if $c_e^{\vdash}(x_e)$ decreased, which implies that $e \in B^j$. But then e is negatively far from h at iteration j, i.e., $-c_e^{\dashv}(x_e^j) < -h_e - \lambda/2$. This together with $c_e^{\vdash}(x_e^{j+1}) \geq c_e^{\dashv}(x_e^j) - \lambda^j$ (by (17))

yields $c_e^{\vdash}(x_e^{j+1}) > h_e + \lambda/2 - \lambda^j \geq h_e - \lambda/2$ so that e is not positively far from h after iteration j. Thus each of the sets P, N, and C is monotone nondecreasing.

Next, we show that there is at least one element e in F^j such that iteration j makes e close (recall that all elements in F^j are far). Lemma 3.3 says that $\lambda^j \leq \lambda$. By the choice of ε^j, either (a) A^j contains e with $c_e^{\vdash}(x_e^{j+1}) \geq c_e^{\vdash}(x_e^j) + \lambda^j$ or (b) B^j contains e with $c_e^{\dashv}(x_e^{j+1}) \leq c_e^{\dashv}(x_e^j) - \lambda^j$. Suppose that (a) takes place (case (b) is similar). Then (using Claim 1 for j) we get

$$c_e^{\vdash}(x_e^{j+1}) \geq c_e^{\vdash}(x_e^j) + \lambda^j \geq h_e - \lambda + \lambda^j \geq h_e - \lambda/2.$$

Thus iteration j adds this e as a new element to C.

Finally, we show that adding this e increases the rank of C. Suppose to the contrary that the rank of the set C at the beginning of iteration j equals the rank of $C \cup \{e\}$. Then a minimal vector $\xi' \in L^{\perp}$ proving the dependence of e on C corresponds to a cocycle, and $e \in \mathrm{supp}(\xi')$. Let $\xi \in L$ be the vector corresponding to F^j. Since $\xi\xi' = 0$ and $\xi_e\xi'_e \neq 0$, there exists another element $e' \neq e$ in both F^j and $\mathrm{supp}(\xi')$. But e is the only element of $\mathrm{supp}(\xi')$ not in C, so $e' \in C \cap F^j$. But each element of F^j was far at the beginning of iteration j, a contradiction. □

We may assume that $\lambda^j \geq \lambda/2$ for $j = i, \ldots, i + \phi^{\perp}$; otherwise, Lemma 3.3 shows that we are already done. Thus there must be an iteration j satisfying Claim 2(ii), i.e., with either (a) A^j containing an element u positively close to h or (b) B^j containing an element u' negatively close to h. Consider the smallest j with this property and assume that (a) takes place (case (b) is similar). Using Claim 1, we have

$$\begin{aligned} c(x^j, F^j) &= c(x^j, F^j) - h(A^j) + h(B^j) \\ &= \sum_{e \in A^j - \{u\}} (c_e^{\vdash}(x_e^j) - h_e) + (c_u^{\vdash}(x_u^j) - h_u) + \sum_{e \in B^j} (h_e - c_e^{\dashv}(x_e^j)) \\ &\geq -\lambda(|A^j| - 1) - \lambda/2 - \lambda|B^j| = -\lambda|F^j| + \lambda/2, \end{aligned}$$

whence $\lambda^j \leq \lambda(1 - 1/2\phi)$. □

This establishes MMCM box 1b in Table 1.

4. Cost distance and dual and primal convergence. Since $\lambda(x)$ is the best possible lower bound on reduced costs over all $h \in L^{\perp}$, it is a measure of dual convergence. However, it remains unclear whether this fact forces fast convergence of the current point x to an optimal point. Here we introduce another, closely related measure of the quality of the current point x which will enable us to show fast convergence of x to an optimal point for some special cases of w and to compute exact optimal solutions in sections 6 and 7.

Given $x \in L$ and $h \in L^{\perp}$, for each $e \in E$, define

$$\langle x_e, h_e \rangle = \begin{cases} h_e - c_e^{\vdash}(x_e) & \text{if } c_e^{\vdash}(x_e) < h_e, \\ c_e^{\dashv}(x_e) - h_e & \text{if } c_e^{\dashv}(x_e) > h_e, \\ 0 & \text{if } c_e^{\dashv}(x_e) \leq h_e \leq c_e^{\vdash}(x_e), \end{cases} \tag{18}$$

and then define

$$\langle x, h \rangle = \max_{e \in E} \langle x_e, h_e \rangle. \tag{19}$$

That is, $\langle x, h \rangle$ is the absolute value of the most negative reduced cost for x and h. Clearly, $\langle x, h \rangle \geq 0$, and if $\langle x, h \rangle = 0$, then x is optimal since (18) turns into (15). We call $\langle x, h \rangle$ the *cost distance* from x to h.

LEMMA 4.1. *Let x^* be an optimal solution to* (4) *and $h^* \in L^\perp$ prove its optimality, i.e., x^* and h^* satisfy* (15). *Let $x \in L$ and suppose that $\delta = \langle x, h^* \rangle > 0$. Then there exists a cycle $F \in \mathcal{F}$ whose cost $c(x, F)$ is at most $-\delta$.*

Proof. Let $u \in E$ be an element with $\langle x_u, h^*_u \rangle = \delta$. Then $x_u \neq x^*_u$ (otherwise, $\delta = 0$). Suppose for definiteness that $x_u < x^*_u$ (the reverse case is similar). Represent $z = x^* - x$ as $z = \alpha_1 \xi^1 + \cdots + \alpha_k \xi^k$, where $\alpha_1, \ldots, \alpha_k > 0$ and $\xi^1, \ldots, \xi^k$ are cycles of L satisfying (7). Since $z_u > 0$, there is ξ^i with $\xi^i_u = 1$; let $F = (A, B) \in \mathcal{F}$ correspond to ξ^i. Then $u \in A$, and from (7), it follows that

$$x^*_e > x_e \text{ for } e \in A \quad \text{and} \quad x^*_e < x_e \text{ for } e \in B.$$

By (5) and (15), $c^{\vdash}_e(x_e) \leq c^{\dashv}_e(x^*_e) \leq h^*_e$ for $e \in A$ and $c^{\dashv}_e(x_e) \geq c^{\vdash}_e(x^*_e) \geq h^*_e$ for $e \in B$. Also $c^{\vdash}_u(x_u) = h^*_u - \delta$. Thus

$$c(x, F) = \sum_{e \in A} c^{\vdash}_e(x_e) - \sum_{e \in B} c^{\dashv}_e(x_e) \leq -\delta + h^*(A) - h^*(B) = -\delta,$$

as required. □

COROLLARY 4.2. $\langle x, h^* \rangle \leq \phi \lambda(x)$.

Proof. If $\langle x, h^* \rangle > \phi \lambda(x)$, Lemma 4.1 shows that there is a cycle F with $c(x, F) < -\phi \lambda(x)$. But then $-\lambda(x) \leq \overline{c}(x, F) < -\phi \lambda(x)/|F| \leq -\lambda(x)$ (since $|F| \leq \phi$), a contradiction. □

In section 6, we will also need to bound $\lambda(x)$ by $\langle x, h^* \rangle$ on the other side.

LEMMA 4.3. $\lambda(x) \leq \langle x, h^* \rangle$.

Proof. Recall that $\lambda(x)$ is the minimum $\lambda \geq 0$ such that there is an $h \in L^\perp$ satisfying (10). Since h^* is one such h, we have

$$\lambda(x) \leq \max_e \max\{h^* - c^{\vdash}_e(x_e), c^{\dashv}_e(x_e) - h^*, 0\} = \langle x, h^* \rangle . \qquad \square$$

If we replace each $w_e(r)$ by $w'_e(r) = k w_e(r/k)$ for an integer $k > 1$, then the right cost $c'^{\vdash}$ for w' satisfies $c'^{\vdash}_e(kr) = c^{\vdash}_e(r)$, and similarly for the left cost $c'^{\dashv}$. Thus if we define $x' = kx$ and $x'^* = kx^*$, then x'^* is optimal for the problem with w', while h^* remains optimal for w'. Therefore, $\langle x', h^* \rangle = \langle x, h^* \rangle$, whereas the ℓ_∞-distance between x' and x'^* has increased by a factor of k. This makes it seem as if cost distance is of limited utility in measuring primal convergence.

However, cost distance is indeed a useful measure of primal convergence for an important class of functions w. We say that w_e is *fast growing* (f.g.) *with speed* α ($\alpha > 0$) in an interval σ of $\mathbb{R}$ if for any $r, r' \in \sigma$, $r < r'$, we have $c^{\dashv}_e(r') - c^{\vdash}_e(r) \geq \alpha(r' - r)$ (this is related to the concept of *strong convexity* in [25, Chapter 7]). For example, the quadratic function $w_e(r) = ar^2 + br + d$ ($a > 0$) is f.g. with $\alpha = 2a$ in $(-\infty, \infty)$. Other examples of f.g. functions are piecewise quadratic, cubic, and exponential functions. If w_e is f.g. for all $e \in E$, we say that w is fast growing, and its speed is defined to be the smallest speed of any w_e.

LEMMA 4.4. *If w_e is f.g. with speed α in an interval σ and both the current value x_e and optimal value x^*_e are in σ, then $|x_e - x^*_e| \leq \alpha^{-1} \langle x_e, h^*_e \rangle$.*

Proof. Assume that $x_e < x^*_e$ (the reverse case is similar) so that $\langle x_e, h^*_e \rangle = h^*_e - c^{\vdash}_e(x_e)$. Then w_e f.g. means that $x^*_e - x_e \leq \alpha^{-1}(c^{\dashv}_e(x^*_e) - c^{\vdash}_e(x_e))$. But the optimality of x^* and h^* implies that $h^* \geq c^{\dashv}_e(x^*_e)$ (by (15)). Plugging this into the bound on $x^*_e - x_e$ and using the value for $\langle x_e, h^*_e \rangle$ give the result. □

5. A faster method: Cancel-and-tighten. MMCM can take up to $\rho^{\perp}$ iterations to reduce $\lambda(x)$ by the factor $(1-1/2\phi)$ (these bounds can be tight; see [28]). It is reasonable to ask if there is a faster way to achieve the same reduction, particularly since both [5] and [2] give "cancel-and-tighten" algorithms which do this in the cases of circulations and cocirculations in networks.

The outline of our faster method is that the first half of each iteration is a *cancel* step that improves a primal x, and the second half is a *tighten* step that improves a dual h. The cancel step iteratively "cancels" (augments x around, in the sense of (9)) cycles whose elements are restricted to be sufficiently far away from the optimality condition (15). When no such cycles remain, we can compute a direction d in which to move our current dual estimate h that guarantees a $(1-1/2\rho)$ decrease (which is almost as good as the $(1-1/2\phi)$ decrease in MMCM). Thus our speedup comes from two sources: First, we do not have to compute minimum mean cycles, but rather we may cancel any sufficiently far cycles. Second, we get a decrease after only one iteration of cancel-and-tighten, rather than $\rho^{\perp}$ iterations of MMCM.

Instead of computing a min mean cycle to find the exact value of $\lambda(x)$, we keep a dual vector $h \in L^{\perp}$ that approximately satisfies the optimality condition (15) with x. We define $\overline{\lambda}$ as the absolute value of the most negative reduced cost w.r.t. h (i.e., $\overline{\lambda} = \langle x, h\rangle$), so that $\overline{\lambda}$ is an upper bound on $\lambda(x)$. By Lemma 3.2, h proves that $\overline{\lambda}$ is a valid upper bound via satisfying

$$c_e^{\vdash}(x_e) - h_e \geq -\overline{\lambda} \quad \text{and} \quad h_e - c_e^{\dashv}(x_e) \geq -\overline{\lambda} \quad \text{for all } e \in E. \tag{20}$$

We try to improve x and h by decreasing the gap between the costs and h. Define the subset $P^{\text{far}} \subseteq E$ (resp. $N^{\text{far}} \subseteq E$) consisting of elements that are positively (resp. negatively) far from h as defined in the proof of Lemma 3.4 (using $\overline{\lambda}$ in place of $\lambda(x)$), and define $E^{\text{far}} = P^{\text{far}} \cup N^{\text{far}}$.

Call a cycle $F = (A, B)$ with $A \subseteq P^{\text{far}}$ and $B \subseteq N^{\text{far}}$ a *far cycle*. The cancel step consists of finding and canceling far cycles until no more exist. Define $C = E - (P^{\text{far}} \cup N^{\text{far}})$ as the set of elements close to h. Then at each inner iteration of the cancel step, we find a minimal-support nonzero solution to the system

$$\begin{aligned} Mz &= 0, \\ z_e &\geq 0 \quad \text{for } e \in P^{\text{far}}, \\ &\leq 0 \quad \text{for } e \in N^{\text{far}}, \\ &= 0 \quad \text{for } e \in C. \end{aligned} \tag{21}$$

System (21) can be solved using linear programming techniques.

The computation of the step length ε for a far cycle $F = (A, B)$ found by solving (21) resembles (16) but is a bit different since the gaps between the costs and h_e's do not all have the same value $-\lambda$ here. Again using (O) (ii) for w_e, we compute

$$\varepsilon_e = \text{an increment } \Delta \text{ s.t.} \begin{cases} c_e^{\dashv}(x_e + \Delta) \leq h_e \leq c_e^{\vdash}(x_e + \Delta) & \text{if } e \in A, \\ c_e^{\dashv}(x_e - \Delta) \leq h_e \leq c_e^{\vdash}(x_e - \Delta) & \text{if } e \in B \end{cases} \tag{22}$$

and the overall step length $\varepsilon = \min\{\varepsilon_e : e \in A \cup B\}$. Then $\varepsilon > 0$.

LEMMA 5.1. *A cancel step preserves* (20) *and uses at most* $\rho^{\perp}$ *iterations, each canceling a far cycle.*

Proof. Since we are canceling cycles consisting only of elements far from h, Claim 1 in the proof of Lemma 3.4 shows that we cannot create any elements violating (20).

Furthermore, the proof of Claim 2 shows that we cannot create any new elements far from h either. Comparing (16) with (22) in the case where $e \in P^{\text{far}}$ (the other case is similar), since $h_e = c_e^{\vdash}(x_e) + (h_e - c_e^{\vdash}(x_e))$ for element e in (22), we are effectively using $h_e - c_e^{\vdash}(x_e)$ for the λ in (16); since $h_e - c_e^{\vdash}(x_e) > \overline{\lambda}/2$, case (i) of Claim 2 cannot hold. Thus $|P^{\text{far}}|$ and $|N^{\text{far}}|$ are monotone nonincreasing.

At least one element in each far cycle canceled (namely, an element determining ε) must become close to h and so drop out of P^{far} or N^{far}. In the same way as in the proof of Claim 2, each such newly close element must increase the rank of the submatrix of K induced by the elements close to h, and hence we can cancel at most $\rho^{\perp}$ cycles before no more far cycles exist. □

It might seem that canceling $\rho^{\perp}$ far cycles might be as much work as canceling $\rho^{\perp}$ min mean cycles, but for the specializations to networks in section 8, the total work to cancel the far cycles in one cancel can be done much faster than even one min mean cycle computation. Even in the general case, solving (21) for a far cycle is much faster than finding a min mean cycle.

The tighten step now looks for a *sufficiently improving direction*, which is a vector $d \in \mathbb{R}^E$ that satisfies the following conditions:

[A] d is an integral vector in $L^{\perp}$.
[B] For all e, $|d_e| \leq \rho - 1$.
[C] For all $e \in P^{\text{far}}$, $d_e \geq +1$.
[D] For all $e \in N^{\text{far}}$, $d_e \leq -1$.

LEMMA 5.2. *If d is a sufficiently improving direction and we replace h by $h' := h - (\overline{\lambda}/2\rho)d$, then $\overline{\lambda}$ is reduced to at most $\overline{\lambda}' := (1 - 1/2\rho)\overline{\lambda}$, i.e., (20) holds for h' and $\overline{\lambda}'$.*

Proof. By [A], $h' \in L^{\perp}$.

Suppose that e is not positively far from h. Then by [B],

$$\begin{aligned} c_e^{\vdash}(x_e) - h'_e &= c_e^{\vdash}(x_e) - h_e + d_e\overline{\lambda}/2\rho \\ &\geq c_e^{\vdash}(x_e) - h_e - (\rho - 1)\overline{\lambda}/2\rho \\ &= c_e^{\vdash}(x_e) - h_e - \overline{\lambda}/2 + \overline{\lambda}/2\rho. \end{aligned}$$

Since $c_e^{\vdash}(x_e) - h_e \geq -\overline{\lambda}/2$ for such an e, we have that this e satisfies (20) in the positive direction for h' and $\overline{\lambda}'$. The negative direction for the case where e is not negatively far from h is similar.

By [C], if $e \in P^{\text{far}}$, then

$$\begin{aligned} c_e^{\vdash}(x_e) - h'_e &= c_e^{\vdash}(x_e) - (h_e - d_e\overline{\lambda}/2\rho) \\ &\geq c_e^{\vdash}(x_e) - h_e + \overline{\lambda}/2\rho. \end{aligned}$$

Since e satisfied (20) in the positive direction for h and $\overline{\lambda}$, it also satisfies (20) in the positive direction for h' and $\overline{\lambda}'$. The negative direction for the case where $e \in N^{\text{far}}$ is similar. □

Lemma 5.2 shows that moving a distance of $\overline{\lambda}/2\rho$ in a sufficiently improving direction reduces $\overline{\lambda}$ by a factor of at least $(1 - 1/2\rho)$, as desired. To finish specifying the tighten step, it remains only to show how to (efficiently) compute a sufficiently improving direction.

Note that when the cancel step is done, system (21) has $z = 0$ as its only feasible solution. In particular, for each $u \in E^{\text{far}}$, there is no far cycle containing u. The following shows that there is a cocycle D^u with corresponding incidence vector ξ^u proving this fact.

LEMMA 5.3. *An element $u \in E^{\text{far}}$ is contained in no far cycle if and only if there is a cocycle $\xi^u \in L^\perp$ such that $\xi^u_u \neq 0$, $\xi^u_e \geq 0$ for all $e \in P^{\text{far}}$, and $\xi^u_e \leq 0$ for all $e \in N^{\text{far}}$.*

Proof. Applying Farkas' lemma to (21) with the additional constraint $z_u = 1$ or $z_u = -1$, we observe that u is in no far cycle if and only if there is a vector $\xi = y^T M$ feasible to

$$\begin{aligned} &\xi_e \geq 0 \quad \text{for } e \in P^{\text{far}}, \\ &\xi_e \leq 0 \quad \text{for } e \in N^{\text{far}}, \\ &\xi_u \neq 0. \end{aligned} \tag{23}$$

Now if (23) has a solution in $L^\perp$ then it has a solution that is a cocycle ξ^u. Conversely, if ξ^u as in the lemma exists then it satisfies (23), thus proving (by Farkas' lemma) that no far cycle contains u. □

Now the tighten step can compute d as follows. Find a subset B^{far} of E^{far} such that $\{\xi^u : u \in B^{\text{far}}\}$ forms a basis for $\{\xi^u : u \in E^{\text{far}}\}$. Then define d to be $\sum_{u \in B^{\text{far}}} \xi^u$.

LEMMA 5.4. *This d is a sufficiently improving direction.*

Proof. Since d is a sum of incidence vectors of cocycles, d certainly satisfies [A].

The dimension of the subspace of $L^\perp$ spanned by $\{\xi^e : u \in E^{\text{far}}\}$ (i.e., the cardinality of B^{far}) is at most the dimension of $L^\perp$, which is $\rho - 1$. This proves [B].

Consider $e \in E^{\text{far}}$. By Lemma 5.3, for all $u \in E^{\text{far}}$, $\xi^u_e \geq 0$ if $e \in P^{\text{far}}$ and $\xi^u_e \leq 0$ if $e \in N^{\text{far}}$. Furthermore, for the vector ξ^e, we have $\xi^e_e \neq 0$. Now since $\{\xi^u : u \in B^{\text{far}}\}$ generates all of $\{\xi^{e'} : e' \in E^{\text{far}}\}$, there must be at least one $u \in B^{\text{far}}$ such that $\xi^u_e \neq 0$. By Lemma 5.3, every such ξ^u_e must have the same sign as ξ^e_e. This proves [C] and [D]. □

In practice, to construct a sufficiently improving direction d efficiently, it suffices to find a basic feasible solution of the linear program obtained by adding to (23) the constraints $-\rho + 1 \leq \xi_e \leq \rho - 1$ $(e \in E)$, $\xi_e \geq 1$ $(e \in P^{\text{far}})$, and $\xi_e \leq -1$ $(e \in N^{\text{far}})$.

This establishes cancel-and-tighten box 1b in Table 1.

6. (Strongly) polynomial bounds on computing exact solutions with piecewise linear costs. Note that if we start with an integral x^0 (often $x^0 = 0$ will work), then with integral breakpoints, the algorithm will always maintain an integral solution. Thus with integral breakpoints, we can guarantee an integer optimal solution. We use x^* (resp. h^*) to denote an optimal circulation (resp. cocirculation).

Note that piecewise linear functions are not f.g. in the sense of section 4. Thus we will need to develop different techniques for proving that our methods can find exact optimal solutions.

6.1. A polynomial bound with integral slopes. We assume that all slopes are integral in this subsection. We borrow an optimality characterization from [5] that depends directly on $\lambda(x)$ and not on $\langle x, h^* \rangle$.

LEMMA 6.1. *In the piecewise linear case with integral slopes, if $\lambda(x) < 1/\phi$, then x is exactly optimal.*

Proof. Let $h \in L^\perp$ be a dual vector proving that $\lambda(x) < 1/\phi$. Then Lemma 3.2 says that the reduced cost of every element in cycle $F = (A, B)$ is more than $-1/\phi$. This implies that $c(x, F) = (c^\vdash - h)(A) + (h - c^\dashv)(B) > -|F|/\phi \geq -1$. But $c(x, F)$ is an integer, so it must be nonnegative. Thus by Lemma 3.1, x is optimal. □

Let x^0 denote our initial circulation. Each finite $c^\dashv_e(x^0_e)$ and $c^\vdash_e(x^0_e)$ will be equal to one of the $O(B)$ possible slopes so that $\lambda(x^0)$ and $\overline{\lambda}$ are at most C. This together with Lemma 6.1 establishes both row 2's in Table 1.

6.2. A strongly polynomial bound for general slopes. The proof of strong polynomiality for the case of linear network flow in [5] seems not to work when some w_e's have more than two breakpoints. However, the alternative proof below (which has the same flavor as a proof that [28] uses for the case of networks) is just as strong as the ones given for special cases in [5, 2, 7] since we will be able to derive exactly the same bounds on running time for those special cases in section 8. At the same time, this proof is somewhat simpler than the proofs in [5, 2, 7].

We use the very close relationship between $\lambda(x)$ and $\langle x, h^* \rangle$ given by Corollary 4.2 and Lemma 4.3, which say that neither number can get very far from the other. Each $c_e^{\vdash}(x_e)$ and $c_e^{\dashv}(x_e)$ equals one of the only $O(B)$ slopes, and we may assume that h^* stays fixed throughout the algorithm. Thus by (18) and (19), $\langle x, h^* \rangle$ always equals $h_e^* - c_e^{\vdash}(x_e)$, $c_e^{\dashv}(x_e) - h_e^*$, or 0 for some element e. Therefore, there are only $O(B)$ different values that $\langle x, h^* \rangle$ can take, a strongly polynomial number.

THEOREM 6.2. *In the piecewise linear case, MMCM requires $O(B\rho^{\perp}\phi \log \phi)$ iterations to compute an exact optimal circulation.*

Proof. Define a *big iteration* as a sequence of iterations that reduce $\lambda(x)$ by a factor of more than $1/\phi$. As argued in the proof of Theorem 2.1, a big iteration consists of $O(\rho^{\perp}\phi \log \phi)$ min mean cycle cancellations. Let x be our point before a big iteration and x' be the point after a big iteration. Then

$$\begin{aligned} \langle x', h^* \rangle &\le \phi\lambda(x') && \text{(by Corollary 4.2)} \\ &< \lambda(x) && \text{(by the definition of big iteration)} \\ &\le \langle x, h^* \rangle && \text{(by Lemma 4.3).} \end{aligned} \tag{24}$$

This shows that $\langle x, h^* \rangle$ strictly decreased during the big iteration. Thus after at most B big iterations, $\langle x, h^* \rangle$ has decreased B times, and its only possible remaining value is $\langle x, h^* \rangle = 0$, implying that x is optimal. □

The argument in Theorem 6.2 does not work for cancel-and-tighten because Lemma 4.3 might not be true for $\overline{\lambda}$ (but Corollary 4.2 *is* true with $\overline{\lambda}$ in place of $\lambda(x)$). That is, there is nothing in cancel-and-tighten to force $\overline{\lambda}$ to take large downwards steps, even when this is possible due to $\langle x, h^* \rangle$ becoming smaller than an optimal reduced cost. However, we can use an idea from [5] and at the beginning of every big iteration, we use a min mean cycle computation to replace $\overline{\lambda}$ by the exact value of $\lambda(x)$. Then (24) becomes true, and the rest of the proof of Theorem 6.2 goes through as before. In applications to networks, the (slow) min mean cycle iterations can be amortized over the (fast) tighten-step iterations so as to not slow down the fast asymptotic running time of cancel-and-tighten; see section 8.

7. Polynomial bounds on computing exact solutions with quadratic and linear costs and integer data. For this case, we assume that for each $e \in E$ the function w_e consists of a sequence of functions $f_e^i = a_e^i x_e^2 + b_e^i x_e + d_e^i$, $i = 1, \ldots, k_e$, where each f_e^i is defined on the interval between the breakpoints p_e^{i-1} and p_e^i ($p_e^{i-1} < p_e^i$); so $p_e^0 = -\infty$, $p_e^{k_e} = \infty$, and $f_e^i(p_e^i) = f_e^{i+1}(p_e^i)$. We assume that all a_e^i, b_e^i, and d_e^i are integers with $a_e^i \ge 0$ and that each finite p_e^i is an integer. If $a_e^i = 0$, then this piece is linear; otherwise, it is quadratic. Thus we have the case of *piecewise mixed linear and quadratic* costs. If all a_e^i are positive, then we have *piecewise quadratic* costs. Slightly redefine C to be the maximum number among all $4a_e^i$, $2|b_e^i|$, and (finite) $2|p_e^i|$ ($e \in E$).

If x_e^* is interior to piece i, define the bounds $p_e = p_e^{i-1}$ and $p_e' = p_e^i$ and force h_e to be the derivative of f_e^i by defining its bounds as $g_e = g_e' = 2a_e^i x_e + b_e^i$. If instead

$x_e^* = p_e^i$, then set $p_e = p'_e = p_e^i$ and set $g_e = c_e^{\dashv}(p_e^i)$ and $g'_e = c_e^{\vdash}(p_e^i)$. Then any x and h feasible to the following linear system must be optimal:

$$\begin{aligned} Mx &= 0, \\ Kh &= 0, \\ p_e \le x_e \le p'_e, &\quad e \in E, \\ g_e \le h_e \le g'_e, &\quad e \in E. \end{aligned} \tag{25}$$

It can be seen that the absolute value of any subdeterminant of the matrix combining all constraints in (25) is at most C^m, where $m = |E|$. This implies the following.

LEMMA 7.1. *In the piecewise mixed linear and quadratic case, there are optimal solutions x^* and h^* such that every component x_e^* of x^* and every component h_e^* of h^* is a rational with numerator and denominator at most C^m.*

We will also need a bound on the cost distance $\langle x^0, h^* \rangle$ between an initial x^0 and an optimal h^*. We can feasibly choose $x^0 = 0$. Each h_e^* is bounded above by $2a_e^i x_e^* + b_e^i$ for some i. By Lemma 7.1, each $x_e^* \le C^m$, so $\langle x^0, h^* \rangle = O(C^{m+1})$, and by Lemma 4.3, $\beta = \lambda(x^0) \le O(C^{m+1})$. Our goal will be to reduce $\langle x, h^* \rangle$ below $1/(2C^{2m})$; by Corollary 4.2, to do this it suffices to reduce $\lambda(x)$ below $\varepsilon = 1/(2\phi C^{2m})$. The running time bounds involve $\log(\beta/\varepsilon)$, which is $O(m \log C)$ since $\phi = o(C^{2m})$.

7.1. The piecewise quadratic case. Since w is strictly convex in this case, problem (4) has a unique optimal solution x^*. Thus if we reduce $|x_e - x_e^*|$ below $1/(2C^{2m})$, then there is a unique rational closest to x_e with denominator at most C^m, and by Lemma 7.1, this rational must be x_e^*. Furthermore, x_e can be efficiently rounded to x_e^* using standard continued fractions techniques [31]. Defining α to be the minimum of $2a_e^i$ among all e and i, each w_e is f.g. with speed α. Therefore, by Lemma 4.4, it suffices to decrease $\langle x_e, h_e^* \rangle$ (which is an upper bound on $\alpha|x_e - x_e^*|$) below $\alpha/(2C^{2m})$. This establishes both row 4's of Table 1.

7.2. The piecewise mixed linear and quadratic case. This case differs from the previous one because the w_e need not be f.g. when some pieces f_e^i are linear.

Our strategy is to run the algorithm long enough to be able to round the elements x_e whose x_e^*'s fall into quadratic pieces as above. After we fix x_e to be x_e^* on such pieces, we are left with a piecewise linear problem where each element has at most two pieces with finite slope, which we can solve in strongly polynomial time as in section 6.

More precisely, let $x \in L$ be any circulation. We say that e is a *quadratic element* of x if (i) x_e is interior to a quadratic piece, (ii) x_e equals a breakpoint between two quadratic pieces, or (iii) x_e equals a breakpoint between a quadratic piece and a linear piece with arbitrarily large slope representing a lower or upper bound; otherwise, e is a *linear element* of x. Define $\Delta = 1/C^m$ and redefine α to be the minimum of $2a_e^i$ over quadratic pieces.

Now run the algorithm in "phase 1" until $\langle x, h^* \rangle < \Delta^2/2$, so we can use continued fractions to round each x_e to a unique x'_e with denominator at most C^m. The next lemma will show that whenever e is a quadratic element of x', then we can set $x_e^* = x'_e$ without loss of optimality. Furthermore, it will show that when e is a linear element of x', we can constrain x_e^* to be in one of the at most two linear pieces adjacent to x'_e, and so we can complete x^* by solving a piecewise linear "phase 2."

LEMMA 7.2. *If element e is quadratic for x'_e, then $x'_e = x_e^*$. If x'_e is interior to a linear piece f_e^i, then every optimal x_e^* is in f_e^i's piece. If x'_e equals the breakpoint*

p_e^i between linear pieces f_e^{i-1} and f_e^i, then every optimal x_e^* is in f_e^{i-1}'s piece or f_e^i's piece. If x'_e equals the breakpoint p_e^i between a linear piece and a quadratic piece, then every optimal x_e^* is in the linear piece.

Proof. Suppose that e is quadratic for x'_e, and let x_e^* and h_e^* come from optimal solutions with denominators at most C^m as in Lemma 7.1. If $x_e^* > x'_e$ (the reverse case is similar), then $x_e^* - x'_e \geq \Delta^2$. Since $|x'_e - x_e| < \Delta^2/2$, we get $x_e^* - x_e \geq \Delta^2/2$. If x'_e is interior to a quadratic piece, let f_e^i be that quadratic piece; otherwise, x'_e equals a breakpoint, say p_e^{i-1}, and we may assume that f_e^i is quadratic. Then $h_e^* \geq 2a_e^i x_e^* + b_e^i$. But now, since $c_e^{\vdash}(x_e) \leq \max\{2a_e^i x_e + b_e^i, 2a_e^i x'_e + b_e^i\}$, we have $\Delta^2/2 > \langle x_e, h_e^* \rangle \geq 2a_e^i(\Delta^2/2)$, a contradiction. Thus we must have $x'_e = x_e^*$ so that e is quadratic for this x_e^* and so for all optimal solutions.

If x'_e is interior to linear piece f_e^i with slope b_e^i, then x_e must also be interior to f_e^i. Thus $\langle x_e, h_e^* \rangle < \Delta^2/2$ implies that

$$|b_e^i - h_e^*| < \Delta^2/2. \tag{26}$$

If there is an optimal x^* with corresponding h^*, where x_e^* does not belong to f_e^i, then by Lemma 7.1 we may assume that there is such a solution with denominators at most $1/\Delta$. If x_e^* is in the interior of a linear piece with slope $b_e^j = h_e^*$, then $|b_e^i - b_e^j| = |b_e^i - h_e^*| \geq 1$, contradicting (26). Otherwise, x_e^* is either interior to a quadratic piece or at a breakpoint nonadjacent to f_e^i. In either case, since x_e^* is outside of f_e^i's interval by at least $\Delta^2/2$, h_e^* must differ from b_e^i by more than $\Delta^2/2$, again contradicting (26). Thus x_e^* is in f_e^i's interval, forcing every optimal solution to use f_e^i's interval.

The remaining two cases have proofs much like the previous case. □

Now we have our algorithm: If e is quadratic for x', then we can fix $x_e^* = x'_e$. Otherwise, we use one of the three linear cases in Lemma 7.2 to constrain x_e to lie in either in a single linear piece or the union of two adjacent linear pieces. This establishes both row 5's of Table 1.

Many applications of piecewise mixed linear and quadratic costs do not need to solve the piecewise linear second phase. For example, if the only piecewise linear parts are bounds (as in [4, 12]), then all elements are quadratic for every solution, so the optimal solution will be fully determined after the first phase. Alternatively, if there is only one linear piece per element, then the second phase can be solved more simply as just a linear feasibility problem with bounds over L.

8. Implementations for networks. We assume that the digraph $G = (V, E)$ has n nodes and $m = |E|$ arcs so that the maximum size of a circuit is $O(n)$ and the maximum size of a cut is $O(m)$. Note that an initial circulation satisfying any bounds implicit in the arc cost functions can be computed with one max flow, and similarly an initial cocirculation can be computed with one shortest path [1]. Thus it is easy to initialize the algorithms and to verify that optimal solutions exist.

8.1. Network circulation algorithms. We consider how to implement cancel-and-tighten for circulations. Make an auxiliary graph consisting of the arcs in P^{far} and the reverses of the arcs in N^{far} so that far cycles are directed cycles. We then adapt a trick from [5] for the classic linear circulation problem. It develops an algorithm based on dynamic trees for the problem of canceling all cycles consisting solely of negative reduced-cost augmentable (admissible) arcs. The key of an arc in its method is the residual capacity of the arc in this direction. Instead, choose the number ε_e defined in (22) for every arc e far from h. Once the method finds and cancels a cycle, it then

cuts all arcs that are no longer augmentable out of the data structure. We apparently need to cut out all arcs that become close to h, but instead we relax our concept of the cancel step a bit. Note that the proof of Claim 1 remains true even if each cycle contains elements satisfying only that $c_e^{\vdash}(x_e) - h_e \le 0$ (for forward elements) or $h_e - c_e^{\dashv}(x_e) \le 0$ (for backwards elements). Thus we can cut out only arcs whose reduced costs become 0; this may cause some arcs that become newly close to h to stay in the dynamic trees and so participate in some future cancellation, but this will not cause a violation of (20). Furthermore, the proof of Claim 2 shows that no new arcs far from h can be created. Since each cancellation cuts at least one arc of the auxiliary graph out of the dynamic trees, we still get the $O(m)$ bound on the number of cancellations. Thus in the same way as in [5], this method takes only $O(m \log n)$ time.

When the cancel step is done, the auxiliary graph is acyclic, so an acyclic node labeling l exists. The label differences $\Delta l_{ij} := l_j - l_i$ form a sufficiently improving direction d for the tighten step and can be computed in linear time [5]. Thus one round of cancel-and-tighten costs $O(m \log n)$ time, which is a lot smaller than the time to compute even one min mean cycle.

Now consider the piecewise linear case from section 6. If we want to use cancel-and-tighten for circulations in networks in the piecewise linear case and we want to achieve the strongly polynomial bound of row 3 in Table 1, we are required to substitute a $O(mn)$ [14] min mean cycle computation once every $O(n \log n)$ tighten steps. Since $O(n \log)$ tighten steps cost $O(mn \log^2 n) > O(mn)$ time, this does not affect our cancel-and-tighten time bound. This establishes column c of Table 1.

8.2. Network cocirculation algorithms. We now consider how to implement cancel-and-tighten for cocirculations. Make an auxiliary graph containing the arcs in P^{far} and the reverses of the arcs in N^{far}, with the ε_e from (22) as lengths, together with both the forward and reverses of arcs close to h with length 0. Then a far cycle is a cut whose forward arcs all have positive lengths. We then adapt a trick from [2] for the classic linear cocirculation problem. Select an arbitrary node r as a source and compute shortest-path distances from r to every other node. Update x by these shortest-path distances, which effectively cancels all far cycles (cuts) with r on the source side. Then do the same thing with all arcs reversed to cancel far cycles with r on the sink side. This takes $O(m + n \log n)$ time.

Since no cut with all forward arcs positive remains after the cancel step, we can find a set of circuits such that each arc in E^{far} gets covered at least once in the proper orientation, and such a circuit cover can be computed in linear time [2]. The sum of the incidence vectors of these circuits is a sufficiently improving direction d for the tighten step. Thus one round of cancel-and-tighten costs $O(m + n \log n)$ time, which is a lot smaller than the time to compute even one min mean cut.

Now consider the piecewise linear case from section 6. If we want to use cancel-and-tighten for cocirculations in networks in the piecewise linear case and we want to achieve the strongly polynomial bound of row 3 in Table 1, we are required to substitute a $O(nMC(n, m))$ min mean cut computation once every $O(m \log n)$ tighten steps [27]. However, $O(m \log n)$ tighten steps cost $O(m \log n(m+n \log n)) < O(nMC(n, m))$ time, which would affect our cancel-and-tighten time bound. However, an idea of Radzik quoted in [2] allows the same strongly polynomial bound to hold even with a weaker *semiexact* min mean cut computation that costs only $O(MC(n, m))$ time. This can now be amortized without loss of efficiency; see [2] for details.

9. How to compute minimum mean cycles. We now explain how to solve the auxiliary problem (AP) in strongly polynomial time for an arbitrary unimodular linear space L. There exist general methods for computing min mean values under various assumptions; see Karzanov [15], Megiddo [21, 22], or Radzik [27]. None of the approaches in [21, 22, 27] seems to yield a strongly polynomial algorithm for (AP); we know of no algorithm for unimodular linear programming that is "linear" in the sense required by Megiddo's methods, and Radzik's method does not even apply to finding min mean circuits in networks.

We assume that L is explicitly given via a system $Mx = 0$ with a totally unimodular $n \times m$ matrix M whose columns are indexed by elements of a set E. We can get a strongly polynomial algorithm by directly solving the linear program (12) using the version of the ellipsoid algorithm developed by Tardos [32]. This depends on realizing that, although the LP is not totally unimodular, the size of its matrix in binary notation is $O(mn)$.

An algorithm that is more combinatorial is derived from the general method in [15] that enables us to reduce (AP) to solving $O(\rho\phi)$ linear programs with totally unimodular constraint matrices. This method will also allow us to solve (AP) for systems which are not totally unimodular.

Recall from the proof of Lemma 3.2 that (AP) is equivalent to solving problem (11). It is clear that (11) is, in turn, a special case of the following problem. Suppose that L' is a (not necessarily unimodular) linear subspace of $\mathbb{R}^{E'}$ given by a system $M'y = 0$ with an integer $n' \times m'$ matrix M' ($m' = |E'|$). Consider the cone $\mathcal{K} = L' \cap \mathbb{R}^{E'}_+ = \{y \in L' : y \geq 0\}$. Let $|y|$ denote the ℓ_1-norm $\sum_{e \in E'} |y_e|$, and for $y \in \mathcal{K} - \{0\}$, define the min mean value of y as $\nu_d(y) = dy/|y|$. Then, given a vector $d \in \mathbb{R}^{E'} - \{0\}$, the *minimum mean problem* (MMP) for $\mathcal{K}$ and d is to compute

$$\nu^* := \nu^*_d := \min_{y \in \mathcal{K} - \{0\}} \nu_d(y)$$

as well as a minimizing y.

An integer vector $y \in \mathcal{K} - \{0\}$ is called *elementary*, or a *cycle*, if $y = y' + y''$ is impossible for any two integral vectors $y', y'' \in \mathcal{K} - \{0\}$. We say that the maximum $q = q(\mathcal{K})$ of y_e's among all elementary vectors y and $e \in E'$ is the *fractionality* of $\mathcal{K}$. (Clearly, q does not exceed the maximum absolute value of a subdeterminant of M'.) Problem (AP) for L with $q > 1$ will be used in the next section. Clearly, there is an optimal solution to the MMP which is an elementary vector. Computationally, given any optimal solution y^* to the MMP, standard linear algebra techniques can extract an optimal cycle from y^*.

For $y \in \mathbb{R}^{E'}$, let $S(y) = \mathrm{supp}(y)$ and let $S^+(y)$, $S^-(y)$, and $S^0(y)$ denote the sets of $e \in E'$ with y_e positive, negative, and zero, respectively. We assume that every $e \in E'$ is contained in the support of at least one cycle in $\mathcal{K}$; otherwise, we could delete e from E'. Each iteration of the algorithm for the MMP transforms the current objective vector d by adding to it a vector $h \in (L')^\perp$ and possibly by shifting d by a constant $\delta \in \mathbb{R}$ (i.e., setting $d_e := d_e + \delta$ for all $e \in E'$). Clearly, the former operation does not change $\nu_d(y)$ for any $y \in \mathcal{K}$, while the latter one increases it by δ (and therefore increases ν^* to $\nu^* + \delta$). At the beginning of the algorithm, we shift the initial d by a sufficiently large negative r in order to ensure that the inequality $\nu^*_d \leq 0$ holds. We maintain $\nu^*_d \leq 0$ throughout the algorithm.

Recall that d^+ denotes the positive part of the current d. At an iteration, an element $u \in E'$ with $d_u < 0$ is fixed, and we solve the following pair of dual linear

programs:

$$\min\{d^+y \,:\, y \in \mathcal{K},\ y_u = 1\} \tag{27}$$

and

$$\max\{h_u \,:\, h \in (L')^\perp,\ \ h_e \geq -d^+_e \ \text{ for } \ e \in E' - \{u\}\}. \tag{28}$$

To see that (27) and (28) are dual LPs, rewrite (27) as $\min\{d^+y \,:\, M'y = 0,\ y \geq 0,\ y_u = 1\}$, let the dual vector for the constraints $M'y = 0$ be γ, and let the dual scalar for $y_u = 1$ be σ. Then if we define $h = -\gamma M'$, we must have $h \in (L')^\perp$ (since the rows of M' are a basis of $(L')^\perp$), and it is clear (since $d^+_u = 0$) that $h_u = \sigma$ at optimality in the dual of (27).

By the above assumption, there is a cycle whose support contains u; thus (27) has a feasible solution. Since the objective value of (27) is bounded below by zero, (27) has an optimal solution. When feasible solutions y and h to (27) and (28) are optimal, then the following (complementary slackness) condition holds:

$$\text{for any } \ e \in E' - \{u\}, \quad d^+_e + h_e > 0 \ \text{ implies } \ y_e = 0. \tag{29}$$

Let y and h be optimal, and define $\widehat{d} = d + h$. If $\widehat{d}_u \geq 0$, we finish the iteration with the new objective vector $d' = \widehat{d}$. If instead $\widehat{d}_u < 0$, we in addition shift $\widehat{d}$ by $\delta := -\widehat{d}y/|y|$, and the resulting vector d' becomes the new objective vector. In the latter case, we also keep y as a possible candidate for being an optimal solution of the original (and each intermediate) problem; set $y^* := y$. Then $d'y^* = 0$.

From (29), we observe that

$$\text{for } \ e \in S(y) - \{u\}, \quad \widehat{d}_e = \begin{cases} 0 & \text{if } d_e \geq 0, \\ d_e & \text{if } d_e < 0. \end{cases} \tag{30}$$

Since y is nonnegative, (30) implies

$$\widehat{d}y \leq \widehat{d}_u y_u = \widehat{d}_u; \tag{31}$$

hence $\delta > 0$ (in case $\widehat{d}_u < 0$). Note also that (30) together with $h_u = d^+y \geq 0$ (since $d^+ \geq 0$ and $y \geq 0$) and $\delta > 0$ preserves the following monotone property (in both cases):

$$S^-(d') \subseteq S^-(d); \qquad \text{and } d'_e \geq d_e \quad \text{for any } e \in S^-(d'). \tag{32}$$

The algorithm finishes if the iteration results in d' with $d'_e \geq 0$ for all $e \in E'$, i.e., $S^-(d') = \emptyset$. Then for the y^* obtained at the last iteration when a shift of d occurs, we have $d'y^* \geq 0$. On the other hand, after that iteration, the current d was transformed only by adding vectors from $L^\perp$, which does not change the inner product of d and y^*. Hence $d'y^* = 0$ (so $d'_e = 0$ for all $e \in S(y^*)$). This and $\nu^*_{d'} \geq 0$ (as $d' \geq 0$) yield that y^* is an optimal solution to the original MMP.

We say that the iteration is *big* if $S^-(d')$ is strictly contained in $S^-(d)$ and *small* otherwise. The number of big iterations does not exceed $N = |S^-(d^0)|$ for the d^0 obtained by the initial shift of the original objective vector. So we have to estimate the number of consecutive small iterations. We impose the additional rule that the same element u is fixed throughout consecutive small iterations. Suppose that the current and the next iterations are small. Let y', h', and d' be the corresponding

objects found at the next iteration. Note that by the definition of small iterations, $S^-(d) = S^-(d')$ so that $Q := Q(d) := S^+(d) \cup S^0(d)$ is independent of d during this sequence of small iterations.

LEMMA 9.1. $y(Q) > y'(Q)$.

Proof. Obviously, $d'_e \geq \delta$ for each $e \in Q$, and (30) implies that $d'_e = \delta$ for $e \in S(y) \cap Q$. This and $d'y = 0$ give

$$d'_u = d'_u y_u \geq \sum(d'_e y_e \,:\, e \in S^-(d)) = -\sum(d'_e y_e \,:\, e \in Q) = -\delta y(Q). \tag{33}$$

On the other hand, we have $\widehat{d}'_u = d'_u + h'_u < 0$ (since the second iteration is small), so

$$-d'_u > h'_u = (d')^+ y' \geq \delta y'(Q) \tag{34}$$

(using LP strong duality for the second iteration and the fact that $d'_e \geq \delta$ for $e \in Q$). Comparing (33) and (34) yields $\delta y(Q) > \delta y'(Q)$, and the result follows because $\delta > 0$. □

Note that the preliminary shift of the original d can be chosen so that the above number N is at most ρ (the rank of M'; since $M' = (M \quad -M)$ in our application, we use the same ρ and ϕ for both M and M'). Indeed, let $p_1 < p_2 < \cdots < p_k$ be all the different values of d_e, $e \in E'$, and let $X_i = \{e \in E \,:\, d_e \leq p_i\}$. Solving the corresponding LP problems, find the minimum i such that X_i contains the support of some nonzero vector of $\mathcal{K}$. Shift d by $-p_i$. This makes the elements of X_{i-1} $(X_i - X_{i-1})$ be of negative (resp. zero) value, and therefore there is $y \in \mathcal{K} - \{0\}$ with $\nu_d(y) \leq 0$. Clearly, we have $N = |X_{i-1}| \leq \rho$.

THEOREM 9.2. *The number of iterations of the algorithm is at most* $\rho \min\{q^2\phi, \eta\}$, *where* η *is the number of elementary vectors in* $\mathcal{K}$. *In particular, if* q *is bounded by a polynomial in* $|E'|$ *and a strongly polynomial algorithm is used to solve* (27) *and* (28), *then this algorithm is strongly polynomial for the MMP.*

Proof. Consider two consecutive small iterations as in Lemma 9.1. As we noted, there must be extreme optimal solutions y and y' to (27) in these iterations. By Lemma 9.1, $y(Q) > y'(Q)$, so the number of consecutive small iterations does not exceed the number of distinct values of $z(Q)$ for extreme vectors z of $\mathcal{K}$ with $z_u = 1$. But if z' is an integral extreme vector of $\mathcal{K}$, then $z'(Q)$ is an integer between 1 and $q\phi$, and there are at most q ways to normalize z' to z so that $z_u = 1$. Thus there can be at most $\min\{q^2\phi, \eta\}$ values for $x(Q)$. Now the result follows from the fact that the number of big iterations is at most ρ. □

For the original function d and the final function d^*, we have $d^* = d + h^* + \delta^* \mathbb{1}_{E'}$, where δ^* is the total shift and h^* is the sum of solutions to (28) over the iterations. Since d^* is nonnegative and $d^*_e = 0$ for any $e \in S(y^*)$,

$$d_e + h^*_e \begin{cases} = -\delta^* & \text{for } e \in S(y^*), \\ \geq -\delta^* & \text{otherwise.} \end{cases} \tag{35}$$

In particular, $\nu^*_d = -\delta^*$. (Note also that (35) proves the equality $\nu^*_d = \max_{h \in (L')^\perp} \min_{e \in E'}(d_e + h_e)$.)

10. The methods for linear spaces of bigger fractionality. We now generalize the method developed in section 3 to linear spaces $L \subseteq \mathbb{R}^E$ given by system $Mx = 0$ when M is not totally unimodular. We briefly describe how to do this, emphasizing only the places where significant differences exist between this case and section 3.

The cost at x of a cycle ξ is defined as

$$c(x,\xi) = \sum\{c_e^{\vdash}(x_e)\xi_e \ : \ e \in S^+(\xi)\} + \sum\{c_e^{\dashv}(x_e)\xi_e \ : \ e \in S^-(\xi)\},$$

and its mean cost $\overline{c}(x,\xi)$ is $c(x,\xi)/|\xi|$ ($S^+(\xi)$ and $S^-(\xi)$ are as defined in the previous section). Then ξ is a minimum mean cycle at x if $\overline{c}(x,\xi)$ is as small as possible, and $\lambda(x)$ is defined to be $\max\{0, -\overline{c}(x,\xi)\}$, where ξ is a minimum mean cycle.

Lemmas 3.1 and 3.2 remain true in this case. MMCM is generalized as follows. For the current $x \in L$, we solve the analogue of the auxiliary problem (AP) to compute $\lambda = \lambda(x)$ and, if $\lambda > 0$, to find a minimum mean cycle ξ of L at x and $h \in L^{\perp}$ satisfying (10). If $\lambda > 0$, we "push $\varepsilon > 0$ around ξ," i.e. transform x into the new current point $x' \in L$ defined as $x' = x + \varepsilon\xi$ (cf. (9)). To compute the step length ε, we first compute the local step lengths ε_e as an increment Δ such that (cf. (16))

$$\begin{cases} c_e^{\dashv}(x_e + \Delta\xi_e) \le c_e^{\vdash}(x_e) + \lambda = h_e \le c_e^{\vdash}(x_e + \Delta\xi_e) & \text{if } e \in S^+(\xi), \\ c_e^{\dashv}(x_e - \Delta\xi_e) \le c_e^{\dashv}(x_e) - \lambda = h_e \le c_e^{\vdash}(x_e - \Delta\xi_e) & \text{if } e \in S^-(\xi), \end{cases}$$

and then $\varepsilon = \min\{\varepsilon_e \ : \ e \in S^+(\xi) \cup S^-(\xi)\}$.

In this case, Lemma 3.3 has essentially the same proof, while Lemma 3.4 should be modified as follows.

LEMMA 10.1. *After every $\rho^{\perp}$ iterations, $\lambda = \lambda(x)$ decreases by a factor of at most $(1 - 1/(2\phi q))$.*

Proof. The proof is similar to that of Lemma 3.4. The only nontrivial point that needs some reproving is when, at some iteration j ($i < j \le i + \rho^{\perp}$), for the chosen cycle ξ^j, there is an element $u \in \mathrm{supp}(\xi^j)$ close to h, and for all $e \in S^+(\xi^j) - \{u\}$ (resp. $e \in S^-(\xi^j) - \{u\}$), we have $c_e^{\vdash} - h_e \ge -\lambda$ (resp. $h_e - c_e^{\dashv} \ge -\lambda$); for simplicity, we replace $c_e^{\vdash}(x_e^j)$ and $c_e^{\dashv}(x_e^j)$ by $c_e^{\vdash}$ and $c_e^{\dashv}$ here. For definiteness, let u be positively close to h. Then

$$\begin{aligned} c(x^j,\xi^j) &= c(x^j,\xi^j) - h\xi^j \\ &= \sum_{e \in S^+(\xi^j) - \{u\}} (c_e^{\vdash} - h_e)\xi_e^j + (c_e^{\vdash} - h_u)\xi_u^j - \sum_{e \in S^-(\xi^j)} (h_e - c_e^{\dashv})\xi_e^j \\ &\ge -\lambda \sum_{e \in S^+(\xi^j) - \{u\}} \xi_e^j \ - \ \frac{1}{2}\lambda\xi_u^j \ + \lambda \sum_{e \in S^-(\xi^j)} \xi_e^j \\ &= -\lambda|\xi^j| + \frac{1}{2}\lambda\xi_u^j \ge -\lambda|\xi^j| + \lambda/2, \end{aligned}$$

whence $-\lambda^j = \overline{c}(x,\xi^j) \ge -\lambda + \lambda/2|\xi^j| \ge -\lambda(1 - 1/2\phi q)$ (since $|\xi^j| \le \phi q$), and the result follows. □

This establishes column a of Table 1.

11. Duality and practical implementations. Throughout our methods, we have liberally used dual vectors $h \in L^{\perp}$, and we computed optimal dual vectors h^*. There is a well-developed theory of duality in convex optimization that shows that not only is h^* optimal in the sense of proving x^*'s optimality via (15), but h^* is the optimal solution of a related optimization problem dual to (4). The form of this problem is

$$\min_{h \in L^{\perp}} \sum_{e \in E} v_e(h_e), \tag{36}$$

where each v_e is the *convex conjugate* of the corresponding w_e (see, e.g., Rockafellar [30], who would say that we have a dual pair of monotropic programming problems).

Suppose that w_e is piecewise linear. Then v_e is also piecewise linear, the breakpoints of v_e are the slopes of w_e, and the slopes of v_e are the breakpoints of w_e. For example, the convex conjugate of the cost function for linear circulations in networks in Figure 1 is the function for linear cocirculations in networks in Figure 2. Similarly, w_e is piecewise mixed linear and quadratic if and only if v_e is piecewise mixed linear and quadratic.

Since we get h^* solving (36) for free when we solve the primal problem (4), if we want to compute x^*, we have the luxury of being free to either solve the primal problem (4) and get x^* directly or solve the dual problem (36) and get x^* as the optimal set of dual variables for (36). This free choice has some practical implications because (although the primal and dual problems are in theory completely symmetric) in practice it may be easier to solve one than the other.

We focus on solving circulation and cocirculation problems in networks. The number of MMCM iterations is $O(mn \log(\beta/\varepsilon))$ for both cases (MMCM boxes 1c and 1d in Table 1). However, the time per iteration (which is dominated by the min mean cycle computation) is, up to log factors, $O(mn)$ for circulations and $O(mn^2)$ for cocirculations using strongly polynomial algorithms, and $O(\sqrt{n}m)$ for circulations and $O(mn)$ for cocirculations using polynomial algorithms under the similarity assumption (namely, that $C = O(n^{O(1)})$; see [1]). Thus it is clearly better to use MMCM on the circulation version of the problem when we have a choice.

For the faster cancel-and-tighten method, the work per iteration is the same up to log factors: $O(m)$ for both circulations and cocirculations. However, the number of iterations is $O(n \log(\beta/\varepsilon))$ for circulations but $O(m \log(\beta/\varepsilon))$ for cocirculations (cancel-and-tighten boxes 1c and 1d of Table 1). In the piecewise linear case, the iteration bound for circulations is $O(\min\{n \log(nC), Bn \log n\})$, and for cocirculations it is $O(\min\{m \log(nC), Bm \log n\})$ (cancel-and-tighten boxes in rows 2 and 3, columns c and d of Table 1). Thus in both cases, solving the circulation version of the problem instead of the cocirculation version will result in a speedup by a factor of roughly $O(m/n)$.

Thus we conclude that when finding an optimal cocirculation in a network, it should be faster to dualize the problem and solve the circulation version of the problem instead. An experimental implementation of the cocirculation methods from [2] (a special case of our methods) in [19] proved to be very slow, confirming our recommendation.

12. Solving integer problems. For some applications it is more realistic to consider problems where the decision variables are restricted to be integers; see, e.g., [24] or [1, Chapter 14]. We show here how to adapt our methods to this restriction.

Formally, we want to solve the following problem:

(37) Find an integral $x \in L$ such that $w(x) = \sum_{e \in E} w_e(x_e)$ is as small as possible.

Define $\widehat{w}_e(x_e)$ so that it matches the value of $w_e(x_e)$ whenever x_e is integral and is linear between consecutive integers, so the $\widehat{w}_e$'s are piecewise linear. It is equivalent to solve (4) using the $\widehat{w}_e$'s and solve (37) with the w_e's. We would like to use the versions of our methods adapted for piecewise linear costs to solve (4) using the $\widehat{w}_e$'s. Unfortunately, the number of pieces in all of the $\widehat{w}_e$'s might be unbounded, so we cannot use the strongly polynomial bound. Also, the $\widehat{w}_e$'s have integral breakpoints

but general slopes, just the reverse of what Lemma 6.1 requires. However, we can (implicitly) dualize the problem to get a problem over the dual space $L^\perp$ with objectives $\widehat{v}_e$, the convex conjugates of the $\widehat{w}_e$ (see section 11). Recall from section 11 that the $\widehat{v}_e$'s are also piecewise linear and have integer slopes, so Lemma 6.1 *does* apply to the dual problem.

To make this work, we need to be able to simulate an oracle for $\widehat{v}_e$ from the given oracle for w_e. For (O) (i), we are given r and want to find $\widehat{c}_e^{\dashv}(r)$ and $\widehat{c}_e^{\vdash}(r)$, where the $\widehat{c}_e$'s are the left and right costs at r w.r.t. $\widehat{v}_e$. We use (O) (ii) for w_e to find a p such that $c_e^{\dashv}(p) \le r \le c_e^{\vdash}(p)$. Set $k = \lceil p \rceil$ and compute the slope of $\widehat{w}_e$ between $k-1$ and k as $w_e(k) - w_e(k-1)$. If $w_e(k) - w_e(k-1) \le r$, then $\widehat{c}_e^{\vdash}(r) = k$; otherwise, it equals $k-1$. Similarly, if $w_e(k) - w_e(k-1) > r$, then $\widehat{c}_e^{\dashv}(r) = k$; otherwise, it equals $k-1$.

For (O) (ii), we are given p and want to find r such that $\widehat{c}_e^{\dashv}(r) \le p \le \widehat{c}_e^{\vdash}(r)$. Since $\widehat{v}_e$ is piecewise linear, such an r always exists which is a breakpoint of $\widehat{v}_e$. Then the two slopes adjacent to this breakpoint r are $\widehat{c}_e^{\dashv}(r)$ and $\widehat{c}_e^{\vdash}(r)$. Redualizing back to the primal and remembering that dualizing interchanges slopes and breakpoints, we want to find a slope r of $\widehat{w}_e$ whose adjacent breakpoints $a = \widehat{c}_e^{\dashv}(r)$ and $b = \widehat{c}_e^{\vdash}(r)$ bracket p. But breakpoints of $\widehat{w}_e$ are just all of the integers, so $a = \lfloor p \rfloor$, $b = \lceil p \rceil$ (if p is an integer, set $b = p+1$ instead), and so r is the linear approximation of the slope between these breakpoints, namely $w_e(b) - w_e(a)$.

Note that duality reverses the roles of the two parts of the oracle: We use (O) (i) for w_e to simulate (O) (ii) for $\widehat{v}_e$, and we use (O) (ii) for w_e to simulate (O) (i) for $\widehat{v}_e$. This allows us to simulate an oracle for $\widehat{v}_e$ using only $O(1)$ calls to the oracle for w_e. Otherwise, we run our algorithm as before (using matrix K in place of matrix M). This yields the bounds in both row 6's of Table 1.

13. The methods for problems with approximate oracles. In the case where some w_e's are general convex functions, it may not be possible to exactly represent $c_e^{\vdash}$ and $c_e^{\dashv}$ in a finite word length, leading to an *evaluation error* δ_e. Similarly, an exact solution to (O) (ii) may be transcendental and/or it might not be easy to solve the equation $w_e'(r) = p$ exactly (which is roughly what (O) (ii) requires). Then (O) (ii) might be implemented as a numerical root-finding procedure with some known error, leading to a *root-finding error* δ_r. We now use the convention that symbols for approximate quantities are the same symbols as the exact quantities with tildes added. We now assume that we have an oracle that

($\tilde{\text{O}}$) (i) when given a "point" $r \in \mathbb{R}$ will return costs $\widetilde{c}_e^{\vdash}(r)$ and $\widetilde{c}_e^{\dashv}(r)$ satisfying $|\widetilde{c}_e^{\vdash}(r) - c_e^{\vdash}(r)| \le \delta_e$ and $|\widetilde{c}_e^{\dashv}(r) - c_e^{\dashv}(r)| \le \delta_e$;

(ii) when given a "scalar" $p \in \mathbb{R}$ will return a point $\widetilde{r}$ with $c_e^{\dashv}(\widetilde{r}) - \delta_r \le p \le c_e^{\vdash}(\widetilde{r}) + \delta_r$ (so $\widetilde{c}_e^{\dashv}(\widetilde{r}) - \delta_e - \delta_r \le p \le \widetilde{c}_e^{\vdash}(\widetilde{r}) + \delta_e + \delta_r$).

The first consequence of these errors is that $\widetilde{\lambda}(x)$, the approximate min mean cycle value that we compute w.r.t. $\widetilde{c}_e^{\vdash}$ and $\widetilde{c}_e^{\dashv}$, will not exactly equal $\lambda(x)$. However, since $\widetilde{c}_e^{\vdash}$ and $\widetilde{c}_e^{\dashv}$ differ from $c_e^{\vdash}$ and $c_e^{\dashv}$ by at most δ_e per element, it is easy to see that $\widetilde{\lambda}(x)$ satisfies

$$|\lambda(x) - \widetilde{\lambda}(x)| \le \delta_e. \tag{38}$$

A quick scan of Lemmas 3.3 and 3.4 (the key lemmas proving polynomial convergence) shows that our arguments there all have a slack of at least $\lambda/2$ in them, except for the argument in Claim 2 that no new element far from h can be created. We want to claim that whenever $\widetilde{\lambda}(x)$ is large enough w.r.t. δ_e and δ_r, the arguments in those lemmas will still hold true with the approximate data $\widetilde{c}_e^{\vdash}$, $\widetilde{c}_e^{\dashv}$, and $\widetilde{\lambda}(x)$ substituted

for $c_e^{\vdash}$, $c_e^{\dashv}$, and $\lambda(x)$. The argument in Claim 2 depends on knowing that for some e (superscripting by iteration number)

$$\begin{aligned} &\widetilde{c}_e^{\vdash}(x_e^j) + \widetilde{\lambda}^j \le \widetilde{c}_e^{\vdash}(x_e^{j+1}) \quad \text{if } e \in A^j \qquad \text{and} \\ &\widetilde{c}_e^{\dashv}(x_e^j) - \widetilde{\lambda}^j \ge \widetilde{c}_e^{\dashv}(x_e^{j+1}) \quad \text{if } e \in B^j, \end{aligned} \tag{39}$$

but ($\tilde{\text{O}}$) (ii) might cause these inequalities to be violated by as much as $\delta := \delta_e + \delta_r$. We can fix this and obtain an easy analysis of the approximate oracle case by slightly increasing our local step length ε_e from (16) to now be an increment Δ such that

$$\begin{cases} \widetilde{c}_e^{\dashv}(x_e + \Delta) - \delta \le \widetilde{c}_e^{\vdash}(x_e) + \widetilde{\lambda} + \delta \le \widetilde{c}_e^{\vdash}(x_e + \Delta) + \delta & \text{if } e \in A, \\ \widetilde{c}_e^{\dashv}(x_e - \Delta) - \delta \le \widetilde{c}_e^{\dashv}(x_e) - \widetilde{\lambda} - \delta \le \widetilde{c}_e^{\vdash}(x_e - \Delta) + \delta & \text{if } e \in B. \end{cases} \tag{40}$$

That is, we call ($\tilde{\text{O}}$) (ii) with $p = \widetilde{c}_e^{\vdash}(x_e) + \widetilde{\lambda} + \delta_e + \delta_r$ when $e \in A$ and $p = \widetilde{c}_e^{\dashv}(x_e) - \widetilde{\lambda} - \delta_e - \delta_r$ when $e \in B$. Since $x_e^{j+1} = x_e^j + \Delta$ for an e achieving the minimum ε_e, this ensures that (39) will hold for this e.

There are two factors that limit how much we can reduce $\widetilde{\lambda}(x)$ before the inaccuracies catch up with us: (i) the maximum error in a $c_e^{\vdash}$ or $c_e^{\dashv}$, namely δ_e, can be at most the slack in the arguments in Lemmas 3.3 and 3.4, namely $\widetilde{\lambda}(x)/2$; and (ii) the increment in reduced cost, namely $\widetilde{\lambda}(x) + \delta_e + \delta_r$, plus the maximum error in $c_e^{\vdash}$ and $c_e^{\dashv}$, namely δ_e, must be smaller than $\widetilde{\lambda} + \widetilde{\lambda}/2$ to ensure that we do not make a positively far element negatively far or vice versa. Now (i) gives the bound $\widetilde{\lambda}(x) > 2\delta_e$ and (ii) gives $\widetilde{\lambda}(x) \ge 2\delta_r + 2\delta_e$. That is, we must have

$$\widetilde{\lambda}(x) \ge 2\delta_e + 2\delta_r. \tag{41}$$

Now, as long as $\widetilde{\lambda}(x)$ satisfies (41), and since we contrived (40) to make (39) true, the whole analysis of Lemmas 3.3 and 3.4 becomes true with $\widetilde{c}_e^{\vdash}$, $\widetilde{c}_e^{\dashv}$, and $\widetilde{\lambda}(x)$ in place of $c_e^{\vdash}$, $c_e^{\dashv}$, and $\lambda(x)$. In particular, this shows that $\widetilde{\lambda}(x)$ will geometrically decrease. Once the algorithm has driven $\widetilde{\lambda}(x)$ down to $2\delta_e + 2\delta_r$, (38) says that $\lambda(x) \le 3\delta_e + 2\delta_r$. This estimate can be used with the cost distance defined in section 4 and the bounds based on it developed in section 7 to get a bound on how close x is to an optimal solution.

These arguments suggest that when ($\tilde{\text{O}}$) (ii) is implemented as a root-finding procedure with an error estimate available at each iteration, then it will be faster to run the root-finder only until the current error in its root, δ_r', is small enough so that $\widetilde{\lambda}(x) - 2\delta_r' > 2\delta_e$.

The same sort of analysis can be pushed through for the cancel-and-tighten method. All of this shows that even with an approximate oracle, our methods can obtain an answer whose accuracy is close to the accuracy of the underlying evaluation and root-finding machinery.

Acknowledgments. We thank the referees for suggesting improvements to this paper and Maurice Queyranne for pointing out an error in an earlier version.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice–Hall, New York, NY, 1993.

[2] T. R. Ervolina and S. T. McCormick, *Two strongly polynomial cut cancelling algorithms for minimum cost network flow*, Discrete Appl. Math, 46 (1993), pp. 133–165.
[3] M. Florian, *Nonlinear cost network models in transportation analysis*, Math. Prog. Stud., 26 (1986), pp. 167–196.
[4] A. Frank and A. V. Karzanov, *Determining the distance to the perfect matching polytope of a bipartite graph,* Technical Report RR 895-M, Laboratoire ARTEMIS IMAG, Université Joseph Fourier, Grenoble, France, 1992.
[5] A. V. Goldberg and R. E. Tarjan, *Finding minimum-cost circulations by canceling negative cycles*, J. Assoc. Comput. Mach., 36 (1989), pp. 873–886.
[6] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin, 1988.
[7] M. Hadjiat, *Un algorithme fortement polynomial pour la tension de coût minimum basé sur les cocycles de coûts moyens minimums*, Technical Report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, France, 1994 (in French).
[8] R. Hassin, *The minimum cost flow problem: A unifying approach to dual algorithms and a new tree-search algorithm*, Math. Programming, 25 (1983), pp. 228–239.
[9] R. Hassin, *A flow algorithm for network synchronization*, Working Paper, Department of Statistics and Operations Research, Tel-Aviv University, Tel-Aviv, 1993.
[10] D. S. Hochbaum, *Polynomial and strongly polynomial algorithms for convex network optimization*, in Network Optimization Problems, D. Z. Du and P. M. Pardalos, eds., World Scientific, Singapore, 1993, pp. 63–92.
[11] D. S. Hochbaum and J. G. Shanthikumar, *Convex separable optimization is not much harder than linear optimization*, J. Assoc. Comput. Mach., 37 (1990), pp. 843–862.
[12] T. Ibaraki, A. V. Karzanov, and H. Nagamochi, *Determining the distance to the flow polyhedron of a network*, private communication, 1993.
[13] K. Iwano, S. Misono, S. Tezuka, and S. Fujishige, *A new scaling algorithm for the maximum mean cut problem*, Algorithmica, 11 (1994), pp. 243–255.
[14] R. M. Karp, *A characterization of the minimum cycle mean in a digraph*, Discrete Math., 23 (1978), pp. 309–311.
[15] A. V. Karzanov, *Minimum mean weight cuts and cycles in directed graphs*, in Methods for Solving Operator Equations, Yaroslavl State University, Yaroslavl, Russia, 1985, pp. 72–83 (in Russian); Amer. Math. Soc. Transl. (2), 158 (1994), pp. 47–55 (in English).
[16] T. L. Magnanti, *Models and algorithms for predicting urban traffic equilibria*, in Transportation Planning Models, M. Florian, ed., North–Holland, Amsterdam, 1984, pp. 153–186.
[17] Y. Mansour, B. Schieber, and P. Tiwari, *Lower bounds for computations with the floor operation*, manuscript, 1988.
[18] S. T. McCormick, *Approximate binary search algorithms for mean cuts and cycles*, Oper. Res. Lett., 14 (1993), pp. 129–132.
[19] S. T. McCormick and L. Liu, *An experimental implementation of the dual cancel and tighten algorithm for minimum cost network flow*, in Network Flows and Matching, D. S. Johnson and C. S. McGeoch, eds., American Mathematical Society DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, AMS, Providence, RI, 1993, pp. 247–266.
[20] S. T. McCormick, T. R. Ervolina, and B. Zhou, *Mean cancelling algorithms for general linear programs, and why they (probably) don't work for submodular flow*, Working Paper 94-MSC-011, Faculty of Commerce, University of British Columbia, Vancouver, BC, Canada, 1994.
[21] N. Megiddo, *Combinatorial optimization with rational objective functions*, Math. Oper. Res., 4 (1979), pp. 414–424.
[22] N. Megiddo, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 30 (1983), pp. 852–865.
[23] M. Minoux, *A polynomial algorithm for minimum quadratic cost flow problems*, Europ. J. Oper. Res., 18 (1984), pp. 377–387.
[24] M. Minoux, *Solving integer minimum cost flows with separable convex cost objective polynomially*, Math. Prog. Stud., 26 (1986), pp. 237–239.
[25] A. S. Nemirovsky and D. B. Yudin, *Problem Complexity and Method Efficiency in Optimization*, Wiley–Interscience, Toronto, 1983.
[26] J. B. Orlin and R. K. Ahuja, *New scaling algorithms for the assignment and minimum cycle mean problems*, Math. Programming, 54 (1988), pp. 41–56.
[27] T. Radzik, *Parametric flows, weighted means of cuts, and fractional combinatorial optimization*, in Complexity in Numerical Optimization, P. Pardalos, ed., World Scientific, Singapore, 1993, pp. 351–386.

[28] T. Radzik and A. V. Goldberg, *Tight bounds on the number of minimum-mean cycle cancellations and related results*, Algorithmica, 11 (1994), pp. 226–242.
[29] R. T. Rockafellar, *Convex Analysis*, Princeton University Press, Princeton, NJ, 1970.
[30] R. T. Rockafellar, *Network Flows and Monotropic Optimization*, Wiley–Interscience, New York, 1984.
[31] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.
[32] É. Tardos, *A strongly polynomial algorithm to solve combinatorial linear programs*, Oper. Res., 34 (1986), pp. 250–256.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1277–1283, October 1997

ALGORITHMS FOR THE CERTIFIED WRITE-ALL PROBLEM*

RICHARD J. ANDERSON† AND HEATHER WOLL‡

Abstract. In this paper, we prove new upper bounds on the complexity of the certified write-all problem with respect to an adaptive adversary. Our strongest result is that for any $\epsilon > 0$, there exists an $O(p^{1+\epsilon})$ work algorithm for the p-processor p-memory cell write-all. We also give a randomized $O(p^2 \log p)$ work algorithm for a p-processor p^2-memory cell write-all.

Key words. certified write-all problem, asynchronous computation, synchronization primitives, adversary models

AMS subject classifications. 68Q10, 68Q22

PII. S0097539794319126

1. Introduction.

A fundamental problem in asynchronous computation is to have p processors set n memory cells to a fixed value, and then signal the completion of the task. This problem is important in its own right as well as for capturing the difficulty of other problems in asynchronous computation such as the step-by-step simulation of a synchronous computation on an asynchronous machine.

Formally, the input to the certified write-all problem is an array $B[1. . n]$ and a variable c. The variables are all initialized to 0. The problem is to set all the variables to 1, with the restriction that c cannot be set to 1 until all the $B[i]$'s are guaranteed to be 1. We have p processors available. We assume a fully asynchronous model, where processors read and write to a global memory.

Each processor issues a stream of read and write instructions. A computation is an interleaving of these instructions. We are interested in worst case asynchronous computation, so we consider the interleaving which maximizes the cost of the computation. A common way to view this is to assume that an all powerful (or adaptive) adversary constructs the interleaving sequence.

We use the *work* measure as the cost of the algorithm. The work of the algorithm is the total number of instructions executed up until the point that the problem is solved.

The write-all problem was introduced by Kanellakis and Shvartsman [KS92] who gave an $O(p \log^2 p)$ work upper bound for a p-processor, p-cell write-all algorithm under the fail-stop model. Martel, Subramonian, and Park [MSP90] gave an $O(p)$ work bound for an $\frac{p}{\log p \log^* p}$-processor, p-cell write-all algorithm under a weaker adversary model than considered here. Martel's algorithm was randomized and assumed that the adversary must set the instruction interleaving prior to viewing the random numbers. The write-all problem for the general (adaptive) adversary model was previously considered by Buss et al. [BKRS96]. They gave an algorithm with $O(p^{\log_2 3})$ work ($\log_2 3 \approx 1.59$). Our algorithm can be viewed as an extension of their method. They also established an $\Omega(p \log p)$ work lower bound for the write-all problem.

* Received by the editors December 9, 1994; accepted for publication (in revised form) May 19, 1995.

http://www.siam.org/journals/sicomp/26-5/31912.html

† Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350 (anderson@cs.washington.edu). This work was partially supported by NSF grant CCR-9204242 and NSF II grant CDA-9123308.

‡ 92 Seymour Terrace, Piscataway, NJ 08854.

2. Block write algorithm. In the write-all problem, we must write to n memory cells using p processors. The difficulty is that an adversary controls the order in which the processors execute. The adversary can cause all processors to halt except for one. This means that a processor must write to all memory cells when the other processors appear to have stopped. To avoid using $\Omega(np)$ work, processors must detect when other processors have written to regions of memory.

One method for reducing the work is to have the processors perform *block writes*. The memory is divided into blocks $B_1, \ldots, B_k$, each of size b. Each block B_i has an associated completion bit b_i. A block write to B_i is done by first reading b_i, and if b_i is 0, writing to the memory cells of B_i and then setting b_i to 1. In other words, B_i is written to only if the completion bit is not set when the write begins. If the processors perform block writes in the same order, then the adversary can cause $\Omega(pn)$ work by having the processors execute in lock step. However, if the processors perform block writes in different orders, then the worst case work can be reduced. For example, suppose that there are two processors P_1 and P_2 and two blocks B_1 and B_2. P_1 does block writes in the order B_1, B_2, and P_2 does block writes in the order B_2, B_1. Either $b_2 = 1$ when P_1 reaches B_2, or $b_1 = 1$ when P_2 reach B_1. This means that the amount of work is at most $\frac{3}{2}n + 7$ instead of $2n$. (The "+7" comes from reading and writing completion bits.) All of our algorithms are based upon using this idea to limit the amount of duplicated work.

The block write algorithm has each processor perform block writes to all cells. The order of the block writes is governed by a different permutation for each processor. A processor with permutation $\pi = \pi_1, \ldots, \pi_k$ performs the block writes in the order $B_{\pi_1}, \ldots, B_{\pi_k}$. We discuss how to analyze the performance of this algorithm and how to pick the permutations to use. We also generalize the algorithm to work with hierarchies of blocks to get different performance bounds.

The adversary chooses an interleaving of the instruction streams to maximize the amount of work. We can summarize this order by looking at the order in which the completion bits are set to 1. For a fixed interleaving of instructions, there is a permutation $\alpha = \alpha_1, \ldots, \alpha_k$ such that the assignments to the completion bits occur in the order $b_{\alpha_1}, \ldots, b_{\alpha_k}$.

A processor *succeeds* on a block write if the completion bit is zero when the write starts, so that the processor writes to the cells of the block. We want to be able to compute the maximum number of successful block writes for a processor, given the processor's permutation π and the adversaries interleaving α. A processor maximizes its number of successful block writes by having the writes occur as early as possible in the interleaving sequence subject to the constraint that they are consistent with α. For example, suppose $\pi = (4, 3, 1, 5, 2, 6)$ and $\alpha = (3, 4, 6, 1, 2, 5)$. The processor can succeed in its writes to blocks B_4, B_1, and B_5. In general, a processor can succeed in its write to block B_{π_i}, if the blocks $B_{\pi_1}, \ldots, B_{\pi_{i-1}}$ are completed before B_{π_i} according the to permutation α. This means that π_i occurs in α after $\pi_1, \ldots, \pi_{i-1}$. The permutation α^{-1} gives the position of i in the permutation α, so the permutation $\alpha^{-1}\pi$ gives the position of π_i in α.[1] In the example, $\alpha^{-1} = (4, 5, 1, 2, 6, 3)$ and $\alpha^{-1}\pi = (2, 1, 4, 6, 5, 3)$. If π_i occurs after $\pi_1, \ldots, \pi_{i-1}$, this means that $(\alpha^{-1}\pi)_i > (\alpha^{-1}\pi)_1, \ldots, (\alpha^{-1}\pi)_i > (\alpha^{-1}\pi)_{i-1}$, so i is a *left-to-right maxima* of $\alpha^{-1}\pi$. This result is summarized in the following lemma.

LEMMA 2.1. *Suppose that a processor performs its block writes in the order* $B_{\pi_1}, \ldots, B_{\pi_k}$ *and the adversary causes the completion bits to be written in the order*

[1] For permutations A and B, the composition permutation AB is defined $(AB)_i = A_{B_i}$.

$b_{\alpha_1}, \ldots, b_{\alpha_k}$. *The maximum number of successful block writes for the processor is bounded by the number of left-to-right maxima in* $\alpha^{-1}\pi$. □

We shall now turn our attention to bounding the number of left-to-right maxima in a set of permutations before returning to the write-all algorithms.

3. Contention of permutations. We define the *contention* of a set of permutations in order to quantify the worst case work that an adversary can cause. This allows us to express our bounds in terms of a simple combinatorial quantity. In this section, we assume that permutations are over the set $\{1, \ldots, n\}$ unless we say otherwise. A *random permutation* is a permutation which has been chosen uniformly from the set of all permutations of $\{1, \ldots, n\}$, and a *set of random permutations* is a set of independently chosen random permutations. The harmonic series is $H_n = \sum_{j=1}^{n} \frac{1}{j}$. This quantity occurs in our bounds on contention of permutations. The value of H_n is very close to the natural logarithm, satisfying the bound $\ln n \leq H_n \leq \ln n + 1$.

Let $\pi = \pi_1, \ldots, \pi_n$ be a permutation. We say π_i is a *left-to-right maxima* if $\pi_i > \pi_k$ for $k < i$. It is well known that the expected number of left-to-right maxima in a random permutation is H_n [Knu73]. We need a bound on the expected number of left-to-right maxima in a set of n random permutations.

We use $LR(\pi)$ to denote the number of left-to-right maxima in a permutation π and $LR(S)$ to denote the total number of left-to-right maxima in a set S of permutations, so $LR(S) = \sum_{\pi \in S} LR(\pi)$.

We begin with a lemma which can be used to analyze the distribution of $LR(\pi)$ when π is a random permutation.

LEMMA 3.1. *Suppose* $\pi = \pi_1, \ldots, \pi_n$ *is a random permutation. For* $1 \leq i \leq n$, *let* $X_i = 1$ *if* π_i *is a left-to-right maxima and* $X_i = 0$ *otherwise. The random variables* $X_1, \ldots, X_n$ *are independent random variables with* $\text{Prob}[X_i = 1] = \frac{1}{i}$.

Proof. The proof is by induction. The key to the proof is to generate random permutations by an algorithm which converts a random permutation on $\{1, \ldots, n-1\}$ into a random permutation on $\{1, \ldots, n\}$ without changing the values of $X_1, \ldots, X_{n-1}$.

Our method for generating a random permutation is as follows:

1. Construct a random permutation $\pi' = \pi'_1, \ldots, \pi'_{n-1}$ on $\{1, \ldots, n-1\}$.
2. Choose a random value r from $\{1, \ldots, n\}$. Let $\pi_n = r$. For $i < n$, let $\pi_i = \pi'_i$ if $\pi'_i < r$ and $\pi_i = \pi'_i + 1$ if $\pi'_i \geq r$.

π_n is a left-to-right maxima if and only if $r = n$, so $\text{Prob}[X_n = 1] = \frac{1}{n}$. This holds for any permutation π', so there is no conditioning between the $X_1, \ldots, X_{n-1}$ and X_n, so the random variables are independent. □

It follows from Lemma 3.1 that if S is a set of n random permutations, then the expected size of $LR(S)$ is nH_n. We need to show that it is very unlikely for $LR(S)$ to be larger than cnH_n for some c. Our proof relies on a "Chernoff bound" due to Raghavan. We restate his result here for completeness.

LEMMA 3.2 (Raghavan). *Let* $X_1, X_2, \ldots, X_r$ *be a sequence of independent Bernoulli trials and let* $\Psi = X_1 + X_2 + \cdots + X_r$. *Suppose* $\text{Exp}[\Psi] = m$. *For* $\delta > 0$,

$$\text{Prob}[\Psi > (1+\delta)m] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{m}.$$

Proof. See [Rag88, Theorem 1]. □

LEMMA 3.3. *Let* S *be a set of* n *random permutations. The probability that* $LR(S) > 3nH_n$ *is at most* $\frac{1}{2^n n!}$.

Proof. Let $S = \{\pi^1, \ldots, \pi^n\}$ be a set of random permutations. If $n \leq 3$ the result is trivial, so we assume $n \geq 4$. Let X_{ij} be a random variable that is one if π^i_j is a left-to-right maximum in π^i and zero otherwise. The key observation for the proof is that this is a set of independent random variables. It is clear that random variables which correspond to different permutations are independent, and Lemma 3.1 shows that random variables associated with the same permutation are also independent.

Let $\Psi = \sum_i \sum_j X_{ij}$. Since $\text{Prob}[X_{ij} = 1] = \frac{1}{p-j+1}$, we have $\text{Exp}[\Psi] = nH_n$. Applying Lemma 3.2 with $\delta = 2$

$$\text{Prob}[\Psi > 3n \ln n] < \left(\frac{e^2}{3^3}\right)^{nH_n} < \left(\frac{1}{e}\right)^{nH_n} \leq \left(\frac{1}{e}\right)^{n \ln n} = \frac{1}{n^n} < \frac{1}{2^n n!}.$$

(We use the assumption that $n \geq 4$ in the last inequality.) □

For permutations π and α, the contention of π with respect to α (denoted $\text{Cont}(\pi, \alpha)$) is defined to be $LR(\alpha^{-1}\pi)$. For a set S of permutations and a permutation α, the contention of S with respect to α is defined $\text{Cont}(S, \alpha) = \sum_{\pi \in S} \text{Cont}(\pi, \alpha)$. The maximum contention of S is $\text{Cont}(S) = \max_\alpha \text{Cont}(S, \alpha)$. We will show that there exists a set S of n permutations with $\text{Cont}(S) \leq 3nH_n$. First, we show that Lemma 3.3 applies, and then we prove the theorem using a probabilistic argument.

LEMMA 3.4. *Let α be a fixed permutation and S a set of n random permutations. The probability that* $\text{Cont}(S, \alpha) > 3nH_n$ *is at most* $\frac{1}{2^n n!}$.

Proof. The set of permutations $S' = \{\alpha^{-1}\pi \mid \pi \in S\}$ is a random set of permutations since multiplication by a fixed permutation can be viewed as a one-to-one mapping. Lemma 3.3 applies directly to give the result. □

We now prove that there exists a set of permutations with low contention. Our proof is based on the probabilistic method pioneered by Erdös [ES74], where we show the existence of a set with a certain property by showing that a random set has the property with probability greater than zero.

THEOREM 3.5. *Let S be a random set of n permutations. The probability that* $\text{Cont}(S) > 3nH_n$ *is at most* $\frac{1}{2^n}$.

Proof. We say that a set S of n permutations is *bad* for a permutation α if $\text{Cont}(S, \alpha) > 3nH_n$. By Lemma 3.4 the probability that a random set of n permutations is bad for a fixed α is less than $\frac{1}{2^n n!}$. Summing over all permutations α shows that with probability less than $\frac{1}{2^n}$, a random set of n permutations is bad for some α. □

COROLLARY 3.6. *There exists a set of S permutations with* $\text{Cont}(S) \leq 3nH_n$.

4. Randomized algorithms for the write-all problem. We use the results of the previous sections to derive several write-all algorithms. The algorithms differ in their ratios of processors to memory locations. The algorithms perform block writes in an order given by a random set of permutations. The following lemma relates the amount of work to the contention.

LEMMA 4.1. *Let $S = \pi^1, \ldots, \pi^p$ be a set of permutations on $1, \ldots, p$. Suppose that processor j performs its block writes in the order $B_{\pi^j_1}, \ldots, B_{\pi^j_p}$. The number of successful block writes is at most* $\text{Cont}(S)$.

Proof. If the adversary fixes the order of writes to $b_1, \ldots, b_p$ to the permutation α, then Lemma 2.1 says that the number of successful block writes for processor j is at most $\text{Cont}(\pi^j, \alpha)$. If we sum over all processors, the number of successful block writes for interleaving α is at most $\text{Cont}(S, \alpha)$. Hence, $\text{Cont}(S)$ is an upper bound on the number of successful block writes. □

Our first algorithm is for the case $n = p^2$, so we want to write to p^2 memory locations using p processors. In this case, we use the block write algorithm described in section 2. The memory is divided into p blocks of size p. A set $S = \{\pi^1, \ldots, \pi^p\}$ of random permutations on $\{1, \ldots, p\}$ is constructed. The code for processor j follows.

```
BlockWriteI
  for i := 1 to p
    k := π_i^j;
    if b_k = 0
      write B_k;
      b_k := 1;
```

By Lemma 4.1 the number of successful block writes is bounded by $3pH_p$ with probability $1 - \frac{1}{2^p}$. Each processor does $O(p)$ additional work in testing completion bits, so the total work is $O(p^2 \log p)$ with high probability.

To improve the result, we would like to decrease the number of memory cells per processor yet still maintain efficiency. We give another randomized algorithm which reduces the block size to $\sqrt{p}$. We do this by generalizing the *BlockWrite*I algorithm to use a two-level hierarchy of blocks. The *big* blocks are $B_1, \ldots, B_{\sqrt{p}}$ with completion bits $b_1, \ldots, b_{\sqrt{p}}$. Block B_i is divided into *small* blocks $B_{i1}, \ldots, B_{i\sqrt{p}}$ with completion bits $b_{i1}, \ldots, b_{i\sqrt{p}}$. The big blocks have size p and the small blocks have size $\sqrt{p}$. The processors are numbered P_{ij} for $1 \le i, j \le \sqrt{p}$. We uses a set of $\sqrt{p}$ random permutation $S = \pi^1, \ldots, \pi^{\sqrt{p}}$ over $\{1, \ldots, \sqrt{p}\}$.

The code for processor P_{ij} follows.

```
BlockWriteII
  for k := 1 to √p
    s := π_k^i;
    if b_s = 0
      for l := 1 to √p
        t := π_l^j;
        if b_st = 0
          write B_st;
          b_st := 1;
      b_s := 1;
```

LEMMA 4.2. *For a random choice of permutations, the algorithm BlockWrite*II *has* $O(p^{3/2} \log^2 p)$ *work with probability at least* $1 - \frac{1}{2^{\sqrt{p}}}$.

Proof. We divide the processors into groups $G_i = \{P_{i1}, \ldots, \mathcal{P}_{i\sqrt{p}}\}$. A group G_i *succeeds* in a block write to B_k if at least one processor in the group succeeds. By considering permutations of writes to $b_1, \ldots, b_{\sqrt{p}}$, Lemma 4.1 implies that the number of successful group writes is at most $\mathrm{Cont}(S)$.

When group G_i has a successful write to B_k, we make the pessimistic assumption that all processors in the group succeed. Lemma 4.1 now implies that the number of successful writes by processors in G_i to the small blocks in $B_{k1}, \ldots, B_{k\sqrt{p}}$ is at most $\mathrm{Cont}(S)$, and the associated work is $O(\sqrt{p}\mathrm{Cont}(S) + p)$. Putting these two together, we get a work bound of $\sqrt{p}\mathrm{Cont}^2(S) + p\mathrm{Cont}(S)$. Theorem 3.5 implies that this is $O(p^{3/2} \log^2)$ with probability at least $1 - \frac{1}{2^{\sqrt{p}}}$. □

5. Deterministic algorithm for the write-all problem. We can adapt the ideas used in the randomized algorithm to get a deterministic algorithm. The only use of randomness is that we do not know how to construct a set of permutations with low contention, so we rely on using a random set, which has low contention with

overwhelming probability. In our deterministic algorithm we increase the number of levels of recursion. As the number of levels of recursion increases, the size of the permutations decreases. When we reduce the size of the permutations used to a constant q, we claim that we can find a good set in constant time by a brute force search. The brute force algorithm tests every set of q permutations by evaluating the set's contention with respect to all permutations. This takes $O(\binom{q!}{q} q! q^2 \log q)$ time, which is a constant.

We show that for any $\epsilon > 0$ there exists a deterministic p processor algorithm for writing to p-memory cells that takes $O(p^{1+\epsilon})$ work. Suppose $p = q^d$ for integers q and d. We view the computation as taking place on a q-ary tree of height d. Each internal node of the tree contains a single bit, which corresponds to the completion bit of a block, and the leaf nodes contain the memory cells. We use a set $S = \{\pi^1, \ldots, \pi^q\}$ of permutations over $\{1, \ldots, q\}$ with $\text{Cont}(S) \leq 3qH_q$. The existence of such a set is guaranteed by Corollary 3.6. When a processor is at a node, it visits the children of that node in an order given by a permutation of S. We label the processors with distinct strings of length d over the alphabet $\{1, \ldots, q\}$. Each processor chooses a permutation to use for all of the nodes at a particular level of the tree. The choice of permutation is based upon the label of the processor. The processor labeled $q_1 \cdots q_d$ considers the children of a node on level l in the order given by π^{q_l}.

We give the code for the recursive algorithm *BlockWrite*III. The blocks are labeled B^l_α where α is a string of length l over $\{1, \ldots, q\}$, and the completion bits are labeled b^l_α. The algorithm assigns 1's to all of the blocks B^d_α. We give the code for the processor $q_1 \cdots q_d$. The initial call is *BlockWrite*III$(0, \lambda)$.

```
BlockWriteIII(l, α)
  if l = d then
    B^l_α := 1
  else
    for i := 1 to q do
      t := π^{q_l}_i;
      if b^{l+1}_α = 0
        BlockWriteIII(l + 1, αt);
        b^{l+1}_{αt} := 1;
```

Since each processor will write to all of the cells in the absence of other processors, it is easy to see that the algorithm solves the write-all problem. We now establish the work bound. We divide the processors into groups, with the groups indexed by strings over $\{1, \ldots, q\}$. For β, a string of length l, the group G^l_β is the set of all processors whose name has as a prefix the string β. We say that G^l_β has a *successful* write to B^l_α if some processor in G^l_β makes a call to *BlockWrite*III(l, α). The next lemma is the key to the performance analysis.

LEMMA 5.1. *If G^l_β has a successful write to B^l_α, then there are at most* $\text{Cont}(S)$ *pairs s and t where $G^{l+1}_{\beta s}$ has a successful write to $B^{l+1}_{\alpha t}$.*

Proof. Suppose that G^l_β has a successful write to B^l_α. Assume that every processor in G^l_β is successful in writing to B^l_α. A processor in $G^{l+1}_{\beta s}$ reads and writes $b^{l+1}_{\alpha 1}, \ldots, b^{l+1}_{\alpha q}$ in the order given by π^s. This is precisely the situation covered in Lemma 4.1, so the number of pairs of s and t where $G^{l+1}_{\beta s}$ has a successful write to $B^{l+1}_{\alpha t}$ is at most $\text{Cont}(S)$. ∎

We can now prove the main theorem. We refer to a successful write of G^l_β to B^l_α as a write on level l.

THEOREM 5.2. *For every $\epsilon > 0$, there exists a deterministic $O(p^{1+\epsilon})$ work algorithm for the p-processor, p-memory cell certified write-all.*

Proof. It follows Lemma 5.1 that the number of successful writes to level l is at most $(\mathrm{Cont}(S))^l$. Each successful write on level l can be performed by $\frac{p}{q^l}$ processors, so the work W is bounded by

$$W \le c \sum_{l=0}^{\log_q p} \frac{p}{q^l} (\mathrm{Cont}(S))^l \le cp \sum_{l=0}^{\log_q p} (3H_q)^l < cp(c' \log q)^{\log_q p}$$

for some constants c and c'. Manipulating logarithms we have

$$\log[(c' \log q)^{\log_q p}] = \log_q p \log\log q^{c'} = \log p \log\log q^{c'} / \log q.$$

So

$$W \le cp^{1+\log\log q^{c'}/\log q}.$$

Since $\log\log q^c / \log q \to 0$, for any $\epsilon > 0$ we can find a q such that $W \le p^{1+\epsilon}$. ∎

REFERENCES

[BKRS96] J. F. BUSS, P. C. KANELLAKIS, P. L. RAGDE, AND A. A. SHVARTSMAN, *Parallel algorithms with processor failures and delays*, J. Algorithms, 20 (1996), pp. 45–86.

[ES74] P. ERDÖS AND J. SPENCER, *The Probabilistic Method in Combinatorics*, Academic Press, New York, 1974.

[Knu73] D. E. KNUTH, *Searching and sorting*, in The Art of Computer Programming, Vol. 3, Addison–Wesley, Reading, MA, 1973.

[KS92] P. KANELLAKIS AND A. SHVARTSMAN, *Efficient parallel algorithms can be made robust*, Distrib. Comput., 5 (1992), pp. 202–217.

[MSP90] C. MARTEL, R. SUBRAMONIAN, AND A. PARK, *Asynchronous PRAMs are (almost) as good as synchronous PRAMs*, in 31st Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 590–599.

[Rag88] P. RAGHAVAN, *Probabilistic construction of deterministic algorithms: Approximating packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130–143.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1284–1309, October 1997

002

COMPUTATIONAL MODELING FOR GENETIC SPLICING SYSTEMS*

SAM MYO KIM†

Abstract. A genetic splicing system involves DNA molecules mixed with enzymes and a ligase that allow the molecules to be cleaved and recombined to produce other molecules in addition to the original ones. Recently, using formal language theory, several researchers have investigated the string properties of DNA molecules that may potentially arise from the original set of molecules under the effect of the given restriction enzymes.

This paper introduces an algorithm which, given a splicing system whose initial set of strings is regular, constructs a finite state automaton that recognizes the set of DNA molecules spliced by the system. This algorithm solves the open problem of constructing such an automaton and shows a direct approach to the proof of regularity of spliced languages.

Key words. finite state automata, languages, genetic splicing, DNA molecules

AMS subject classification. 68Q05

PII. S0097539794263890

1. Introduction.

Genetic splicing is one of the most popular techniques in the field of genetic engineering. It involves restriction enzymes and DNA molecules. Restriction enzymes are endodeoxyribonucleases that recognize specific nucleotide sequences in double-stranded DNA and cleave both strands of the double helix. In molecular biology each double-stranded DNA molecule is represented in terms of paired symbols from four alphabet A, T, C, and G, which denote adenine, cytocine, guanine, and thymine, respectively. The pairs are A/T, T/A, G/C, and C/G. In [5], Head used formal language theory for the study of the potential effect of a set of restriction enzymes and a ligase that allow a set of DNA molecules to be cleaved and reassociated to produce further molecules. He introduced a new generative formalism called a splicing system as an abstract model for such a biological setting and analyzed the associated languages.

The effect of a restriction enzyme is to cut the molecule into two pieces at a specific pattern of the molecule which is defined by the enzyme involved. For example, consider the effect of enzyme *EcoRI* represented by the pair of strings

GAATTC
CTTAAG

on a small hypothetical DNA molecule represented by the pair of strings

....GCTACTAGAATTCGCGCTA....
....CGATGATCTTAAGCGCGAT.... .

The enzyme works on the molecule by finding a substring pair identical to the enzyme, indicated by the underlines. The enzyme cuts this molecule into two staggered pieces

....GCTACTAG AATTCGCGCTA....
....CGATGATCTTAA GCGCGAT.... .

* Received by the editors March 2, 1994; accepted for publication (in revised form) September 19, 1995. This research was partially supported by the Korea Science and Engineering Foundation and National Science Foundation grants CCR-9114725 and CDA-8805910 while the author was a professor at Rensselaer Polytechnic Institute.

http://www.siam.org/journals/sicomp/26-5/26389.html

† Department of Computer Engineering, Kyungpook National University, Taegu, Korea (kims@bh.kyungpook.ac.kr).

Notice that G/C on the left piece (in this notation G is from the top string of the piece and C from the bottom string) and C/G on the right are not involved in the cut. However, they are needed for finding the right pattern and the cut that takes place. Enzymes are classified into two groups depending on where the overhangs occur. In the above example, the left piece has the overhang on the lower half of the strand and the right piece on the top half.

There are enzymes which cut the other way. For example, enzyme *HhaI*

GCGC
CGCG

will cut the above molecule as follows.

....GCTACTAGAATTCGCG CTA....
....CGATGATCTTAAGC GCGAT.... .

The staggered ends of a DNA molecule recombine with others if they match. For example, if there exists the following piece cut from some other molecule

AATTACATT....
TGTAA....

then by combining with the first half of the fragments cut by *EcoRI*, the following DNA molecule can be formed.

....GCTACTAGAATTACATT....
....CGATGATCTTAATGTAA.... .

In [5], it was proved that if the initial set of DNA strings is finite, the strings generated by the system are strictly locally testable [5], and if the initial set is regular, the set of strings generated is also regular [1]. In [2] an algorithm was presented which, given a splicing system, constructs an automaton which recognizes the language of the splicing system. This algorithm works only for the class of permanent splicing systems. It has been an open problem to develop a more powerful algorithm that does not require the permanence property of splicing systems.

We solve this open problem by introducing an algorithm which uses the so-called monomialization technique that we have developed based on the concept of input memory span of [7]. Given a splicing system with its initial set of strings given in terms of the finite state transition graph of an automaton that recognizes the set, our algorithm constructs an automaton that recognizes the set of strings generated by the system. This paper also shows constructive proofs of the strict locality in [5] and the regularity in [1] of the spliced languages.

Section 2 describes a formal definition of genetic splicing systems which abstracts the above biological concept, investigates the language generated by a splicing system, and introduces lemmas that will be used in sections 4 and 5. Section 3 develops useful concepts and related lemmas concerning finite state transition graphs that will be used in sections 4 and 5 together with the ones developed in section 2. Section 4 introduces an algorithm which constructs an automaton for a given splicing system and analyzes its output. Section 5 proves the main theorems of the paper and, finally, section 6 gives some concluding remarks.

2. Splicing systems. This section formally defines splicing systems, which were first introduced in [5], and develops some lemmas that will be used in sections 4 and 5.

DEFINITION 2.1 (see [5]). *A* splicing system *is a quadruple* $S = (A, I, B, C)$, *where*

A: *a finite alphabet,*

I: *a set of initial strings in* A^*, *and*

B, C: finite sets of triples (u, x, v), $u, x, v \in A^*$.

Note that the size of I can be infinite. The sets B and C are called *left-hand patterns* and *right-hand patterns*, respectively. A substring uxv in a string over the alphabet A is called *left-hand site* if (u, x, v) is a left-hand pattern and *right-hand site* if (u, x, v) is a right-hand pattern. String x is called the *crossing* of the site uxv. In this paper we will use u, x, and v, and their subscript symbols to denote patterns and sites. For a string $wuxvz$ with a site uxv, we call wux the *left half* of $wuxvz$ with respect to site uxv, and xvz the *right half* of $wuxvz$ with respect to site uxv. We assume that for each string in I there are an unbounded number of copies available whenever they are needed for splicing. In the biological sense, we may assume that alphabet A consists of the four symbols each representing one of the four paired symbols $A/T, T/A, G/C$, and C/G, and B and C represent, respectively, the two types of enzymes depending on the locations of the overhangs.

DEFINITION 2.2. *For a splicing system $S = (A, I, B, C)$, by $L(S)$ we denote the set of strings generated by S which is formally defined as follows.*

(1) $I \subseteq L(S)$.
(2) *If $w_1u_1xv_1z_1$ and $w_2u_2xv_2z_2$ are in $L(S)$, and u_1xv_1 and u_2xv_2 are sites of the same hand, then $w_1u_1xv_2z_2$ and $w_2u_2xv_1z_1$ are also in $L(S)$.*

A language L is *splicing language* if there exists a splicing system S which generates L. In part (2) of the above definition, string $w_1u_1xv_2z_2$ is produced by splicing $w_1u_1xv_1z_1$ and $w_2u_2xv_2z_2$ on sites u_1xv_1 and u_2xv_2, respectively. Notice that $w_1u_1xv_2z_2$ is formed by writing the left half of $w_1u_1xv_1z_1$ w.r.t $u_1x_1v_1$ followed by the right half of $w_2u_2xv_2z_2$ w.r.t. u_2xv_2 with the crossing x overlapped. We will denote this operation as follows:

$$w_1\underline{u_1xv_1}z_1 \not\vdash w_2\underline{u_2xv_2}z_2 = w_1u_1xv_2z_2.$$

Notice that spliced sites are underlined. In the literature the following two classes of splicing systems have been studied, in particular, for their language properties and characterizations in terms of finite state automata.

DEFINITION 2.3 (see [2]). *A splicing system $S = (A, I, B, C)$ is* permanent *if, for each pair of strings $w_1u_1xv_1z_1$ and $w_2u_2xv_2z_2$ in A^* with sites u_1xv_1 and u_2xv_2 of the same hand, it has the following property: if y is a substring of w_1u_1x (respectively, xv_2z_2) that is the crossing of a site in $w_1u_1xv_1z_1$ (respectively, $w_2u_2xv_2z_2$), then this same substring y of $w_1u_1xv_2z_2$* is *the crossing of a site in $w_1u_1xv_2z_2$.*

Notice that the substring y must exist either in w_1u_1x of $w_1u_1xv_1z_1$ or in xv_2z_2 of $w_2u_2xv_2z_2$.

DEFINITION 2.4 (see [5]). *A* persistent *splicing system is defined as Definition* 2.3 *with the word* "is" *occurring in boldface replaced by* "contains an occurrence of."

Clearly, permanence implies persistence. To help the reader understand the above two definitions, consider the following operation of a splicing system:

$$w_1\underline{u_1xv_1}z_1 \not\vdash w_2\underline{u_2xv_2}z_2 = w_1u_1xv_2z_2.$$

Since x is a substring in w_1u_1x that is the crossing of a site in $w_1u_1xv_1z_1$, if the system is permanent, then the same x in $w_1u_1xv_2z_2$ is the crossing of a site in $w_1u_1xv_2z_2$. If the system is persistent, the same x contains an occurrence of the crossing of a site in $w_1u_1xv_2z_2$. In other words, $w_1u_1xv_2z_2$ contains a site whose crossing is contained in the substring x. Now, suppose that $w_1u_1xv_2z_2 = w_3u_3x_3v_3z_3$ such that $u_3x_3v_3$

is a site and the crossing x is contained in $x_3v_3z_3$. Consider the following splicing operation:

$$w_4\underline{u_4x_3v_4}z_4 \not\vdash w_3\underline{u_3x_3v_3}z_3 = w_4u_4x_3v_3z_3.$$

The substring x, which is contained in $x_3v_3z_3$, also exists in $w_4u_4x_3v_3z_3$ and, hence, the same x should have the crossing of a site. Now, we investigate some interesting splicing operations and introduce several lemmas that will be used in section 5. We need the following well-known theorem (e.g., [4, p.7]).

THEOREM 2.5. *Let $\alpha, \gamma \in A^+$ and $\beta \in A^*$ such that $\alpha\beta = \beta\gamma$. Then there exist $\alpha_1, \alpha_2 \in A^*$ and $p \geq 0$ such that $\alpha = \alpha_1\alpha_2$, $\gamma = \alpha_2\alpha_1$, and $\beta = \alpha^p\alpha_1 = \alpha_1\gamma^p$.*

Note that $p = \lfloor |\beta|/|\alpha| \rfloor$ and $|\alpha_1| = |\beta| \bmod |\alpha|$.

LEMMA 2.6. *Let $\alpha, \beta, \gamma \in A^+$ such that $\alpha\beta = \beta\gamma$. Then, for all $k > |\beta|$, string β is a substring of both α^k and γ^k.*

Proof. By Theorem 2.5 there are $\alpha_1, \alpha_2 \in A^*$ such that $\alpha = \alpha_1\alpha_2$ and $\beta = \alpha^p\alpha_1 = \alpha_1\gamma^p$, where $p = \lfloor |\beta|/|\alpha| \rfloor \geq |\beta|$. Clearly, $\alpha^{p+1} = \alpha^p\alpha_1\alpha_2 = \beta\alpha_2$ and $\gamma^{p+1} = \alpha_2\alpha_1\gamma^p = \alpha_2\beta$. □

LEMMA 2.7. *Let $w_1, w_2, w_3 \in L(S)$ of a splicing system $S = (A, I, B, C)$ such that for some sites uxv, $u_1x_1v_1$, and $u_2x_2v_2$,*

$$w_1 = w_{11}uxvz_1 = w_{12}u_1x_1v_1y_1z_1,$$
$$w_2 = w_{21}uxvy_0z_2 = w_{22}u_2x_2v_2y_2y_0z_2, \text{ and}$$
$$w_3 = w_{31}u_1x_1v_1y_1z_3 = w_{32}u_2x_2v_2y_2y_0z_3,$$

where $w_{ij}, z_1, z_2, z_3 \in A^*$ and $y_0 \in A^+$. Then $w_{11}uxv(y_0)^+z_2 \subseteq L(S)$.

Proof. We prove the lemma by showing that $w_{11}uxv(y_0)^kz_2 \in L(S)$, for all $k \geq 1$, by induction. The following splicing operation shows that $w_{11}uxv(y_0)^kz_2 \in L(S)$ for $k = 1$:

$$w_{11}\underline{uxv}z_1 \not\vdash w_{21}\underline{uxv}y_0z_2 = w_{11}uxvy_0z_2.$$

Suppose that $w_{11}uxv(y_0)^iz_2 \in L(S)$ for all $i < k$. We have $w_{11}uxv(y_0)^iz_2 = w_{12}u_1x_1v_1y_1(y_0)^iz_2$. The following sequence of splicing produces $w_{11}uxv(y_0)^{i+1}z_2$:

(1) $w_{31}\underline{u_1x_1v_1}y_1z_3 \not\vdash w_{12}\underline{u_1x_1v_1}y_1(y_0)^iz_2 = w_{31}u_1x_1v_1y_1(y_0)^iz_2$
$= w_{32}u_2x_2v_2y_2y_0(y_0)^iz_2$,

(2) $w_{22}\underline{u_2x_2v_2}y_2y_0z_2 \not\vdash w_{32}\underline{u_2x_2v_2}y_2y_0(y_0)^iz_2 = w_{22}u_2x_2v_2y_2y_0(y_0)^iz_2$
$= w_{21}uxvy_0(y_0)^iz_2$,

(3) $w_{11}\underline{uxv}z_1 \not\vdash w_{21}\underline{uxv}y_0(y_0)^iz_2 = w_{11}uxvy_0(y_0)^iz_2 = w_{11}uxv(y_0)^{i+1}z_2$. □

If we choose $w_1 = w_2 = w_3$, $z_1 = z_2 = z_3$, and $uxv = u_1x_1v_1y_1 = u_2x_2v_2y_2$ in Lemma 2.7, we get $w_{11} = w_{12} = w_{31}$ and $w_{21} = w_{22} = w_{32}$, and the first two expressions of the lemma become redundant. For this simple case of the lemma we present the following corollary.

COROLLARY 2.8. *Let $w \in L(S)$ of a splicing system. If $w = w_1uxvz = w_2uxvy_0z$, for some $w_1, w_2, z \in A^*$, $y_0 \in A^+$ and a site uxv, then $w_1uxv(y_0)^*z \subseteq L(S)$.*

LEMMA 2.9. *Let $w_1, w_2, w_3 \in L(S)$ of a splicing system S that satisfies the condition of Lemma 2.7 above. Then there exists a constant c such that for all $k \geq c$, the string $(y_0)^k$ contains the site uxv.*

Proof. Conditions (1), (2), and (3) of Lemma 2.7 implies that strings w_1, w_2, and w_3 can be superposed as shown in Figure 1. In the figure, heavy vertical lines indicate exact positions and others can vary. It is easy to see that the longest string among $uxvy_0$, $u_2x_2v_2y_2y_0$, $u_1x_1v_1y_1$, and uxv has the others as suffixes. The figure shows the case in which $u_2x_2v_2y_2y_0$ is the longest.

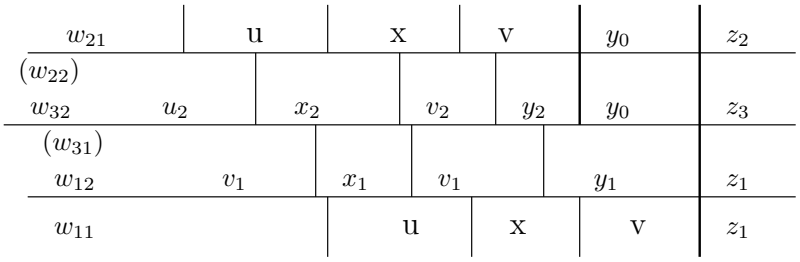


FIG. 1. *Superposing w_1, w_2, and w_3 for Lemma* 2.9.

Whichever the case, we have $zuxv = uxvy_0$ for some $z \in A^+$. By Lemma 2.6 with $\beta = uxv$ and $\gamma = y_0$, string ${y_0}^k$ contains the site uxv for all $k > |uxv|$. □

LEMMA 2.10. *Let $w_1, w_2, w_3 \in L(S)$ of a splicing system $S = (A, I, B, C)$ such that*

$$w_1 = w_{11}uxvz_1 = w_{12}u_1x_1v_1y_1vz_1,$$
$$w_2 = w_{21}u'xv'y_0z_2 = w_{22}u_2x_2v_2y_2y_0z_2, \text{ and}$$
$$w_3 = w_{31}u_1x_1v_1y_1z_3 = w_{32}u_2x_2v_2y_2y_0z_3,$$

for some $w_{ij}, z_1, z_2, z_3 \in A^*$ and $y_0 \in A^*$, where uxv and $u'xv'$ are sites of the same hand, and $u_1x_1v_1$ and $u_2x_2v_2$ are arbitrary sites. Then $w_{11}ux(v'y_0)^+z_2 \subseteq L(S)$.

Proof. Notice that $w_{11}ux = w_{12}u_1x_1v_1y_1$, $w_{21}u'xv' = w_{22}u_2x_2v_2y_2$, and $w_{31}u_1x_1v_1y_1 = w_{32}u_2x_2v_2y_2y_0$. We will prove the lemma by showing that $w_{11}ux(v'y_0)^kz_2 \in L(S)$ for all $k \geq 1$. The following splicing operation shows that $w_{11}ux(v'y_0)^kz_2 \in L(S)$ for $k = 1$:

$$w_{11}\underline{uxv}z_1 \not\vdash w_{21}\underline{u'xv'}y_0z_2 = w_{11}uxv'y_0z_2.$$

For $k > 1$, suppose that $w_{11}ux(v'y_0)^iz_2 \in L(S)$ for all $i < k$. Then by the property of w_1, we have

$$w_{11}ux(v'y_0)^iz_2 = w_{12}u_1x_1v_1y_1(v'y_0)^iz_2.$$

We can produce $w_{11}ux(v'y_0)^{i+1}z_2$ by the following sequence of splicing operations:

(1) $w_{31}\underline{u_1x_1v_1}y_1z_3 \not\vdash w_{12}\underline{u_1x_1v_1}y_1(v'y_0)^iz_2 = w_{31}u_1x_1v_1y_1(v'y_0)^iz_2$
$= w_{32}u_2x_2v_2y_2y_0(v'y_0)^iz_2,$

(2) $w_{22}\underline{u_2x_2v_2}y_2y_0z_2 \not\vdash w_{32}\underline{u_2x_2v_2}y_2y_0(v'y_0)^iz_2 = w_{22}u_2x_2v_2y_2y_0(v'y_0)^iz_2$
$= w_{21}u'xv'y_0(v'y_0)^iz_2,$

(3) $w_{11}\underline{uxv}z_1 \not\vdash w_{21}\underline{u'xv'}y_0(v'y_0)^iz_2 = w_{11}uxv'y_0(v'y_0)^iz_2$
$= w_{11}ux(v'y_0)^{i+1}z_2.$ □

In Lemma 2.10 if $w_1 = w_2 = w_3$, $vz_1 = z_2 = z_3$, $ux = u_1x_1v_1y_1$, and $u'xv' = u_2x_2v_2y_2$, the first two expressions become redundant. For this simple case of the lemma we present the following corollary.

COROLLARY 2.11. *Let $w \in L(S)$ of a splicing system S such that $w = w_1uxvz = w_2u'xv'y_0vz$ for some $w_1, w_2, z \in A^*$ and sites uxv and $u'xv'$ of the same hand. Then $w_1ux(v'y_0)^*vz \subseteq L(S)$.*

LEMMA 2.12. *Let $w_1, w_2, w_3 \in L(S)$ of a splicing system S that satisfies the condition of Lemma* 2.10 *above. If S is persistent, there exists a constant c such that for all $k \geq c$ the string $(v'y_0)^k$ contains a site.*

Proof. It suffices to show that x is a substring of $(v'y_0)^k$. As for the proof of Lemma 2.9, we can superpose w_1, w_2, and w_3 as shown in Figure 2, which implies

$zx = xv'y_0$ for some $z \in A^+$. By Lemma 2.6 with $\gamma = v'y_0$ and $\beta = x$, string x is a substring of $(v'y_0)^k$ for all $k > |v'y_0|$. □

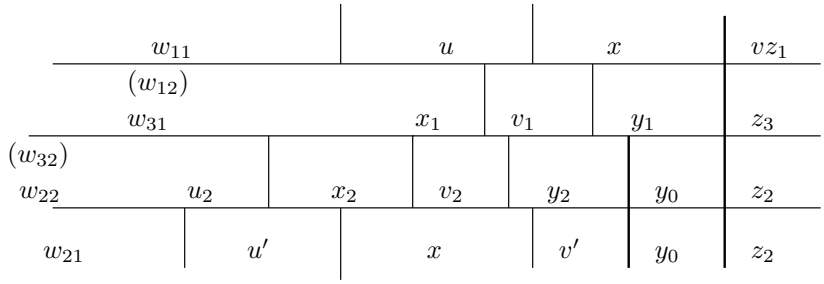


FIG. 2. *Superposing w_1, w_2, and w_3 for Lemma* 2.12.

3. Automata and input memory spans. This section investigates finite state transition graphs and develops some concepts and lemmas that will be used in sections 4 and 5 for constructing an automaton which recognizes the language of a given splicing system. In this paper we use the notation of [9] for the automaton $M = (Q, A, \delta, q_{st}, F)$. The automata that our algorithm constructs is nondeterministic. Thus, $\delta(p, x) \subseteq Q$, for a state $p \in Q$ and string $x \in A^*$. We write $q \in \delta(p, x)$ to refer to a sequence of transitions in response to the input string x beginning at p and ending at q. (Note that q_{st} designates the start state of an automaton.) By a graph we mean the state transition graph of the automaton, which is an edge-labeled directed graph. Following the convention we allow an edge to have more than one label. We assume that every state is reachable from the start state, and from every state an accepting state is reachable. Hence, the graph does not have dead states.

A *path* is a sequence of, possibly repeating, states $q_0 q_1 \dots q_n$ such that there is a directed edge from q_i to q_{i+1}, $0 \le i < n$. An *accepting path* is a path which starts from the start state and ends in an accepting state. A *span* is a string of symbols collected along a path one symbol from each edge. Notice that there can be more than one span on a path because more than one symbol can be assigned on an edge. If $q \in \delta(p, x)$, there is a path from p to q which has span x. An *accepting span* is a span of an accepting path. By A^k ($A^{\le k}$), we denote the set of strings of length k ($\le k$) over the alphabet A. By $L(G)$ we denote the set of accepting spans in state transition graph G, i.e., the language of the automaton represented by graph G. We need the following definition which was introduced in [7].

DEFINITION 3.1 (input memory span). *Let $M = (Q, A, \delta, q_{st}, F)$ be an automaton. A string w is an input memory span (IMS) of state q if there is a state p such that $q \in \delta(p, w)$. For a state $q \in Q$ and a nonnegative integer k, the set of input memory spans of order k of state q, denoted by $IMS(q, k)$, is defined as follows:*

$$IMS(q,k) = \{x \mid x \in A^k, q \in \delta(q_{st}, wx), w \in A^*\} \bigcup \{x \mid x \in A^{<k}, q \in \delta(q_{st}, x)\}.$$

We can prove that this definition is equivalent to the statement of the following lemma which is used for computing $IMS(q, k)$ of all states q of a given automaton as shown in Figure 3. We leave the proof of the lemma for the reader.

LEMMA 3.2. *Let q be a state of an automaton $M = (Q, A, \delta, q_{st}, F)$; then $IMS(q, k)$ can be defined as follows.*

(1) *$IMS(q, 0) = \{\epsilon\}$ and, for all $k \ge 0$, $\epsilon \in IMS(q_{st}, k)$.*

(2) *For $k > 0$, $a \in A$ and $x \in A^*$ $xa \in IMS(q,k)$, if and only if there exists $p \in Q$ such that $x \in IMS(p, k-1)$ and $q \in \delta(p,a)$.*

Procedure Compute_IMS(G, k)
(//G is a finite state transition graph which is defined as $G = (A, Q, \delta, q_{st}, F)$.
This procedure computes IMS of order k of all states in G. //)
for each state $q \in Q$ let $IMS(q, 0) = \{\epsilon\}$;
for $i = 1$ to k **do**
for each state $q \in Q$ **do**
if q is the start state **then** let $IMS(q, i) = \{\epsilon\}$
else let $IMS(q, i) = \emptyset$;
for $i = 1$ to k **do**
begin
for each state $q \in Q$ **do**
if $q \in \delta(p, a)$, for some $p \in Q$ and $a \in A$, **then**
for each $x \in IMS(p, i-1)$ put xa in $IMS(q, i)$;
end;

FIG. 3. *Algorithm for computing IMS.*

DEFINITION 3.3 (monomial state, monomial automaton). *A state q is monomial with respect to string x, if $x \notin IMS(q, |x|)$ or x is the only IMS of length $\leq |x|$ of q, i.e., $\{x\} = IMS(q, |x|)$. An automaton is monomial w.r.t. string x if every state of the automaton is monomial w.r.t. x. An automaton is monomial of order k if every state of the automaton is monomial w.r.t. every string of length k.*

If a state q of an automaton is not monomial w.r.t. a string x, we can transform the automaton to an equivalent one such that the state is monomial w.r.t. x. Appendix A shows an algorithm for monomializing a state transition graph G w.r.t. a string x. (For now, ignore statements 5 and 11 which are for algorithm $SPLICE$ in Appendix B to block splitting "merged" transitions. We will be back to this later in section 4.)

The basic idea is state splitting. Let $x = a_1 a_2 \dots a_n$, $n \geq 1$, and $x_i = a_1 a_2 \dots a_i$, with $x_0 = \epsilon$. By definition, the graph G is monomial w.r.t. the null string ϵ. Suppose that G is monomial w.r.t. string x_i, $i < n$. If a state q is not monomial w.r.t. a string $x_i a_{i+1}$, then the algorithm splits q into two equivalent states, say q_1 and q_2, such that $IMS(q_1, i+1) = \{x_i a_{i+1}\}$ and $x_i a_{i+1} \notin IMS(q_2, i+1)$. Figure 4 shows an example of monomializing a graph w.r.t. string *aaaa*. Notice that state q in part (a) of the figure, which is not monomial w.r.t. string *aaaa*, is split into two together with its ancestors along the path (of shaded nodes) which has span *aaaa*.

LEMMA 3.4. *Let M be an automaton with n states and alphabet size c. Automaton M can be monomialized to order k by increasing the number of states to no more than nc^{k+1}.*

Proof. By Definition 3.1, for a state q, we have

$$|IMS(q,k)| \leq c^k + c^{k-1} + \cdots + c^0 = \frac{c^{k+1} - 1}{c - 1} \leq c^{k+1}.$$

Hence, by splitting each state into no more than c^{k+1} equivalent states we can monomialize the automaton to order k. The resulting automaton will have no more than nc^{k+1} states. □

DEFINITION 3.5 (see [8]). *For a string $x \in A^*$ and a nonnegative integer k, define*

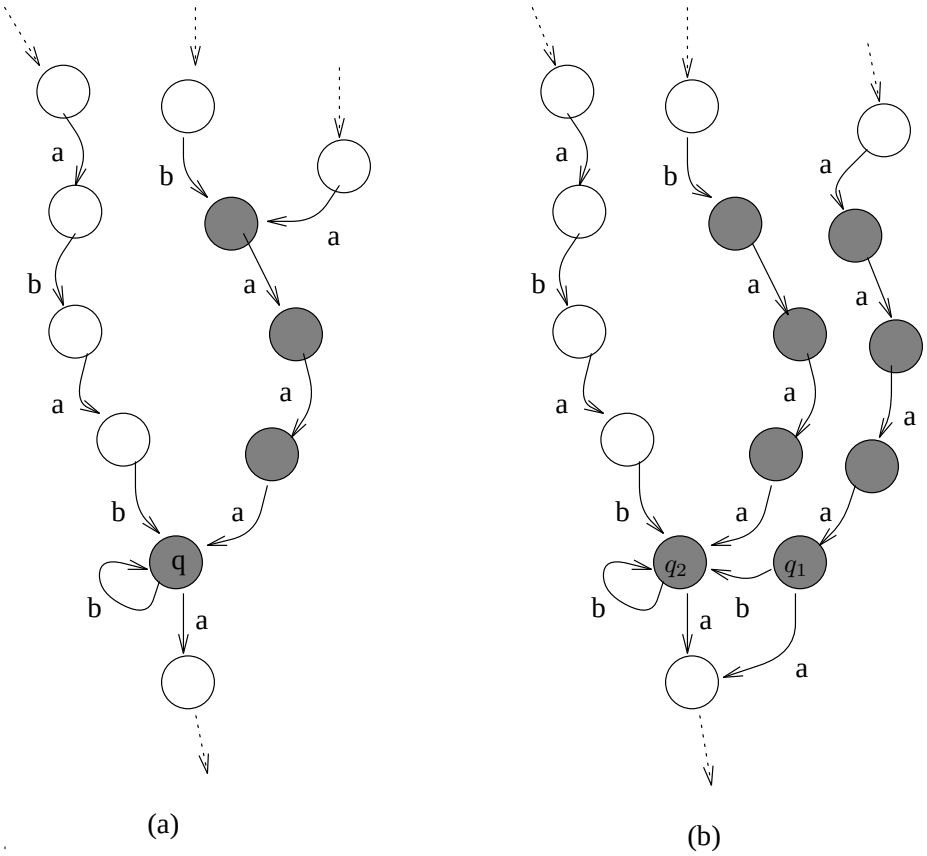


FIG. 4. *Monomializing state q w.r.t. string aaaa.*

$f_k(x)$*: the prefix of* x *of length* k*,*
$t_k(x)$*: the suffix of* x *of length* k*, and*
I_{k+1}*: the set of all substrings of* x *of length* $k+1$*.*

A language $L \subseteq A^*$ *is* $(k+1)$*-testable in strict sense* ($(k+1)$*-LTSS*) *if and only if there exists finite sets* α*,* β*, and* γ *such that*

$$x \in L \Longleftrightarrow (f_k(x) \in \alpha) \bigwedge (t_k(x) \in \gamma) \bigwedge (I_{k+1}(x) \subseteq \beta).$$

A language is strictly locally testable (or locally testable in the strict sense, or LTSS) if it is $(k+1)$*-LTSS for some* $k \geq 0$*.*

In other words, a language is $LTSS$ if its membership of a string x can be decided by inspecting the substrings of x of some constant length. It is irrelevant to the order of appearance of the substrings and other global properties of the string. An automaton is $LTSS$ if its language is $LTSS$. Theorem 3.6 below shows a simple necessary and sufficient condition that the state transition graph of a reduced deterministic finite automaton should have if it is $LTSS$.

In [5], it is shown that the languages generated by persistent splicing systems are strictly locally testable if the initial set I is finite. This proof was based on an algebraic property of locally testable languages. In section 5 we will give another proof of this fact by showing that the automaton constructed in section 4 is strictly

locally testable. For the proof we use the following property of the state transition graphs of strictly locally testable automata that we introduced in [6].

THEOREM 3.6 (see [6]). *Let $M = (Q, A, \delta, q_{st}, F)$ be a deterministic finite state automaton which is reduced. The language $L(M)$ is LTSS if and only if there is no pair of distinct states $p, q \in Q$ which are not dead state and $w \in A^+$ such that $\delta(p, w) = p$ and $\delta(q, w) = q$.*

In other words, an automaton is not $LTSS$ if its state transition graph has two distinct looping paths with the same span. For example, Figure 5(a) shows an automaton which violates Theorem 2.5 in two cases: one is $\delta(r, 0) = r$ and $\delta(q, 0) = q$, and the other is $\delta(p, 011) = p$ and $\delta(q, 011) = q$. The automaton in Figure 5(b) is $LTSS$.

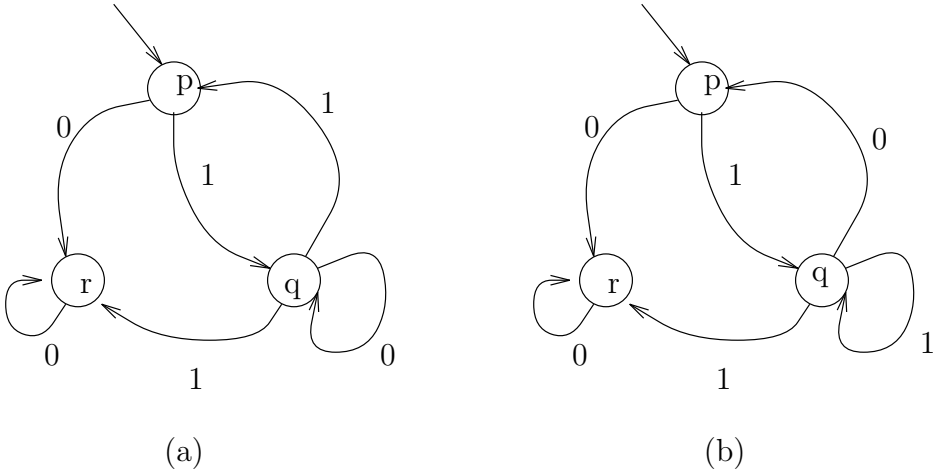


FIG. 5. (a) *An automaton which is not LTSS*; (b) *an LTSS automaton.*

If the state transition graph of a nondeterministic automaton does not have two identical looping paths, then it is $LTSS$ because its reduced deterministic version of the state transition graph will also have no such looping paths as we now prove.

LEMMA 3.7. *Let $M = (Q, A, \delta, q_{st}, F)$ be a nondeterministic automaton having the property that, for all $p, q \in Q$ and a string $x \in A^+$, if $p \in \delta(p, x)$ and $q \in \delta(q, x)$, then $p = q$. Then the language $L(M)$ is LTSS.*

Proof. Let $n = |Q|$. Assuming the hypothesis of the lemma, we note that, for all $w_1, w_2 \in A^*$ and $y \in A^+$ such that both $\delta(q_{st}, w_1 y^n)$ and $\delta(q_{st}, w_2 y^n)$ are defined, string $w_1 y^i w_3 \in L(M)$ if and only if $w_2 y^i w_3 \in L(M)$ for all $i \geq n$.

Let $M' = (Q', A, \delta', q'_{st}, F')$ be the reduced deterministic version of M. For a pair of nondead states $p', q' \in Q'$ and $y \in A^+$, suppose that $\delta'(p', y) = p'$ and $\delta'(q', y) = q'$. Let $w_1, w_2 \in A^*$ such that $\delta(q'_{st}, w_1) = p'$ and $\delta(q'_{st}, w_2) = q'$. In the nondeterministic automaton M, both $\delta(q_{st}, w_1 y^n)$ and $\delta(q_{st}, w_2 y^n)$ should be defined. Hence, for all $w_3 \in A^*$ and $i \geq n$, string $w_1 y^i w_3 \in L(M)$ if and only if $w_2 y^i w_3 \in L(M)$. It follows that string $w_1 y^i w_3 \in L(M')$ if and only if $w_2 y^i w_3 \in L(M')$, which implies that, for all $w_3 \in A^*$, $\delta(p', w_3)$ is in an accepting state if and only if $\delta(q', w_3)$ is. This implies that $p' = q'$. By Theorem 3.6, $L(M)$ is $LTSS$. ∎

We can easily extend Lemma 3.7 as follows.

LEMMA 3.8. *Let $M = (Q, A, \delta, q_{st}, F)$ be a nondeterministic automaton. The language $L(M)$ is LTSS if for every pair of identical looping paths corresponding to $p \in \delta(p, w)$ and $q \in \delta(q, w)$, for some $p, q \in Q$ and $w \in A^+$ in the transition graph,*

the looping paths have a common state r such that $r \in \delta(p, w_1)$ and $r \in \delta(q, w_1)$ for some prefix w_1 of w.

Proof. Let $w = w_1 w_2$. Since $r \in \delta(r, w_2 w_1)$, the two looping paths can be merged into a single looping path without affecting the language of the automaton. Let M' be the resulting automaton. By Lemma 3.7 $L(M')$ is $LTSS$. □

4. Constructing an automaton for a splicing system. This section describes an algorithm *CONSTRUCT* which, given a splicing system $S = (A, I, B, C)$, constructs the finite state transition graph of an automaton whose language is $L(S)$. Appendix B shows a high level description of the algorithm. We assume that I is regular which is given in terms of the finite state transition graph G of an automaton which recognizes I. Graph G is iteratively modified by subroutine *SPLICE* which merges and links certain states until no such modifications are possible.

Let G_i, $i \geq 1$, be the graph that results from ith iteration of the algorithm with $G_0 = G$. For each pattern (u, x, v), subroutine $SPLICE$ modifies G_i in three major steps in each iteration: state monomialization and merging in step I, linking in step II, and state collapsing in step III. In steps I(1) and I(2) the algorithm monomializes graph G w.r.t. uxv using algorithm $Monomialize$ in Appendix A which was described in section 3. Note that by statements 5 and 11 of the algorithm, if a state q has incoming transitions which are marked as "merged" by step I-5 of algorithm $SPLICE$, those "merged" transitions are kept merged either in q_1 or q_2 depending on whether q_1 has a marked incoming transition or not. Figure 6 shows what will happen if the graph in Figure 5 is monomialized w.r.t. string *aaaa* with both of the incoming transitions (thick edges) to the dark-colored state marked as "merged."

For a pattern (u, x, v) of system S, let $Q(uxv)$ denote the set of states p whose IMS of length $|uxv|$ contains uxv, i.e., $uxv \in IMS(p, |uxv|)$, after the monomialization step. Let $Q(ux, v)$ denote the set of states q such that $ux \in IMS(q, |ux|)$ and $r \in \delta(q, v)$ for a state $r \in Q(uxv)$. In step I the algorithm finds $Q(uxv)$ and $Q(ux, v)$ for each pattern (u, x, v) and merges all states in $Q(uxv)$ into a single state, which we will denote by $MQ(uxv)$, and marks all converged transitions as "merged" (by step I(5)). These "merged" transitions will be kept merged on a common state throughout the computation.

In step II the algorithm constructs a link with span v' from every state in $Q(ux, v)$ to state $MQ(u'xv')$ for every pattern (u', x, v') of the same hand with the same crossing as that of pattern (u, x, v). We call states in $Q(ux, v)$ *link sources* and the state $MQ(u'xv')$ *link destination*. Note that links are not marked as "merged", though they end at a merged state. Figure 7 illustrates merging and linking operations for sites uxv and $u'xv'$, where highlighted nodes are merged states whose incoming transitions corresponding to the solid edges will be marked as "merged."

Step III collapses equivalent states that meet certain conditions. This step is needed to guarantee that the graph does not grow indefinitely. We will go back to this step for further details after Lemma 4.4.

Figure 8 shows an example of constructing an automaton which recognizes the language generated by splicing system $S = (A, I, B, C)$, where $A = \{a, b\}$, $I = \{baa, aaba, bb\}$, $B = \{(b, a, a), (a, a, b), (ba, b, a), (\epsilon, b, b)\}$, and $C = \emptyset$. Part (a) of the figure is the state transition graph which recognizes I, part (b) is the result of processing sites (b, a, a) and (a, a, b), and part (c) is the final graph after processing sites (ba, b, a) and (ϵ, b, b). Notice that the highlighted state in part (b) is split into two in part (c) by monomialization step. We leave it for the reader to show that the language of the automaton is $L(S) = \{baa, aaba, bb, aaa, baba, babb, ba\}$.

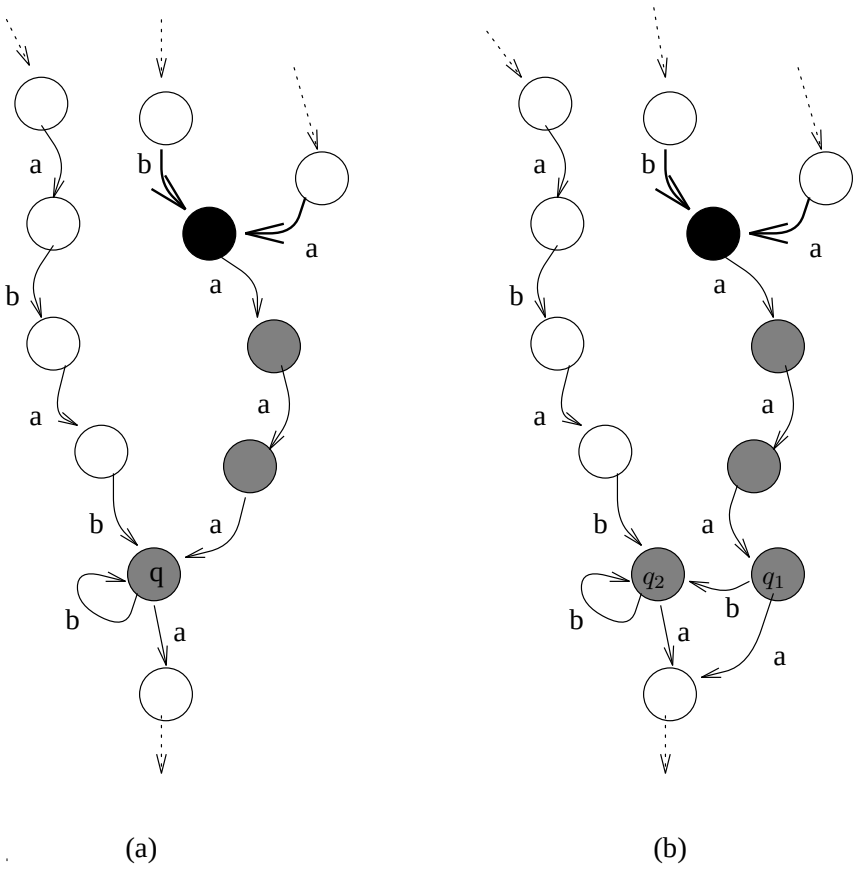


FIG. 6. *State splitting to monomialize state q w.r.t. aaaa.*

Now, we study algorithm $CONSTRUCT$ and develop several lemmas that will be used in section 5 where we will prove that if G is the output of the algorithm then $L(G) = L(S)$. We first present the main theorem of this section which proves that algorithm $CONSTRUCT$ terminates under the presumption of finiteness of G_i, which will be ascertained by Lemma 4.7.

THEOREM 4.1. *If the size of the graph G_i is finite for all $i \geq 0$, then there exists a k such that $L(G_k) = L(G_{k+1})$ and algorithm $CONSTRUCT$ terminates after $(k+1)$st iteration.*

Proof. Suppose that the algorithm does not terminate. This implies that during each iteration subroutine $SPLICE$ finds $|Q(uxv)| > 1$ for some pattern (u, x, v). Since the size of the automaton is finite, it should be that $G_i = G_j$, for some $j > i \geq 0$, possibly with different labels on states. Suppose that in jth iteration it was found that $|Q(uxv)| > 1$, and all the states in $Q(uxv)$ have been merged into one state by step I(5) of algorithm $SPLICE$. Then, in ith iteration, the same states in $Q(uxv)$ should have been merged into a state with all the converged transitions to that state marked as "merged" by step I(5) of algorithm $SPLICE$. Since "merged" transitions to a state are kept in a common state (by statements 5 and 11 of algorithm *Monomialize*), algorithm $SPLICE$ should have $|Q(uxv)| = 1$ in jth iteration. We are in a contradiction. It follows that in jth iteration every nonempty set $Q(uxv)$

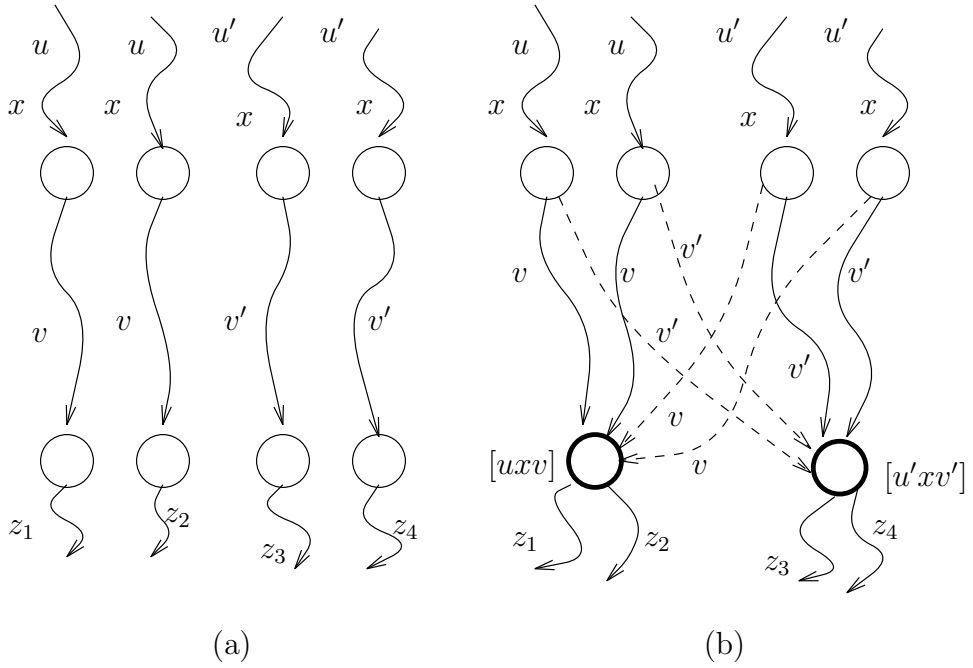


FIG. 7. *Merging and linking.*

should have only one element whose incoming transitions have been merged in previous iteration. No state merging or linking will occur in this iteration and the graph does not change, causing the algorithm to terminate in the next iteration. We take $i = k$ for the theorem. □

For a string w, by $SF(w)$ we denote the set of suffixes of w. Let p and q be two states. By I_{pq} we denote the set of spans which start from p and end at q. By F_p and T_p we, respectively, denote the set of spans that start from q_{st} and end in p and the set of spans that start from p and end at an accepting state. Formally,

$F_p = \{w \mid w \in A^*, p \in \delta(q_{st}, w)\}$,
$T_p = \{w \mid w \in A^*, r \in \delta(p, w), r \in F\}$,
$I_{pq} = \{w \mid w \in A^*, q \in \delta(p, w)\}$.

LEMMA 4.2. *Let p be a state in $Q(uxv)$ which is not monomial w.r.t. a site uxv after the monomialization step (i.e., step* (I-2)*) of algorithm SPLICE. For every span $w \in F_p$, if $uxv \notin SF(w)$, then there exists a span $\hat{w} \in F_p$ which satisfies the following conditions:*

(a) *$uxv \in SF(\hat{w})$,*
(b) *$w = w_1u'x'v'y$ and $\hat{w} = \hat{w}_1u'x'v'y$, for some $w_1, \hat{w}_1, y \in A^*$ and a site $u'x'v'$, such that $w_1u'x'v', \hat{w}_1u'x'v' \in F_r$, for a state r.*

Proof. If there is no string $\hat{w}$ that satisfies condition (a), then $uxv \notin IMS(p, |uxv|)$ and state p is monomial w.r.t. uxv by Definition 3.3. We are in a contradiction. Suppose that w and $\hat{w}$ do not satisfy condition (b) of the lemma. Then by algorithm *Monomialize*, state r and all its descendents up to p along the path corresponding to $p \in \delta(r, y)$ should have been split, with p split into two equivalent states p_1 and p_2 which are monomial w.r.t. string uxv. It follows that no such state p can be in $Q(uxv)$ that is not monomial w.r.t. string uxv. Again, we are in a contradiction. □

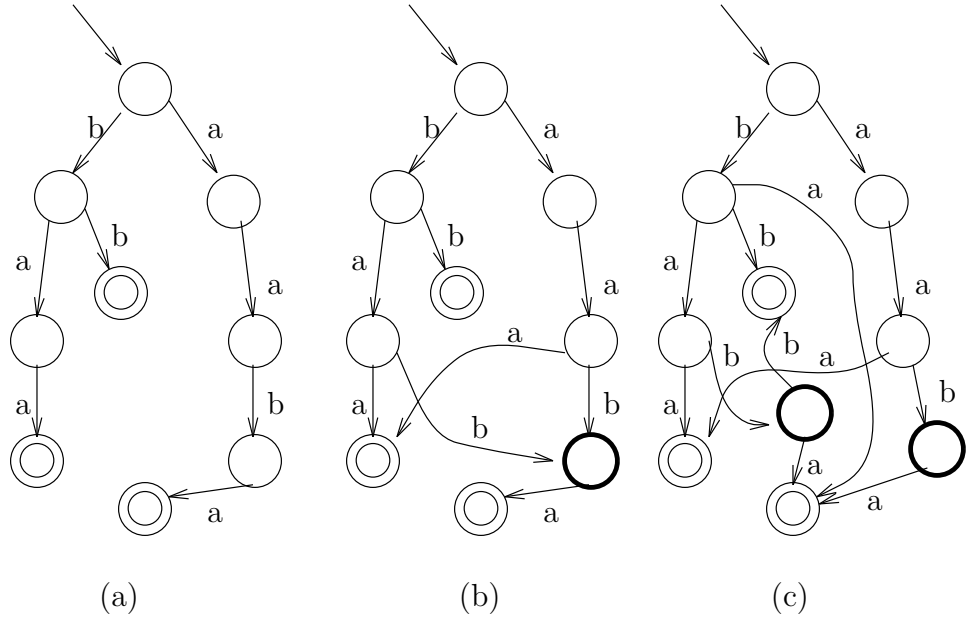


FIG. 8. *Constructing an automaton for* $S = (A, I, B, C)$, *where* $A = \{a, b\}$, $I = \{baa, aaba, bb\}$, $B = \{(b, a, a), (a, a, b), (ba, b, a), (\epsilon, b, b)\}$ *and* $C = \emptyset$.

Actually, condition (b) of Lemma 4.2 implies that both of the last transitions of $r \in \delta(q_{st}, w_1u'x'v')$ and $r \in \delta(q_{st}, \hat{w}_1u'x'v')$ have mark "merged" given by algorithm $SPLICE$ when it computed $Q(u'x'v')$ and merged the set into a state. Consequently, monomializing the graph w.r.t. string uxv, algorithm *Monomialize* should have kept both of these marked transitions merged at state r. (Recall statements 5 and 11 of algorithm *Monomialize* and Figure 6.) Similarly, the following lemma holds.

LEMMA 4.3. *Let* p *be a state in* $Q(ux, v)$ *which is not monomial w.r.t.* ux *after the monomialization step (i.e., step* I(2)*) of algorithm* $SPLICE$ *for site* uxv. *For every span* $w \in F_p$, *if* $ux \notin SF(w)$, *then there exists a span* $\hat{w} \in F_p$, *not necessarily distinct from* w, *which satisfies the following conditions:*

(a) $ux \in SF(\hat{w})$,
(b) $w = w_1u'x'v'y$ *and* $\hat{w} = \hat{w}_1u'x'v'y$, *for some* $w_1, \hat{w}_1, y \in A^*$ *and a site* $u'x'v'$, *such that both* $w_1u'x'v', \hat{w}_1u'x'v' \in F_r$, *for a state* r.

Clearly, if w is an accepting span of G_i, it will remain as an accepting span, possibly of a different path, after monomialization and merging in step I, linking in step II, and link merging in step III. Hence, we have the following.

LEMMA 4.4. *For a string* $w \in A^*$, *if* $w \in L(G_i)$, *then* $w \in L(G_{i+1})$.

Now, we are ready to show that the size of graph G_i is bounded by some constant factor of the size of the input G_0. By Lemma 3.4, monomializing G_0 increases its size by no more than a constant factor. Clearly, merging does not increase the graph size. Linking introduces new paths and consequently increases the graph size. If one of the states on the links becomes a link source which in turn introduces new links, recursively inducing an unbounded number of links, the graph may grow unbounded and the algorithm will not terminate. Figure 9(a) illustrates this possibility, where the highlighted node is a link destination $MQ(u_0xv_0)$, and link sources in $Q(u_1x, v_1)$ are labeled by $[u_1x]$. (In the figure only one link with span v_0 is shown from each link

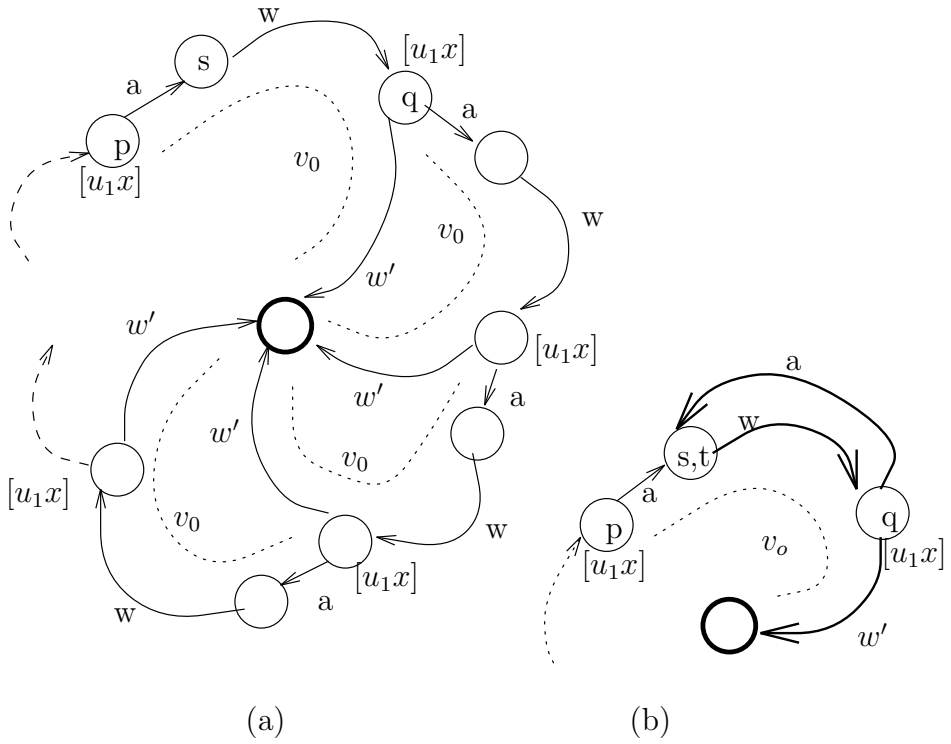


FIG. 9. *Link recurrence and link merging.*

source to the link destination.)

We solve this problem by merging those links whose link sources have the same input memory spans of order equal to the length of the longest site of the system. Part (b) of Figure 9 illustrates the result of this merging operation on part (a) of the figure.

Let $p, q \in Q(u_1x, v_1)$ and $r = MQ(u_0xv_0)$ in G_i, and in step II the algorithm has put links with span v_0 from p to r and q to r. Let s and t be the first states on these links as shown in Figure 9(a). (Note that in the figure $v_0 = aww'$, where a, w, w' are, respectively, the spans on p to s, s to q, and q to r paths.) If we can show that s and t are equivalent in the output graph of algorithm CONSTRUCT, we can merge them and their successors pairwise.

Notice that if neither s nor t is an ancestor of the other, they are equivalent in G_i. However, if a state on either p to r link or q to r link (not both) is later merged into (or linked to) other state, s and t can no longer be equivalent. This is possible because the states on p to r link may have different input memory spans from those of states on q to r link. If t is a descendant of s on identical links that will be recursively generated an unbounded number of times as shown in Figure 9(a), we can also merge s and t and their successors pairwise because $T_s = T_t = w(aw)^*w'T_r$. The following lemma formally presents this idea, which is implemented by step III of algorithm *SPLICE*.

LEMMA 4.5. *Let $r = MQ(u_0xv_0)$ and $p, q \in Q(u_1x, v_1)$ for some sites u_0xv_0 and u_1xv_1 of the same hand. Let k be the length of the longest possible site of the system. If $IMS(p, k) = IMS(q, k)$, then we can merge the two links from p to r and from q*

to r, excluding p and q, into a single link without affecting the language of the output graph from algorithm CONSTRUCT.

Proof. Let $ps_0s_1 \dots s_n$ be p to r link and $qt_0t_1 \dots t_n$ be q to t link, where $s_n = t_n = r$. Since $IMS(p,k) = IMS(q,k)$, for a site $u'x'v'$, state s_i is in $Q(u'x', v')$ if and only if t_i is in $Q(u'x', v')$, and s_i is in $Q(u'x'v')$ if and only if t_i is in $Q(u'x'v')$. Hence, if s_i is merged into (or linked to) a state, t_i will be merged into (or linked to) the same state. States s_i and t_i remain equivalent throughout the computation. We can merge the two links by collapsing s_i and t_i pairwise, for all i, $0 \le i \le n$, without affecting the language of the graph. □

LEMMA 4.6. *If I is given in terms of a finite state transition graph G_0, then, for every $i \ge 0$, the number of linking states in G_i is finite.*

Proof. Let $c = |A|$, and let k_1 and k_2, respectively, be the length of the longest possible site of the splicing system and the length of the longest link. After step III no two separate links exist from a pair of link sources p and q which are linked to the same destination with the property that $IMS(p, k_1) = IMS(q, k_1)$. Let $\mu = c^{k_1+1}$, which is the largest possible size of the set of IMS of a state (recall the proof of Lemma 3.4). There are no more than 2^μ different sets of input memory spans of length $\le k_1 + 1$. Hence, there are no more than 2^μ separate links in G_i that link to the same merged state. There are no more than $k_2 - 1$ states on a link, and at most $|B| + |C|$ merged states exist in G_i. It follows that G_i has $(k_2 - 1)(|B| + |C|)2^\mu$ linking states, which is finite. □

LEMMA 4.7. *If I is given in terms of a finite state transition graph G_0, then G_i is finite for all $i \ge 0$.*

Proof. By Lemma 3.4 monomializing G_0 increases the graph size by no more than a constant factor of the graph size. By Lemma 4.6, the total number of states introduced by linking is also no more than a constant factor of the given graph size. If a link is introduced from a link source p to a link destination q, the IMS of q and its descendents may change, and, consequently, monomialization during the subsequent iterations of the algorithm may increase the number of states. However, monomialization introduces no more than k_1 states per link introduced, where k_1 is the maximum site length of the system. Since the number of links is finite, throughout the computation, the number of states introduced by monomialization is also finite. □

5. Characterization theorem. This section proves a characterization of genetic splicing systems in terms of an automaton by showing that for a given splicing system S, the automaton generated by algorithm $CONSTRUCT$ recognizes the language $L(S)$. In particular, we prove the following three main theorems of the paper.

THEOREM 5.1. *Let G be the output graph from algorithm CONSTRUCT for a splicing system $S = (A, I, B, C)$ with I given in terms of the state transition graph of an automaton whose language is I. Then $L(G) = L(S)$.*

Proof. Lemmas 5.9 and 5.10 will, respectively, prove that $L(G) \subseteq L(S)$ and $L(S) \subseteq L(G)$. □

Since I is given in terms of the finite state transition graph of an automaton whose language is I, Theorem 5.1 implies a constructive proof of the following theorem which was proved in [1] using alphabetic dominos.

THEOREM 5.2. *Let $S = (A, I, B, C)$ be a splicing system. If I is regular, so is $L(S)$.*

THEOREM 5.3. *Let $S = (A, I, B, C)$ be a splicing system with finite I. If S is persistent, then $L(S)$ is LTSS.*

Proof. We defer the proof till we prove Lemma 5.11.

To prove Theorem 5.1, we first investigate how each step of algorithm $SPLICE$ affects the language $L(G_i)$. The algorithm has four operations that may affect the language: monomialization (step I(2)), state merging (step I(5)), linking (step II(2)), and link merging (step III). State monomialization and link merging do not affect the language because the one splits a state into equivalent states and the other merges equivalent states that will remain equivalent throughout the computation. By the following two lemmas we shall investigate the effect of state merging and linking operations.

LEMMA 5.4. *For a pattern (u, x, v) of a splicing system S, let G' be the resulting graph after merging the states in $Q(uxv)$ of a graph G to a single state $MQ(uxv)$ in step* I(5) *of algorithm $SPLICE$. If $L(G) \subseteq L(S)$, then $L(G') \subseteq L(S)$.*

Proof. Let $p, q \in Q(uxv)$. Clearly, all accepting spans that are introduced by merging p and q are in the following set $(F_p + F_q)(I_{pq} + I_{qp})^*(T_p + T_q)$. (Recall the notation of F_p, T_p, and Ipq from section 4.) No other spans are affected by the operation. Hence, for the proof it is enough to show the following:

$$(F_p + F_q)(I_{pq} + I_{qp})^*(T_p + T_q) \subseteq L(S).$$

Let $w \in F_p$, $w' \in F_q$, $z \in T_p$, $z' \in T_q$, and $y_k y_{k-1} \dots y_1 \in (I_{pq} + I_{qp})^*$, for some $k \geq 0$, where y_i is either in I_{pq} or in I_{qp}. Let $Y_{ki} = y_k y_{k-1} \dots y_i$, $Y_{ii} = y_i$, and $Y_{0i} = \epsilon$. For the proof it is enough to show that, for all $k \geq 0$,

$$(w + w')Y_{k1}(z + z') \subseteq L(S).$$

We show this by induction on k. Since $p \in Q(uxv)$, by Lemma 4.2, for every $w \in F_p$, if $uxv \notin SF(w)$, there exists $\hat{w} \in F_p$ such that $uxv \in SF(\hat{w})$, $\hat{w} = \hat{w}_1 u_1 x_1 v_1 y$ and $w = w_1 u_1 x_1 v_1 y$ for some $\hat{w}_1, w_1, y \in A^*$ and a site $u_1 x_1 v_1$. The same property holds for every string $w' \in F_q$.

For any pair of strings $w \in F_p$ and $w' \in F_q$, find $\hat{w} \in F_p$ and $\hat{w}' \in F_q$ such that $uxv \in SF(\hat{w})$, $uxv \in SF(\hat{w}')$, $w = w_1 u_1 x_1 v_1 y$, $\hat{w} = \hat{w}_1 u_1 x_1 v_1 y$, $w' = w'_2 u_2 x_2 v_2 y'$, and $\hat{w}' = \hat{w}'_1 u_1 x_1 v_1 y$. Since $wz, \hat{w}z, w'z', \hat{w}'z' \in L(G)$, by the hypothesis of the lemma we have $wz, \hat{w}z, w'z', \hat{w}'z' \in L(S)$. With $\hat{w}z = \hat{w}_0 uxvz$ and $\hat{w}'z' = \hat{w}'_0 uxvz'$, the system generates $\hat{w}z'$ and $\hat{w}'z$ as follows:

$$\hat{w}_0 \underline{uxv} z \not\rightarrow \hat{w}'_0 \underline{uxv} z' = \hat{w}_0 uxvz' = \hat{w}z',$$

$$\hat{w}'_0 \underline{uxv} z' \not\rightarrow \hat{w}_0 \underline{uxv} z = \hat{w}'_0 uxvz = \hat{w}'z.$$

It follows that

$$(\hat{w} + \hat{w}')(z + z') \subseteq L(S).$$

With $wz = w_1 u_1 x_1 v_1 yz$ and $\hat{w}(z + z') = \hat{w}_1 u_1 x_1 v_1 y(z + z')$, the system generates all strings in $w(z + z')$ as follows:

$$w_1 \underline{u_1 x_1 v_1} yz \not\rightarrow \hat{w}_1 \underline{u_1 x_1 v_1} y(z + z') = w_1 u_1 x_1 v_1 y(z + z') = w(z + z').$$

Likewise, with $w'z = w'_1 u_2 x_2 v_2 y'z$ and $\hat{w}'(z + z') = \hat{w}'_1 u_2 x_2 v_2 y'(z + z')$ the system produces $w'(z + z')$ as follows:

$$w'_1 u_2 x_2 v_2 y' z \not\rightarrow \hat{w}'_1 u_2 x_2 v_2 y'(z + z') = w'_1 u_2 x_2 v_2 y(z + z') = w'(z + z').$$

It follows that for all $w \in F_p$, $w' \in F_q$, $z \in T_p$, and $z' \in T_q$,

$$(w + w')(z + z') = (w + w')Y_{01}(z + z') \subseteq L(S).$$

Now, suppose that $(w + w')Y_{i1}(z + z') \subseteq L(S)$, for all i, $0 \leq i < k$. If $y_{i+1} \in I_{pq}$, clearly, $wy_{i+1} \in F_q$ and hence, by Lemma 4.2 there exists $\hat{w}' \in F_q$ such that $uxv \in SF(\hat{w}')$, $wy_{i+1} = w_{i+1}u_1x_1v_1y$ and $\hat{w}' = \hat{w}'_1u_1x_1v_1y$, for some $w_{i+1}, \hat{w}'_1, y \in A^*$ and a site $u_1x_1v_1$. Since $wy_{i+1}z' \in L(G)$, by the hypothesis of the lemma we have $wy_{i+1}z' \in L(S)$. By the induction hypothesis we have $\hat{w}'Y_{i1}(z + z') \subseteq L(S)$. With $wy_{i+1}z' = w_{i+1}u_1x_1v_1yz'$ and $\hat{w}'Y_{i1}(z + z') = \hat{w}'_1u_1x_1v_1yY_{i1}(z + z')$ the system generates all strings in $wY_{(i+1)1}(z + z')$ as follows:

$$\begin{aligned} &w_{i+1}\underline{u_1x_1v_1}yz' \not\rightarrow \hat{w}'_1\underline{u_1x_1v_1}yY_{i1}(z + z') \\ &\quad = w_{i+1}u_1x_1v_1yY_{i1}(z + z') = wy_{i+1}Y_{i1}(z + z') = wY_{(i+1)1}(z + z'). \end{aligned}$$

If $y_{i+1} \in I_{qp}$, then $w'y_{i+1} \in F_p$ and $w'y_{i+1}z \in L(G)$. Applying the same argument above with the roles of p and q (and w and w') interchanged, we can show that the system generates $w'Y_{(i+1)1}(z + z')$. It follows that

$$(w + w')Y_{(i+1)1}(z + z') \subseteq L(S),$$

which implies that $(w + w')Y_{k1}(z + z') \subseteq L(S)$ for all $k \geq 0$. □

Lemma 5.4 is concerned with one merging operation in step I(5) of the algorithm. Obviously, we can extend the lemma for a sequence of merging operations as follows.

LEMMA 5.5. *Let G' be the resulting graph when step* I *of algorithm SPLICE completes its merging operation on G for all sites. If $G \subseteq L(S)$, then $L(G') \subseteq L(S)$.*

LEMMA 5.6. *For two distinct patterns of the same hand (u, x, v) and $(u'xv')$, which have the same crossing, let $p \in Q(ux, v)$ and $q = MQ(u'xv')$ in a graph G. Let G' be the graph that is constructed from G by linking a path from p to q with span v' in step* II *of algorithm SPLICE. If $L(G) \subseteq L(S)$, then $L(G') \subseteq L(S)$.*

Proof. Let $r = MQ(uxv)$. Figure 10 illustrates the effect of the linking operation. We first consider the case $I_{qp} = \emptyset$. Clearly, adding p to q path with span v' in G will result in adding the set $F_pv'T_q$ to $L(G)$. Let $w \in F_p$ and $z' \in T_q$. We will show that $wv'z' \in L(S)$. Let $z \in T_r$ and $\hat{w}' \in F_q$ such that $\hat{w}' = \hat{w'}_0u'xv'$. Notice that, since q is a merged state, such $\hat{w'}$ exists by Lemma 4.2. Since $wvz, \hat{w}'z' \in L(G)$, by the hypothesis of the lemma we have $wvz, \hat{w}'z' \in L(S)$. Since $p \in Q(ux, v)$, by Lemma 4.3, there exists $\hat{w} \in F_p$ such that $ux \in SF(\hat{w})$ (i.e., $\hat{w} = w_0ux$ for some $w_0 \in A^*$), $w = w_1u_1x_1v_1y$ and $\hat{w} = \hat{w}_1u_1x_1v_1y$ for some $w_1, \hat{w}_1, y \in A^*$ and a site $u_1x_1v_1$. Since $\hat{w}vz \in L(G)$, we have $\hat{w}vz \in L(S)$.

With $\hat{w}vz = \hat{w}_0uxvz$ and $\hat{w}'z' = \hat{w}'_0u'xv'z'$ the system produces $\hat{w}v'z'$ as follows:

$$\hat{w}_0\underline{uxv}z \not\rightarrow \hat{w}'_0\underline{u'xv'}z' = \hat{w}_0uxv'z' = \hat{w}v'z'.$$

With $\hat{w}v'z' = \hat{w}_1u_1x_1v_1yv'z'$ and $wz = w_1u_1x_1v_1yz$, the system generates $w_1u_1x_1v_1yv'z' = wv'z'$ as follows:

$$w_1\underline{u_1x_1v_1}yz \not\rightarrow \hat{w}_1\underline{u_1x_1v_1}yv'z' = w_1u_1x_1v_1yv'z' = wv'z'.$$

It follows that $F_pv'T_q \subseteq L(S)$, which implies $L(G') \subseteq L(S)$.

Now, suppose that $I_{qp} \neq \emptyset$. Adding a p to q link induces a cycle as Figure 10 shows. For the proof we examine how the set of strings in $F_pT_p + F_qT_q$ will be affected

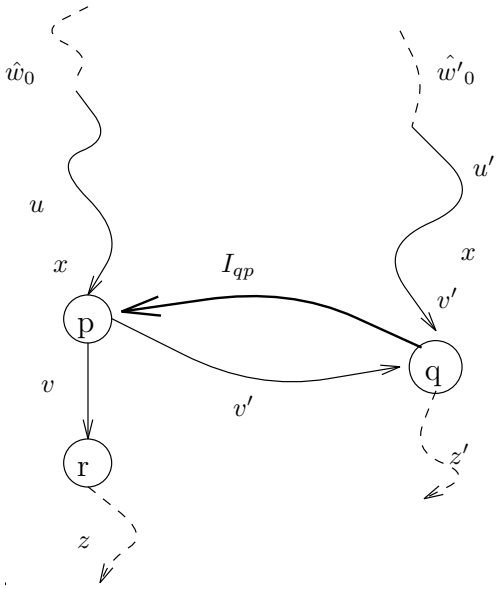


FIG. 10. *Linking from p to q with span v′.*

when the algorithm adds p to q path with span v'. Notice that F_pT_p is the set of accepting spans in G that start with a string in F_p, and F_qT_q is the set of accepting spans that start with a string in F_q. The set of spans in G' that start from q and end in an accepting state is

$$T_q + I_{qp}(v'I_{qp})^*(vT_r + (v' + I_{pq})T_q).$$

Notice that $I_{qp}(v'I_{qp})^*$ denotes all spans that start from q and end in p. The second term of the above expression denotes the set of strings that start with a span in I_{qp} and end with a span in either in T_r or T_q. Let

$$R = I_{qp}(v'I_{qp})^*(vT_r + (v' + I_{pq})T_q).$$

In $L(G')$, the set of accepting spans which start with a string in F_q is $F_q(T_q + R)$, and the set of accepting spans which start with a string in F_p is $F_p(T_p + (I_{pq} + v')R)$. So, for the proof of the lemma it is enough to show that

$$F_q(T_q + R) + F_p(T_p + (I_{pq} + v')R \subseteq L(S).$$

Since $F_qT_q + F_pT_p \subseteq L(S)$ by the hypothesis of the lemma, we only need to show that

$$F_qR + F_p(I_{pq} + v')R \subseteq L(S).$$

We show this in two parts: $F_qR \subseteq L(S)$ and $F_p(I_{pq} + v')R \subseteq L(S)$.

Part I. Proof of $F_qR = F_qI_{qp}(v'I_{qp})^*(vT_r + (v' + I_{pq})T_q) \subseteq L(S)$. Let $w' \in F_q$, $y_0 \in I_{qp}$, $z \in T_r$, $z' \in T_q$ and $t \in v' + I_{pq}$. Let $Y_{k1} = v'y_kv'y_{k-1} \dots v'y_1 \in (v'I_{qp})^*$, $k \geq 0$, where $Y_{ii} = v'y_i$ and $Y_{0i} = \epsilon$, for all i, $1 \leq i \leq k$. For the proof it is enough to show the following:

$$w'y_0Y_{k1}(vz + tz') \subseteq L(S).$$

Since $w'y_0vz \in L(G)$, by the hypothesis of the lemma we have $w'y_0vz \in L(S)$. Since $w'y_0 \in F_p$ and $p \in Q(ux, v)$, by Lemma 4.3 there exists $\hat{w} \in F_p$ such that $ux \in SF(\hat{w})$, $\hat{w} = \hat{w}_1u_1x_1v_1y$, and $w'y_0 = w'_1u_1x_1v_1y$ for some $\hat{w}_1, w'_1, y \in A^*$ and a site $u_1x_1v_1$. Since $\hat{w}vz \in L(G)$, we have $\hat{w}vz \in L(S)$. Find $\hat{w}' \in F_q$ such that $u'xv' \in SF(\hat{w}')$. Since $\hat{w}'z' \in L(G)$, we have $\hat{w}'z' \in L(S)$ by the hypothesis of the lemma.

Splicing $\hat{w}vz = \hat{w}_0uxvz$ and $\hat{w}'z' = \hat{w}'_0u'xv'z'$ the system generates $\hat{w}v'z'$ as follows:

$$\hat{w}_0\underline{uxv}z \not\rightarrow \hat{w}'_0\underline{u'xv'}z' = \hat{w}_0uxv'z' = \hat{w}v'z'.$$

Let $y' \in I_{pq}$. Then $\hat{w}y'z' \in L(G)$ and by the hypothesis of the lemma $\hat{w}y'z' \in L(S)$. For an arbitrary $t \in v' + I_{pq}$, we know that $\hat{w}tz' \in L(S)$. With $\hat{w}tz' = \hat{w}_1u_1x_1v_1ytz'$ and $w'y_0vz = w'_1u_1x_1v_1yvz$, the system generates $w'y_0tz'$ as follows:

$$w'_1u_1x_1v_1yvz \not\rightarrow \hat{w}_1u_1x_1v_1ytz' = w'_1u_1x_1v_1ytz' = w'y_0tz'.$$

Since $w'y_0vz \in L(S)$, we have proved that

$$w'y_0(vz + tz') = w'y_0Y_{01}(vz + tz') \subseteq L(S).$$

Now, suppose that $w'y_0Y_{i1}(vz+tz') \subseteq L(S)$, for some i, $0 \leq i < k$. For the proof of Part I, we will show that $w'y_0Y_{(i+1)1}(vz + tz') \subseteq L(S)$.

Since y_0 is an arbitrary string in I_{qp}, obviously, $w'y_{i+1}Y_{i1}(vz+tz') \subseteq L(S)$. Since $w'y_0 \in F_p$, by Lemma 4.3 there exists $\hat{w} \in F_p$ such that $ux \in SF(\hat{w})$, $\hat{w} = \hat{w}_1u_1x_1v_1y$, and $w'y_0 = w'_1u_1x_1v_1y$ for some $\hat{w}_1, w'_1, y \in A^*$ and a site $u_1x_1v_1$.

Let $\hat{w}'$ be a string in F_q such that $u'xv' \in SF(\hat{w}')$. Clearly, $\hat{w}'y_{i+1}Y_{i1}(vz+tz') \subseteq L(S)$. Since $\hat{w}vz \in L(G)$, by the hypothesis of the lemma we have $\hat{w}vz \in L(S)$. Since $\hat{w}vz = \hat{w}_0uxvz$ and $\hat{w}'y_{i+1}Y_{i1}(vz + tz') = \hat{w}'_0u'xv'y_{i+1}Y_{i1}(vz + tz')$, the system produces $\hat{w}v'y_{i+1}Y_{i1}(vz + tz')$ by the following operation:

$$\hat{w}_0\underline{uxv}z \not\rightarrow \hat{w}'_0\underline{u'xv'}y_{i+1}Y_{i1}(vz+tz') = \hat{w}_0uxv'y_{i+1}Y_{i1}(vz+tz') = \hat{w}v'y_{i+1}Y_{i1}(vz+tz').$$

With $w'y_0vz = w'_1u_1x_1v_1yvz$ and $\hat{w}v'y_{i+1}Y_{i1}(vz+tz') = \hat{w}_1u_1x_1v_1yv'y_{i+1}Y_{i1}(vz+tz')$, finally the system produces $w'y_0Y_{(i+1)1}(vz + tz')$ by the following splicing operation:

$$w'_1\underline{u_1x_1v_1}yvz \not\rightarrow \hat{w}_1\underline{u_1x_1v_1}yv'y_{i+1}Y_{i1}(vz + tz')$$

$$= w'_1u_1x_1v_1yv'y_{i+1}Y_{i1}(vz + tz') = w'y_0Y_{(i+1)1}(vz + tz').$$

It follows that $w'y_0Y_{k1}(vz + tz') \subseteq L(S)$, for all $w' \in F_q$, $y_0 \in I_{qp}$, $z \in T_r$, $z' \in T_q$, $t \in v' + I_{pq}$, and $Y_{k1} \in (v'I_{qp})^*$. This implies that $F_qR \subseteq L(S)$.

Part II. Proof of $F_p(I_{pq}+v')R \subseteq L(S)$. Obviously, $F_pI_{pq} \subseteq F_q$. Since $F_qR \subseteq L(S)$ from Part I of the proof, we have $F_pI_{pq}R \subseteq L(S)$. Hence, for the proof it is enough to show that $F_pv'R \subseteq L(S)$. Let $w \in F_p$. We will show that $wv'R \subseteq L(S)$. Since $wvz \in L(G)$, we have $wvz \in L(S)$ by the hypothesis of the lemma. Find $\hat{w}' \in F_q$ such that $u'xv' \in SF(\hat{w}')$. From Part I we know that $\hat{w}'R \subseteq L(S)$.

If $ux \in SF(w)$, then since $wvz = w_0uxvz$ and $\hat{w}'R = \hat{w}'_0u'xv'R$, the system produces $w_0uxv'R = wv'R$ by

$$w_0\underline{uxv}z \not\rightarrow \hat{w}'_0\underline{u'xv'}R = w_0uxv'R = wv'R.$$

If $ux \notin SF(w)$, then by Lemma 4.3 there exists $\hat{w} \in F_p$ such that $ux \in SF(\hat{w})$, $\hat{w} = \hat{w}_1 u_1 x_1 v_1 y$, and $w = w_1 u_1 x_1 v_1 y$ for some $\hat{w}_1, w_1, y \in A^*$ and a site $u_1 x_1 v_1$. Since $\hat{w}vz \in L(G)$, we have $\hat{w}vz \in L(S)$. Since $\hat{w}vz = \hat{w}_0 uxvz$, the system produces all strings in $\hat{w}v'R$ as follows:

$$\hat{w}_0 \underline{uxv} z \vdash \hat{w}'_0 \underline{u'xv'} R = \hat{w}_0 uxv'R = \hat{w}v'R.$$

Finally, since $\hat{w}v'R = \hat{w}_1 u_1 x_1 v_1 y v'R$ and $wvz = w_1 u_1 x_1 v_1 y vz$, the system generates all strings in $wv'R$ by the following operation:

$$w_1 \underline{u_1 x_1 v_1} yvz \vdash \hat{w}_1 \underline{u_1 x_1 v_1} yv'R = w_1 u_1 x_1 v_1 yv'R = wv'R.$$

It follows that $F_p v'R \subseteq L(S)$. □

Lemma 5.6 is concerned with one linking operation in step II. As for the merging operation, we can extend this lemma for a sequence of linking operations.

LEMMA 5.7. *Let G' be the resulting graph processed by step* II *of algorithm SPLICE with graph G. If $L(G) \subseteq L(S)$, then $L(G') \subseteq L(S)$.*

Step III, which merges links, actually merges states that will remain equivalent throughout the computation as it was shown by Lemma 4.5. Since Step III does not change the language, we have the following.

LEMMA 5.8. *Let G' be the resulting graph processed by step* III *of algorithm SPLICE with graph G. If $L(G) \subseteq L(S)$, then $L(G') \subseteq L(S)$.*

Now, we are ready to present the two lemmas that prove Theorem 5.1.

LEMMA 5.9. *Let G be the output graph from algorithm CONSTRUCT for a splicing system $S = (A, I, B, C)$ with I given in terms of the state transition graph of an automaton whose language is I. Then $L(G) \subseteq L(S)$.*

Proof. By Lemma 4.4, if w is in $L(G_i)$, it is also in $L(G_{i+1})$. Clearly, $L(G_0) = I$. Since by definition $I \subseteq L(S)$, we have $L(G_0) \subseteq L(S)$. By Theorem 4.1 algorithm *CONSTRUCT* terminates after some finite nth iteration. Thus we have $G = G_n$. Suppose that $L(G_i) \subseteq L(S)$ for all $i < n$. By Lemmas 5.5, 5.7, and 5.8 we know that $L(G_{i+1}) \subseteq L(S)$, which implies that $L(G) \subseteq L(S)$. □

LEMMA 5.10. *$L(S) \subseteq L(G)$ for the same G and S of Lemma* 5.9.

Proof. Suppose $L(S) - L(G) \neq \emptyset$. Since $L(S)$ is spliced starting with I and $I \subseteq L(S) \bigcap L(G)$, there should be a string $w \in L(S) - L(G)$ that is generated by splicing with some strings in $L(S) \bigcap L(G)$. Let $w = w_0 x_1 w_1 x_2 w_2 \dots x_k w_k$, for some $k \geq 1$, such that x_i, $1 \leq i \leq k$ is a crossing of a pattern. (Notice that x_i is a crossing that is not necessarily embedded in a site.) This implies that $L(G)$ should have strings $w_0 x_1 z_0$, $y_1 x_1 w_1 x_2 z_1$, $y_2 x_2 w_2 x_3 z_2, \dots, y_k x_k w_k$, for some $y_i, z_j \in A^*$, $1 \leq i \leq k$, $0 \leq j \leq k-1$, such that, for each i, the two x_i's that appear in each pair of adjacent strings in the sequence are crossings which are embedded in sites of the same hand. Let $u_i x_i v_i$ and $u'_i x_i v'_i$ be those two sites that have crossing x_i.

Since $w_0 x_1 z_0, y_1 x_1 w_1 x_2 z_1 \in L(G)$, algorithm *SPLICE* should have added an accepting span $w_0 x_1 w_1 x_2 z_1$ in step I or II of the algorithm when it processed the site whose crossing is x_1. Since the graph has two spans $w_0 x_1 w_1 x_2 z_1$ and $y_2 x_2 w_2 x_3 z_2$, the algorithm should have added an accepting span $w_0 x_1 w_1 x_2 w_2 x_3 z_2$ in step I or step II of the algorithm when it processed crossing x_2, and so on. Finally, the algorithm should have added an accepting span $w_0 x_1 w_1 x_2 w_2 \dots x_k w_k$, which is w. It follows that $w \in L(G)$, a contradiction. □

In [5], it was shown that if I is finite set, $L(S)$ is strictly locally testable. We prove the same result by showing that the reduced deterministic version of M does

not have two nondead states p and q such that $\delta(p,w)=p$ and $\delta(q,w)=q$, for any $w\in A^+$, which is a necessary and sufficient condition that a reduced deterministic finite automaton must satisfy to be strictly locally testable [6]. For the proof we need the following lemma.

LEMMA 5.11. *Let y be a span of a cycle in G_i. If the system S is persistent and $|I|$ is finite, there is a constant c such that, for all $k\geq c$, string y^k contains a site.*

Proof. If $|I|$ is finite, graph G_0 has no cycle. Clearly, during the computation a new cycle will be introduced either by merging, linking, or link collapsing. More specifically, a cycle will be introduced only when there exist two states p and q such that p is an ancestor of q and either one of the following conditions is satisfied. Let uxv and $u'xv'$ be sites of the same hand and let k be the length of the longest site.

(1) $p,q\in Q(uxv)$.
(2) $p=MQ(u'xv')$ and $q\in Q(ux,v)$.
(3) $p,q\in Q(ux,v)$ such that $IMS(p,k)=IMS(q,k)$ and there exists $r=MQ(u'xv')$.

If condition (1) is satisfied, a cycle will be introduced when p and q are merged in step I of the algorithm. If condition (2) is satisfied, step II will create a cycle when it constructs a link from q to p with span v'. If condition (3) is satisfied, step III will create a cycle when it collapses the two links from p to r and from q to r, both with span v'.

Case (1). $p,q\in Q(uxv)$. Let $y_0\in I_{pq}$. Step I of the algorithm merges p and q and creates a cyclic with span y_0. Let $w_3\in F_p$. Since $p\ inQ(uxv)$, by Lemma 4.2 there exists $w_2\in F_p$ such that $w_2=w_{21}uxv=w_{22}u_2x_2v_2y_2$ and $w_3=w_{32}u_2x_2v_2y_2$. Since $w_3y_0\in F_q$, again by Lemma 4.2 there exists $w_1\in F_q$ such that $w_1=w_{11}uxv=w_{12}u_1x_1v_1y_1$ and $w_3y_0=w_{31}u_1x_1v_1y_1$. Let $z\in T_q$. We have three accepting spans w_1z, w_2y_0z, and w_3y_0z that have the following properties.

$$w_1z=w_{11}uxvz=w_{12}u_1x_1v_1y_1z,$$
$$w_2y_0z=w_{21}uxvy_0z=w_{22}u_2x_2v_2y_2y_0z, \text{ and}$$
$$w_3y_0z=w_{31}u_1x_1v_1y_1z=w_{31}u_2x_2v_2y_2y_0z.$$

By Theorem 5.1, $w_1z,w_2y_0z,w_3y_0z\in L(S)$. By Lemmas 2.7 and 2.9, there exists a constant c such that for all $k\geq c$, the string $(y_0)^k$ has a site.

Case (2). $p=MQ(u'xv')$ and $q\in Q(ux,v)$. Let $y_0\in I_{pq}$. Linking q to p with span v' in step II, the algorithm creates a cycle with span $v'y_0$. Let $w_3\in F_p$. Since $p\in Q(u'xv')$, by Lemma 4.2 there exists $w_2\in F_p$ such that $w_2=w_{21}u'xv'=w_{22}u_2x_2v_2y_2$ and $w_3=w_{32}u_2x_2v_2y_2$. Since $w_3y_0\in F_q$, by Lemma 4.3 there exists $w_1\in F_q$ such that $w_1=w_{11}ux=w_{12}u_1x_1v_1y_1$ and $w_3y_0=w_{31}u_1x_1v_1y_1$. Let $r=MQ(uxv)$ and $r\in T_r$. We have accepting spans w_1vz, w_2y_0vz, and w_3y_0vz that have the following property:

$$w_1vz=w_{11}uxvz=w_{12}u_1x_1v_1y_1vz,$$
$$w_2y_0vz=w_{21}u'xv'y_0vz=w_{22}u_2x_2v_2y_2y_0vz, \text{ and}$$
$$w_3y_0vz=w_{31}u_1x_1v_1y_1vz=w_{31}u_2x_2v_2y_2y_0vz.$$

By Theorem 5.1, $w_1vz,w_2y_0vz,w_3y_0vz\in L(S)$, and by Lemmas 2.10 and 2.12 there exists a constant c such that for all $k\geq c$, the string $(v'y_0)^k$ contains a site.

Case (3). $p,q\in Q(ux,v)$ such that $IMS(p,k)=IMS(q,k)$ and there exists $r=MQ(u'xv')$. Let s and t, respectively, be the first states of the links from p to r and from q to r with span v' that have been introduced by step II. Let a be the label on the q to t transition, which is the first symbol of v'. Clearly, p to s transition has the same label a. Let $w\in I_{sq}$. Then, collapsing states s and t introduces a loop with span aw. We will show that there is a constant c such that $(aw)^k$ contains a site for

all $k > c$. Obviously, $aw \in I_{pq}$. We will actually show that for any $y_0 \in I_{pq}$, string ${y_0}^k$ contains a site.

Let $w_1 \in F_p$. We have $w_1 y_0 \in F_q$. Suppose that q is not monomial w.r.t. ux. By Lemma 4.3, we have $w_1 y_0 = w_{12} u_1 x_1 v_1 y_1$ for some $w_{12}, y_1 \in A^*$ and a site $u_1 x_1 v_1$. (Notice that $|ux| > |u_1 x_1 v_1 y_1|$. Otherwise, q should be monomial *w.r.t.* ux.) Since $IMS(p,k) = IMS(q,k)$, state p is also not monomial *w.r.t.* ux and $w_1 = w_{11} u_1 x_1 v_1 y_1$. Let $z_1 \in T_q$. String $w_1 y_0 z_1$ is an accepting span, and we have $w_1 y_0 z_1 = w_{11} u_1 x_1 v_1 y_1 y_0 z_1 = w_{12} u_1 x_1 v_1 y_1 z_1$. Since $|y_0| \geq 1$, we have $u_1 x_1 v_1 y_1 y_0 = z u_1 x_1 v_1 y_1$ for some $z \in A^+$. By Theorem 5.1, $w_1 y_0 z_1 \in L(S)$, and by Corollary 2.8, $w_{11} u_1 x_1 v_1 y_1 (y_0)^+ z_1 \subseteq L(S)$. By Lemma 2.9, there exists a constant c such that for all $k \geq c$ the string $(y_0)^k$ has a site.

Now, suppose that q is monomial w.r.t. ux. We have $w_1 y_0 v \in F_r$. Let $z_1 \in T_r$. Then $w_1 y_0 v z_1$ is an accepting span. Since $IMS(p,k) = IMS(q,k)$, state p is also monomial w.r.t. ux. Hence, we have $w_1 y_0 v' z_1 = w_{11} uxy_0 v' z_1 = w_{11} zuxv' z_1$ for some $z_1 \in A^+$. Since $uxy_0 = zux$, by Lemma 2.6 there is a constant c such that ${y_0}^k$ contains ux, for all $k \geq c$. By the property of persistence there should be a constant $c' \geq c$ such that for all $k \geq c'$, ${y_0}^k$ has a site. ∎

Now, we are ready to prove Theorem 5.3.

Proof. Let G be the output of algorithm *CONSTRUCT*. To prove that $L(G)$ is strictly locally testable, we show that the reduced deterministic version of G satisfies Theorem 2.5. Suppose that the graph has a pair of states p and q such that $\delta(p,w) = p$ and $\delta(q,w) = q$ for some $y \in A^+$. Let C_p and C_q denote the cycles, respectively, corresponding to $\delta(p,w) = p$ and $\delta(q,w) = q$. By Lemma 5.11, there is a constant c such that for all $k \geq c$, the string w^k has a site, say uxv. This implies that C_p and C_q have states p' and q', respectively, such that $p', q' \in Q(uxv)$. States p' and q' should have been merged into a state $r = MQ(uxv)$ by step I(5) of algorithm *SPLICE*. State r belongs to both of the cycles C_p and C_q, and $r \in \delta(r, w_2 w_1)$, for some w_1 and w_2 such that $w = w_1 w_2$. By Lemma 3.8 the language $L(G)$ is $LTSS$. ∎

6. Concluding remarks. We have introduced an algorithm, which, given a splicing system with its initial set of strings given in terms of the state transition graph of an automaton that recognizes the set, constructs an automaton that recognizes the language generated by the system. This solves the open problem in [2]. With the construction we could show that if the system is persistent, the splicing language is $LTSS$. This result also shows a constructive proof of regularity of splicing languages when I is regular, which was proven in [1] by using an algebraic system called alphabetic dominos.

The algorithm in [2] works for permanent splicing systems. Recently, one of the reviewers commented that this algorithm had been extended to the class of so-called twist-free splicing systems [3]. A pair of distinct patterns (u,x,v) and (w,y,z) are twisted if there is another pair of patterns (u',x,v') and (w',y,z') such that either one of the following conditions are satisfied.

(a) For some $u_1 \in A^+$ such that $ux = u_1 u_2$, either one of u_1 and w' is suffix of the other, and either one of u_2 and z is prefix of the other.

(b) For some $z_2 \in A^+$ such that $z = z_1 z_2$, either one of z_2 and v' is prefix of the other, and either one of z_1 and ux is suffix of the other.

A splicing system is twist free if it has no pair of twisted patterns. Twist freeness implies both permanence and persistence. If a splicing system has a pair of twisted patterns, a site can be destroyed. The splicing system in Figure 8 has one twisted pair (ba, b, a) and (b, a, a). Our algorithm solves the problem of twisted patterns by

monomializing the graph as shown in the figure (see the split states highlighted).

We can think of multihanded splicing systems $S = (A, I, H_1, H_2, \ldots, H_n)$, a generalization of the two-handed splicing model. Our algorithm and the main results can be easily extended to such a generalization.

The automata that our algorithm constructs are nondeterministic like the other algorithms introduced in [2] and [3]. It would be challenging to develop a practical algorithm that can construct a deterministic automaton for a given splicing system. There is considerable room for improvement in our algorithm. For the improvement we may note that, in biology, sites are almost always 4, 6, or 8 bps long, and most sites consist of reverse palindromes. Can we use these properties for more efficient construction? Is the algorithm practical for such data? Can we improve it?

Appendix A.

procedure $Monomialize(G, x)$
(//This algorithm monomializes the finite state transition graph G of an automaton $M = (Q, A, \delta, q_{st}, F)$ w.r.t. string $x \in A^+$. It is assumed that the automaton, which is nondeterministic, has no ϵ transition.

The algorithm uses two subroutines $Split1(G, q, q_1, q_2, c)$ and $Split2(G, q, q_1, q_2, c)$. Given a state q which is not monomial w.r.t. symbol c, algorithm $Split1(G, q, q_1, q_2, c)$, splits q into two equivalent states q_1 and q_2 and makes them monomial w.r.t. symbol c such that $IMS(q_1, 1) = \{c\}$ and $c \notin IMS(q_2, 1)$. Given G which is monomial w.r.t. a string $x \in A^+$ and a state q which is not monomial w.r.t. xc, $Split2(G, q, q_1, q_2, c)$ splits q into two equivalent states q_1 and q_2 and monomializes them w.r.t. string xc such that $IMS(q_1, |xc|) = \{xc\}$ and $xc \notin IMS(q_2, |xc|)$. Statements 5 and 11 are for algorithm $SPLICE$ in Appendix B to block splitting transitions marked as "merged." //)

begin

1. Let $x = a_1 a_2 \ldots a_n$;
2. **for** each state q such that $q \in \delta(p, a_1)$, for some $p \in Q$, **do**
3. **if** there exits $b \neq a_1$ such that $q \in \delta(r, b)$, for some $r \in Q$, **then**
   begin
4. $Split1(G, q, q_1, q_2, a_1)$ and mark q_1 with "mon";
5. **if** there is a transition $q_1 \in \delta(p, a_1)$, $p \in Q$, which has mark "merged"
   then change all transitions $q_2 \in \delta(r, b)$, $b \in A$, $r \in Q$,
   that have mark "merged" to $q_1 \in \delta(r, b)$;
   end
6. **else** mark q with "mon";
7. **for** $i = 2$ to n **do** (// Monomialize G w.r.t. string $a_1 a_2 \ldots a_i$. //)
   begin
8. **for** each $q \in Q$ such that $q \in \delta(p, a_i)$, for some p which has mark "mon,"
   do
9. **if** there exists a transition $q \in \delta(r, b)$ such that
   either r has no mark "mon" or $b \neq a_i$ **then**
   begin
10. $Split2(G, q, q_1, q_2, a_i)$ and mark q_1 with $\hat{m}$;
11. **if** there is a transition $q_1 \in \delta(p, a_1)$, $p \in Q$, which has mark "merged"
    then change all transitions $q_2 \in \delta(r, b)$, $b \in A$, $r \in Q$
    that have mark "merged" to $q_1 \in \delta(r, b)$;

end
12. **else** mark q with $\hat{m}$;
13. Erase marks "mon," if any, from all states and then change marks $\hat{m}$ to "mon"; **end**;
14. Erase marks "mon," if any, from all states;
end;

procedure $Split1(G, q, q_1, q_2, c)$
(// This procedure splits state q into two equivalent states q_1 and q_2 and monomializes both of them w.r.t. symbol c such that $IMS(q_1, 1) = \{c\}$ and $c \notin IMS(q_2, 1)$. //)
begin
1. Introduce new states q_1 and q_2;
2. Let q_1 and q_2 be accepting states if state q is;
3. **if** $q = q_{st}$ **then** let q_2 be the start state q_{st};
4. **for** every looping transition $q \in \delta(q, a)$, $a \in A$, **do**
5. **if** $a = c$ **then** let $q_1 \in \delta(q_1, a)$ and $q_1 \in \delta(q_2, a)$
6. **else** let $q_2 \in \delta(q_2, a)$ and $q_2 \in \delta(q_1, c)$;
7. **for** every transition $q \in \delta(p, a)$, for $a \in A$ and state $p \neq q$ **do**
8. **if** $a = c$ **then** let $q_1 \in \delta(p, a)$
9. **else** let $q_2 \in \delta(p, a)$;
10. **for** every transition $s \in \delta(q, a)$, for $a \in A$ and state $s \neq q$ **do**
11. let $s \in \delta(q_1, a)$ and $s \in \delta(q_2, a)$;
12. Delete q and all its incoming and outgoing transitions;
end;

procedure $Split2(G, q, q_1, q_2, c)$
(// This procedure splits q into two equivalent states q_1 and q_2 and monomializes them w.r.t. xc if the state transition graph G is monomial w.r.t. a nonnull string x. Assume that every state p is marked with "mon" if $IMS(p, |x|) = \{x\}$ and state q is not monomial w.r.t. string xc. //)
begin
1. Introduce new states q_1 and q_2;
2. Let q_1 and q_2 be accepting states if q is;
3. **for** every looping transition $q \in \delta(q, a)$, $a \in A$, **do**
4. let $q_2 \in \delta(q_2, a)$ and $q_2 \in \delta(q_1, a)$;
5. **for** every transition $q \in \delta(p, a)$, for $a \in A$ and state $p \neq q$ **do**
6. **if** $a = c$ and p has mark "mon" **then** let $q_1 \in \delta(p, a)$
7. **else** let $q_2 \in \delta(p, a)$;
8. **for** every transition $s \in \delta(q, a)$, for $a \in A$ and state $s \neq q$ **do**
9. let $s \in \delta(q_1, a)$ and $s \in \delta(q_2, a)$;
10. Delete q and all its incoming and outgoing transitions;
end;

Appendix B.

procedure $CONSTRUCT(S)$
(// $S = (A, I, B, C)$ is a splicing system, where I is given in terms of the state transition graph G of an automaton whose language is I. Using subroutine $SPLICE$, this algorithm transforms G to G' such that $L(G') = L(S)$. //)

begin
  Find the length k of the longest pattern;
  repeat
    $SPLICE(G, B)$; $SPLICE(G, C)$;
  until (G is not changed);
end.

procedure $SPLICE(G, X)$
(//Given a set of strings in terms of the state transition graph G, this algorithm transforms G to G' such that $L(G')$ is exactly the set of strings generated by splicing $L(G)$ with pattern X. //)
begin
  repeat
I. **for** each pattern $(u, x, v) \in X$ **do** (// Merge states in $Q(uxv)$.//)
    begin
1. Compute IMS of length $|uxv|$ of all states;
2. $Monomialize(G, uxv)$;
3. Construct the sets $Q(uxv)$ and $Q(ux, v)$;
4. **If** $|Q(uxv)| > 1$ **then**
5. merge the set $Q(uxv)$ into one state and mark all the converged transitions as "merged";
    end;
II. **for** each pattern $(u, x, v) \in X$ **do** (// Put links. //)
    begin
1. **if** the set $Q(uxv)$ is not empty **then**
2. **for** each pattern $(u', x, v') \in X$ such that $u \neq u'$ or $v \neq v'$ **do**
      add a path with span v' from each $r \in Q(ux, v)$ to the state $MQ(u'xv')$;
    end;
III **for** each pattern (u, x, v) **do** (// Merge links.//)
    for each pair $p, q \in Q(ux, v)$ **do**
      if $IMS(p, k) = IMS(q, k)$ **then** merge the links from p to r and from q to r, for every merged state $r = MQ(u'xv')$ with $u \neq u'$ or $v \neq v'$;
  until (no states are merged in Sep I);
end;

Acknowledgments. The author thanks Robert McNaughton for many fruitful discussions and suggestions. I am indebted to the reviewers for their helpful recommendations, especially to the one who suggested the simple proofs for Lemmas 2.9 and 2.12.

REFERENCES

[1] K. Culik II and T. Harju, *Splicing semigroups of dominos and DNA*, Discrete Appl. Math., 31 (1991), pp. 261–277.
[2] R. W. Gatterdam, *Algorithms for splicing systems*, SIAM J. Comput., 21 (1992), pp. 507–520.
[3] R. W. Gatterdam, *DNA and Twist Free Splicing Systems,* personal communication, 1995.
[4] M. A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.
[5] T. Head, *Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors*, Bull. Math. Biol., 49 (1987), pp. 737–759.

[6] S. M. Kim and R. McCloskey, *A characterization of constant-time cellular automata computation*, Phys. D, 45 (1990), pp. 404–419.
[7] R. Martin, *Studies in Feedback-Shift-Register Synthesis of Sequential Machines*, M.I.T. Press, Cambridge, MA, 1969.
[8] R. McNaughton and S. Papert, *Counter-Free Automata*, M.I.T. Press, Cambridge, MA, 1971.
[9] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

SIAM J. COMPUT
Vol. 26, No. 5, pp. 1310–1342, October 1997

003

FAST MANAGEMENT OF PERMUTATION GROUPS I*

LÁSZLÓ BABAI†, EUGENE M. LUKS‡, AND ÁKOS SERESS§

Abstract. We present new algorithms for permutation group manipulation. Our methods result in an improvement of nearly an order of magnitude in the worst-case analysis for the fundamental problems of finding strong generating sets and testing membership. The normal structure of the group is brought into play even for such elementary issues. An essential element is the recognition of large alternating composition factors of the given group and subsequent extension of the permutation domain to display the natural action of these alternating groups. Further new features include a novel fast handling of alternating groups and the sifting of defining relations in order to link these and other analyzed factors with the rest of the group. The analysis of the algorithm depends on the classification of finite simple groups. In a sequel to this paper, using an enhancement of the present method, we shall achieve a further order of magnitude improvement.

Key words. permutation group algorithm, strong generating set

AMS subject classifications. 68Q40, 20B40

PII. S0097539794229417

1. Introduction.

Since the size of a permutation group G on n letters can be exponential in n, it is customary, for computational purposes, to specify G by a small list of generators. However, the succinctness of such a representation raises the issue of whether we can deal effectively with the groups that we can specify. Can one, for example, find the order of G and test membership in G without enumerating all of its elements?

In fact, in the late sixties, Sims developed efficient algorithms for permutation group manipulation [Si70]. These included the key notion of a *strong generating set* (SGS) which is the underlying concept in essentially all polynomial-time algorithms in computational group theory. Given a chain $G = G_0 \geq G_1 \geq \cdots \geq G_m = 1$ of subgroups of G, an SGS with respect to this chain is a set $T \subset G$ such that $T \cap G_i$ generates G_i for each i. Sims's algorithm uses the point stabilizer chain; that is, G_i is the pointwise stabilizer of the first i points of the permutation domain.

While Sims's methods for constructing an SGS have been widely used in computational group theory since their inception, the question of their asymptotic efficiency was not resolved until 1980. Furst, Hopcroft, and Luks [FHL] observed that a version of Sims's algorithm runs in polynomial time, namely $O(n^6 + sn^2)$ steps, where s is the number of generators given for G. Subsequently, Knuth [Kn] and Jerrum [Je82], [Je86] gave variants with running time $O(n^5 + sn^2)$. All of these algorithms rest on the most elementary group theory.

*Received by the editors October 28, 1994; accepted for publication (in revised form) September 20, 1995.

http://www.siam.org/journals/sicomp/26-5/22941.html

†Department of Algebra, Eötvös University, Muzeum krt. 6-8, Budapest H-1088, Hungary and Department of Computer Science, University of Chicago, 1100 East 58th Street, Chicago, IL 60637-1504 (laci@cs.uchicago.edu). The research of this author was partially supported by NSF grant CCR-9014562 and OTKA (Hungary) grant 2581.

‡Computer and Information Sciences Department, University of Oregon, Eugene, OR 97403 (luks@cs.uoregon.edu). The research of this author was partially supported by NSF grant CCR-9013410.

§Department of Mathematics, Ohio State University, 231 West 18th Street, Columbus, OH 43210 (akos@math.ohio-state.edu). The research of this author was partially supported by NSF grants CCR-9201303 and CCR-9503430.

Since Knuth's note of 1981 (a preliminary version of [Kn]) and Jerrum's 1982 paper, the $O(n^5)$ bound has achieved notoriety and is generally believed to be the best that can be obtained via Sims's approach alone (cf. Remark 2.13). The main result of this paper is the improvement of the worst case bound by nearly one order of magnitude.

THEOREM 1.1. *Given a permutation group G by a list S of generators, $|S| = s$, the following problems can be solved in $O(n^4 \log^{c_1} n + sn^2)$ time.*

(a) *Find a set of strong generators.*

(b) *Find the order of G.*

(c) *Test membership of any permutation in G. Additional tests cost only $O(n^2)$ each.*

(d) *Find the pointwise stabilizer of a subset of the permutation domain.*

(e) *Construct a generator-relation presentation $\langle X|R\rangle$ of G in which $|X| = O(n \log^{c_2} n)$ and $|R| = O(n^2 \log^{c_3} n)$.*

In order to avoid long timing expressions as in Theorem 1.1 and concentrate on the essential part of the improvements, we introduce a "soft version" of the big-O notation. For two functions $f(n), g(n)$, we write $f(n) = O^{\sim}(g(n))$ if $f(n) \leq Cg(n) \log^c n$ (c, C are positive constants). Thus the time bound for basic permutation group manipulation in Theorem 1.1 is $O^{\sim}(n^4 + sn^2)$. We do not try, at this time, to minimize the exponent of $\log n$. Straightforward estimates give $c_1 = 7, c_2 = 2$, and $c_3 = 4$.

The new algorithms are not merely improved versions of previous SGS constructions. All of those predecessors construct an SGS with respect to the chain of point stabilizer subgroups. A key departure from the traditional approach is the use of another sort of subgroup tower, one which is not easily observable solely in terms of the action on the original permutation domain. Its very specification subsumes knowledge of the group structure. We construct an SGS with respect to a subgroup chain $G = G_0 \geq G_1 \geq \cdots \geq G_m = 1$ such that each G_i is *normal* in G and the factor groups G_i/G_{i+1} are either products of isomorphic alternating groups or subgroups of products of small primitive groups ("small" in this context means of order $n^{c \log n}$).

Naive divide-and-conquer of the permutation domain provides some normal subgroups of G in the kernels of induced actions on orbits or blocks of imprimitivity; the new machinery comes into play precisely when such decomposition bottoms out. The structure of large primitive groups allows an augmentation of the domain that readmits naive decomposition. This idea is part of the NC-procedure developed for the same problem [BLS87]. However, the sequential algorithm cannot be viewed as the sequential implementation of the parallel one. Even a knowledgeable implementation of the relevant part of parallel ideas would require $O^{\sim}(n^6)$ at best.

The timing analysis depends on the classification of finite simple groups via information on the order of primitive permutation groups whose socle is not the product of alternating groups. We remark, however, that there is an *elementary* version of the algorithm breaking the $O(n^5)$ barrier. Instead of the classification, we may use Babai's bound [Ba] on the order of uniprimitive groups and Pyber's recent bound [Py] on the order of doubly transitive groups to obtain an $O^{\sim}(n^{4.5})$ algorithm. In fact, the elementary algorithm is simpler in the sense that we do not have to detect alternating groups in socles of primitive groups involved in G (unless the primitive group itself is alternating or symmetric in its natural action on blocks of imprimitivity of G). Both Babai's and Pyber's results are within a logarithmic factor (in the exponent) from optimal; the loss in running time is due to the fact that we do not have elementary

estimates for the order of primitive groups with nonalternating-type socle. We sketch the elementary version in section 9.

We mention two further aspects which are important differences from previous methods. Exploiting the normality of subgroups in the new subgroup chain, we first obtain only normal generators, i.e., generators whose normal closure is the given subgroup. Another difference is the novel handling of full symmetric and alternating groups. We formulate the latter result as a separate theorem.

THEOREM 1.2. *From a given list of generators of the symmetric or alternating group, one can construct an SGS with respect to the chain of point stabilizer subgroups in* $O^{\sim}(n^3 + sn^2)$ *time. (The term "construct" refers to the operations of taking products and powers of permutations.) Moreover, there is a Las Vegas algorithm achieving the same goal in* $O^{\sim}(n^2 + sn)$ *expected running time.*

A random algorithm is Las Vegas if it never returns incorrect answers. We require that the SGS is constructed from the given generators via permutation multiplications since we apply this result when the symmetric group is involved in a larger permutation group G and acts on blocks of imprimitivity of G. We can guarantee that a given permutation from the symmetric group belongs to G only if it is constructed by the aforementioned operations.

We remark that the *random subproduct method*, originally developed to prove the random part of Theorem 1.2, was substantially extended by Babai, Cooperman, Finkelstein, Luks, and Seress [BCFLS91], [BCFLS95] to yield an elementary Monte Carlo algorithm for the basic tasks mentioned in Theorem 1.1 which runs in $O(n^3 \log^4 n + sn \log n)$ time. (A Monte Carlo algorithm may return a wrong answer with a fixed but arbitrarily small probability.)

In a sequel to this paper, we shall extend our method to achieve a further order of magnitude improvement in the running time.

THEOREM 1.3 (see [BLS]). *Given a permutation group* G *by a list* S *of generators,* $|S| = s$*, the following problems can be solved in* $O^{\sim}(sn^3)$ *time:*

(a) *All items listed in Theorem* 1.1.

(b) *Finding a composition series of* G.

Let us remark that the length of the input is $\Theta(sn)$ so this is an $O^{\sim}(n^3)$ algorithm as a function of the input length. For the more complicated task of computing a composition series, Theorem 1.3 gives an improvement of *five* orders of magnitude from Luks's original algorithm [Lu87]. This result requires a deeper probe into the structure of primitive groups with different types of socle, in the spirit of the O'Nan–Scott theorem [Sc], [Cam].

Like the method of this paper, the $O^{\sim}(sn^3)$ algorithm examines the primitive groups involved in G and locates the large alternating composition factors. It differs in its handling of the nonalternating part of G and a reduction of the number of "normal generators" for consecutive groups in the normal series. Specifically, the arguments in sections 6 and 7 are improved. A part of these results appeared in [BLS93].

As presented in sections 3–8, our $O^{\sim}(n^4)$ algorithm requires $O^{\sim}(n^3 + sn^2)$ memory. In section 9 we indicate how to decrease the memory requirement to $O^{\sim}(n^2 + sn)$. Also, without loss in time efficiency, the algorithm can output Jerrum's compressed data structure [Je86] for an SGS with respect to the point stabilizer subgroup chain; this requires only $O(n^2)$ space and supports membership testing in $O(n^2)$ time per test.

At this point, our emphasis is on the theoretical improvement realized by our

algorithm. Practical computations often deal with so-called *small-base groups*, i.e., families of groups satisfying $\log|G| < \log^c n$ for some fixed constant c. For small-base group inputs, the traditional algorithms run in $O^{\sim}(n^2)$ time and our method becomes essentially a version of the traditional approach. The attention given to the small-base case is, in part, due to the fact that interesting permutation representations of the nonalternating simple groups tend to have a small base. However, it is also the case that it has often been impractical to deal with large-base groups. Thus, aspects of the new methods should be important in practice where there is a need to deal with groups where $\log|G|$ is, say, proportional to n and when hardware is improved to allow the usage of $\Theta(n^2)$ memory for n in the tens of thousands.

2. Definitions and preliminaries. We refer to any standard text, e.g. [Ha], for basic facts about groups. For permutation group concepts we refer to [Wi] and [Cam]. We mention two sources of information on the classification of finite simple groups [Go], [Car], but no knowledge of these works is required. Cameron [Cam] gives a fine survey of all the consequences of the simple groups classification relevant to our work.

2.1. Group theory. We write $H \leq G$ if H is a subgroup of G and $H \triangleleft G$ if H is a normal subgroup of G.

LEMMA 2.1 (see [Ha, p. 96]). *Let $H \leq G$ and assume S is a set of generators of G and R is a complete set of right coset representatives of G mod H. Then the set*

$$\{\rho\sigma\rho_1^{-1} : \rho, \rho_1 \in R, \sigma \in S, \rho\sigma\rho_1^{-1} \in H\}$$

generates H.

The generators described here are called *Schreier generators* of H; their number is $|S||G:H|$ (these are not necessarily distinct).

For $Q \subset G$, the *normal closure* $\langle Q^G\rangle$ of Q in G is the smallest normal subgroup of G containing Q. More generally, for $K \leq G$, $\langle Q^K\rangle$ is the smallest subgroup of G containing Q and normalized by K. We say Q is a set of *normal generators* for H if $H = \langle Q^G\rangle$. For $\tau, \sigma \in G$, τ^σ denotes the *conjugate* $\sigma^{-1}\tau\sigma$. A group $G \neq 1$ is called *simple* if it has no nontrivial normal subgroups. We call G *semisimple* if it is the direct product of simple groups. If these simple groups are isomorphic then G is *characteristically simple*. A *composition series* of G is any series $1 = G_r \triangleleft \cdots \triangleleft G_1 \triangleleft G_0 = G$ where the quotients G_{i-1}/G_i are simple; these quotients are the *composition factors*. The group G is *solvable* if all composition factors of G are cyclic. We need the following well-known fact (see, e.g., [Sc]).

PROPOSITION 2.2. *Let H be a subgroup of the semisimple group $G = \prod_{i=1}^m T_i$ such that all T_i are simple nonabelian and H projects onto each factor. Then H is direct product of "diagonal" subgroups; more precisely, the T_i can be arranged into blocks of isomorphic groups so that, after a suitable renumbering of the factors,*

$$H = \mathrm{Diag}(T_1 \times \cdots \times T_{k_1}) \times \cdots \times \mathrm{Diag}(T_{k_{r-1}+1} \times \cdots \times T_{k_r}).$$

In other words, having identified the groups in each block, H consists precisely of the elements of the form

$$(\alpha_1, \ldots, \alpha_1), \ldots, (\alpha_r, \ldots, \alpha_r).$$

The *socle* of G is the subgroup generated by all minimal normal subgroups and is denoted by $\mathrm{Soc}(G)$. The socle is semisimple.

The *automorphism group* of G is denoted by $\mathrm{Aut}(G)$. Every element $g \in G$ induces an *inner automorphism* $x \mapsto g^{-1}xg$. The group of inner automorphisms, $\mathrm{Inn}(G)$, is normal in $\mathrm{Aut}(G)$. The factor group $\mathrm{Out}(G) = \mathrm{Aut}(G)/\mathrm{Inn}(G)$ is the *outer automorphism group*. One of the classification-dependent results required by our algorithm analysis is the so-called Schreier conjecture.

THEOREM 2.3 (Schreier conjecture). *The outer automorphism group of a finite simple group is solvable.*

2.2. Permutation groups. The group of all permutations of an n-element set A is denoted $\mathrm{Sym}(A)$, or $\mathrm{Sym}(n)$ if the specific set is unessential. Subgroups of $\mathrm{Sym}(n)$ are the *permutation groups* of *degree* n. The *even* permutations of A form the *alternating group* $\mathrm{Alt}(A)$ (or $\mathrm{Alt}(n)$). We refer to $\mathrm{Sym}(A)$ and $\mathrm{Alt}(A)$ as the *giants*. These two families of groups require special treatment in most algorithms (see sections 5 and 8).

The *support* $\mathrm{supp}(\pi)$ of $\pi \in \mathrm{Sym}(A)$ consists of those elements of A actually displaced by π, i.e., $\{a \in A : a^\pi \neq a\}$. The *degree* of π is $\deg(\pi) = |\mathrm{supp}(\pi)|$.

We say that G *acts on* A if a homomorphism $G \to \mathrm{Sym}(A)$ is given. This action is *faithful* if its kernel is the identity. The *orbit* of $a \in A$ under G is the set of images $\{a^\gamma : \gamma \in G\}$. G is *transitive* on A if there is only one orbit. We say G is t*-transitive* if the action of G induced on the set of ordered t-tuples of distinct elements of A is transitive ($t \leq n$). The maximum such t is the *degree of transitivity* of G. The degree of transitivity of the giants is $\geq n - 2$.

THEOREM 2.4. *If G is 2-transitive and $|G| \geq n^{2+\log n}$ then G is giant.*

This is an immediate consequence of the classification of doubly transitive groups, which is essentially due to Curtis, Kantor, and Seitz [CKS]. Their work is based on detailed knowledge of the finite simple group classification. For the list of doubly transitive groups, see, e.g., [Cam].

Actually, we could use a weaker version of Theorem 2.4, with no loss in the asymptotic analysis of running time. The following result has a strikingly simple, elementary proof.

THEOREM 2.5 (see [Py]). *There exists an explicitly computable constant c such that if G is 2-transitive and $|G| \geq n^{c \log^2 n}$ then G is giant.*

2.3. Orbits, orbitals, blocks, stabilizers. If G acts on A, the orbits of the induced (componentwise) G-action on $A \times A$ are called *orbitals* [Si67]. The *stabilizer* of $x \in A$ is the subgroup $G_x = \{\gamma \in G : x^\gamma = x\}$. If G is transitive on A then there is a bijection between the orbitals of G and the orbits of G_x. For an orbital Θ of G and $x \in A$, the (out)neighbors of x in the (di)graph (A, Θ) form the orbit $\Theta(x) = \{y | (x, y) \in \Theta\}$ of the stabilizer G_x. For $B \subset A$, we use G_B for the *pointwise stabilizer* $\bigcap_{x \in B} G_x$ of B, and $G_{\{B\}}$ for the setwise stabilizer $\{\gamma \in G : B^\gamma = B\}$ of B. If $B \subset A$ is stabilized by G, then we denote by G^B the restriction of G to B, so that $G^B \leq \mathrm{Sym}(B)$. Then, $G(B) = G^B_{\{B\}}$ denotes the image of the action of the setwise stabilizer of B on B.

If G is transitive on A and $G_x = 1$ for some (any) $x \in A$, then G is said to be *regular*. If G is transitive and $D \subseteq A$, D is called a *block* (for G) if for all $\gamma \in G$, either $D^\gamma = D$ or $D^\gamma \cap D = \emptyset$, and G is called *primitive* if no nontrivial blocks exist. (Trivial blocks have 0, 1, or $|A|$ elements.) If D is a block then the set of images of D is called a *block system* and an action of G is induced on the block system. The block system is *minimal* if that action is primitive.

For section 4, we need the following elementary results on the structure of primitive groups. They all follow from the O'Nan–Scott theorem [Sc] (cf. [Cam], [Lu87]).

THEOREM 2.6. *Let $G \leq \mathrm{Sym}(A)$ be primitive. If $\mathrm{Soc}(G)$ is abelian then $n = p^d$ for some prime p, A can be identified with the d-space over $GF(p)$ and (via this identification) $G \leq \mathrm{AGL}(d,p)$, the group of affine transformations of A, and $\mathrm{Soc}(G) \cong \mathbf{Z}_p^d$ is the group of translations of A.*

THEOREM 2.7. *Let $G \leq \mathrm{Sym}(A)$ be primitive. Then*

$$\mathrm{Soc}(G) = T_1 \times \cdots \times T_d$$

where the T_i are isomorphic simple groups. If $\mathrm{Soc}(G)$ is nonabelian then G contains a normal subgroup N such that

(a) $\mathrm{Soc}(G) \leq N \leq \mathrm{Aut}(T_1) \times \cdots \times \mathrm{Aut}(T_d)$;

(b) *G/N is a subgroup of S_d;*

(c) $n \geq 5^d$.

In the particular case that the isomorphic T_i are alternating groups, we say that G is of *alternating type*.

THEOREM 2.8. *Let $G \leq \mathrm{Sym}(A)$ be primitive. If G has more than one minimal normal subgroup then G has precisely two minimal normal subgroups, each of order $|A|$.*

2.4. Primitive groups of Cameron type. A remarkable class of primitive groups of alternating type is obtained as follows.

First we define a class of imprimitive groups. Let B be a set of k elements, $k \geq 5$, and $1 \leq s < k/2$. Let $C = rB = B_1 \dot\cup \cdots \dot\cup B_r$ denote the disjoint union of r copies of B. An *s-transversal* of C is a subset $X \subset C$ such that $|X \cap B_i| = s$ for $i = 1, \ldots, r$. Let A denote the set of s-transversals and let $n = |A| = \binom{k}{s}^r$. The *wreath product* $W(B,r) = \mathrm{Sym}(B) \wr S_r \leq \mathrm{Sym}(C)$ consists of all permutations of C that respect the partition $\{B_i\}$. Clearly,

$$\mathrm{Soc}(W(B,r)) = \mathrm{Alt}(B_1) \times \cdots \times \mathrm{Alt}(B_r).$$

Now let $W(B,r) \geq G \geq \mathrm{Soc}(W(B,r))$ and assume G acts transitively on the set of blocks $\{B_i\}$. Under these conditions, the action of G on A is *primitive* (and alternating type, since $\mathrm{Soc}(G) = \mathrm{Soc}(W(B,r))$). We say that the primitive groups obtained this way are of *Cameron type*.

THEOREM 2.9 (see [Li]). *If G is a primitive group of degree n and order $> n^{9 \log n}$ then G is of Cameron type.*

This is the third consequence of the simple groups classification that we require. The name "Cameron type" acknowledges the first version of Theorem 2.9 by Cameron [Cam], who formulated the lower bound as $> n^{c \log n}$, without explicit determination of the constant $c = 9$. For large n, c approaches 1. We remark that the actual value of c plays no role in the algorithms; their analysis depends only on the existence of c.

2.5. Cameron schemes. For application in section 4, we introduce a combinatorial structure associated with the action of $W(B,r)$ on A. Let A, B, C be as above. For an s-transversal $X \in A$, let $X_i = X \cap B_i$. For $X, Y \in A$, let $d_i = |X_i \cap Y_i|$ and let $f_1 \leq f_2 \leq \cdots \leq f_r$ be the sorted sequence $\{d_i\}$. We call $(f_1, \ldots, f_r)$ the *intersection pattern* of X and Y. Let us partition $A \times A$ according to intersection patterns: $A \times A = R_0 \cup \cdots \cup R_N$. We call the system $C(n,k,s,r) = (A; R_0, \ldots, R_N)$ the *Cameron scheme* with parameters (n,k,s,r). This is a particular *association scheme*

[Bos], [Del], [MS]; it includes the Hamming schemes ($s = 1$) and the Johnson schemes ($r = 1$) as particular cases. The scheme can be thought of as a coloring of the edges of the complete graph on n vertices (including self-loops); we refer to the R_i as *color classes*.

It is clear that each group of Cameron type acts on a Cameron scheme. In fact, the color classes are precisely the orbitals of the action of $W(B, r)$ on A. It may, however, happen that the color classes split under the action of a Cameron-type group $G \le W(B, r)$. In a key subroutine, NATURAL_ACTION, we recover the imprimitive action of G on $C = rB$ using the orbital structure of the primitive G-action on A, thereby reducing the Cameron-type groups to imprimitive groups with a block system of $r \le \log n / \log 5$ blocks, with giants acting on each block.

Some elementary observations about the orbital structure will be useful in this computation. Let Σ_i be the color class corresponding to the intersection pattern $(s - i, s, \dots, s)$ and Φ to $(0, 0, \dots, 0)$.

LEMMA 2.10. *Let G be a Cameron-type group acting on the points A of a Cameron scheme $C(n, k, s, r)$ and suppose $k \ge 2rs^2$. Then the following hold.*

(a) *Σ_1 is the second smallest orbital of G.*

(b) *Φ is the largest orbital of G.*

Proof. We note first that Σ_i $(0 \le i \le s)$ and Φ are orbitals of G; i.e., they do not split. For Φ this follows from the fact that $G \ge \mathrm{Alt}(k)^r$. For Σ_i we need in addition that the stabilizer of any $a \in A$ acts transitively on the set of blocks $\{B_i\}$.

Proof of (a): Fix $x \in A$ and consider an orbital Θ. We have to prove that $|\Sigma_1(x)| < |\Theta(x)|$ for any Θ other than Σ_1 and the diagonal Σ_0 (the diagonal is the *smallest* orbital). Observe, since $k \ge 2s^2$, that $\binom{k-s}{s} \ge s(k - s)$ with strict inequality when $s > 1$.

For $s \ge i > 1$,

$$|\Sigma_i(x)| = r\binom{s}{s-i}\binom{k-s}{i} > rs(k-s) = |\Sigma_1(x)|.$$

Assume now that Θ is contained in the color class with intersection pattern $(i_1, i_2, \dots)$ where $i_1 \le i_2 < s$; let $(x, y) \in \Theta$. Just counting the images of y under the stabilizer of x in $\mathrm{Alt}(k)^r$ we obtain

$$|\Theta(x)| \ge \binom{s}{i_1}\binom{k-s}{s-i_1}\binom{s}{i_2}\binom{k-s}{s-i_2}$$

$$\ge s^2(k-s)^2 > rs(k-s),$$

the final inequality using $k \ge 2r$.

Proof of (b): We have to prove that Φ is the largest color class in the Cameron scheme. (Note that G plays no role here.)

We use the fact that, for $1 \le i \le s$,

$$r\binom{k-s}{s-i}\binom{s}{i} < \binom{k-s}{s}.$$

To see this, note that $k \geq \max\{2rs^2, 2s+1\}$ implies $k - 2s + 1 > rs^2$, so that

$$\frac{\binom{k-s}{s}}{\binom{k-s}{s-i}} = \prod_{j=0}^{i-1} \frac{k-2s+i-j}{s-j} > \prod_{j=0}^{i-1} \frac{rs^2}{s-j} \geq rs^i \geq r\binom{s}{i}.$$

Now let the color class Θ have intersection pattern $(0^{r_0}, \ldots, s^{r_s})$. (The exponents denote multiplicities.) Then

$$|\Theta(x)| = \binom{r}{r_0, r_1, \ldots, r_s} \prod_{i=0}^{s} \binom{k-s}{s-i}^{r_i} \binom{s}{i}^{r_i}$$

$$< \binom{r}{r_0, r_1, \ldots, r_s} \binom{k-s}{s}^{r} \frac{1}{r^{r-r_0}} < \binom{k-s}{s}^{r} = |\Phi(x)|. \qquad \square$$

2.6. Strong generators. In our algorithms, permutation groups are input and output via sets of generators. A standard tool for permutation group computation is an SGS [Si70]. An SGS with respect to the subgroup chain $G = G_0 \geq G_1 \geq \cdots \geq G_m = 1$ is a set $T \subset G$ such that $T \cap G_i$ generates G_i for all i.

Let C_i be a set of (right) coset representatives for G_{i-1} mod G_i, $i = 1, 2, \ldots, m$. Then any $\alpha \in G$ has a unique factorization $\alpha = \rho_m \cdots \rho_2 \rho_1$ with $\rho_i \in C_i$. An SGS T is computationally effective if, for any $\alpha \in G_{i-1}$, there are fast procedures for determining the coset of G_i to which α belongs and constructing a representative for this coset from T.

We construct an SGS with respect to a subgroup chain $G = G_0 \geq G_1 \geq \cdots \geq G_m = 1$ such that each G_i is normal in G and the factor groups G_{i-1}/G_i are either subgroups of direct products of small primitive groups ("small levels") or direct products of alternating groups ("alternating levels"). To achieve the effectiveness mentioned above, we, in fact, construct an SGS with respect to a refinement $G = H_0 \geq H_1 \geq \cdots \geq H_{m'} = 1$ of the subgroup chain $G = G_0 \geq G_1 \geq \cdots \geq G_m = 1$. Namely, we construct a permutation representation for each G_{i-1} with kernel G_i. The refinement between G_{i-1} and G_i is a pointwise stabilizer chain in this representation. The advantage of a pointwise stabilizer chain is that it is easy to recognize the coset to which a given permutation belongs: given $\alpha \in H$, its coset $H_x\alpha$ is determined by x^α.

If a pointwise stabilizer chain is long, it requires too much time and storage to store all coset representatives at each level. Hence, in alternating groups, we use the following Jerrum-style [Je86] compact SGS. Suppose that $K \cong \mathrm{Alt}(m)$ and K acts naturally on a set $B = \{x_1, \ldots, x_m\}$ with K_i the pointwise stabilizer of $\{x_1, \ldots, x_i\}$. Let the set T consist of the permutations $\pi_1, \pi_2, \ldots, \pi_{m-1}$ satisfying the following properties. For all $1 \leq k \leq m-2$, π_k fixes pointwise $x_1, \ldots, x_{k-1}$ and π_{m-1} fixes pointwise $x_1, \ldots, x_{m-3}$. Moreover, $x_k^{\pi_k} = x_{k+1}$ for $k = 1, 2, \ldots, m-2$ and $x_{m-2}^{\pi_{m-1}} = x_m$. Suppose we store just the products $\mu_i = \pi_1\pi_2\cdots\pi_i$ for $i \leq m-2$ and $\mu_{m-1} = \pi_1\pi_2\cdots\pi_{m-3}\pi_{m-1}$. Then $\{\mu_{i-1}^{-1}\mu_j : i-1 \leq j \leq m-1\}$ is a complete set of coset representatives for K_i in K_{i-1}. Thus, any coset representative within the chain can be obtained with one multiplication. (We use the term *multiplication* for the evaluation of $\alpha^{-1}\beta$ as well as $\alpha\beta$; clearly the timings are the same.)

It is useful to observe that an SGS for a factor group G/N, lifted to G and appended to an SGS for N, gives an SGS for G. With an abuse of language, we call a subset $C \subset G$ a set of strong generators of G/N if C is a lifting of such a set. Suppose $\alpha \in G$ is factored according to a fixed SGS of G/N, that is, $\bar{\alpha} = \bar{\rho}_l \cdots \bar{\rho}_2 \bar{\rho}_1$, where the bar signifies the image mod N. Then $\nu = \alpha(\rho_l \cdots \rho_1)^{-1} \in N$ and we call ν the *siftee* of α into N. The following notion plays an important role in reducing the number of generators we use for N. If $C \subset G$ is an SGS for G/N and $S^* \subset G$ is a set of generators for G/N such that $C \subset \langle S^* \rangle$ (in G, not only in G/N), then we say that S^* is *compatible* with C.

2.7. Sims's algorithm. Sims's algorithm for constructing strong generators has been formulated for the case when G_i is the stabilizer of the first i points of the permutation domain. An efficient version of Sims's method has been analyzed by Knuth [Kn]. In this subsection, we describe a slight extension of the latter version.

We consider the action of $G = \langle Q \rangle \leq \mathrm{Sym}(A)$ on the set $C = \{1, 2, \ldots, m\}$. Let G_i be the pointwise stabilizer of $\{1, 2, \ldots, i\}$,

$$G = G_0 \geq G_1 \geq \cdots \geq G_m = N,$$

where N is the kernel of the G-action on C. Our objective is to find an SGS of G/N.

During the procedure, we maintain lists T_i, $i = -1, 0, \ldots, m-1$ and R_i, $i = 0, 1, \ldots, m$, where R_i is a not-necessarily-complete list of right coset representatives of G_{i-1} mod G_i; and $T_i \subseteq G_i$ such that $\langle T_i \rangle \supseteq \bigcup_{j \geq i+1} R_j$. The lists $T_{-1} := Q$ and $R_0 := \{1\}$ do not change during the procedure; all other lists may sometimes be augmented. Each time T_i is augmented, the group $\langle T_i \rangle^C$ increases.

We employ the following SIFT routine which attempts to factor $\pi \in G_k$ (k is part of the input) over the current partial coset lists. If it does not succeed then it inserts a new coset representative in the appropriate R_{j+1}, updates the T_i, $k+1 \leq i \leq j$, and sets $k := j$. In any case, at the conclusion of SIFT, $\pi \in N R_m R_{m-1} \cdots R_{k+1}$.

procedure SIFT$(\pi, C, k, \{T_i\}, \{R_i\})$
Initialize: $\sigma := \pi$, $j := k$.
    while $j \leq m-1$ **and** $\sigma \neq 1$ **do**
        if $\sigma \in G_{j+1}\alpha$ for some $\alpha \in R_{j+1}$
            then set $\sigma := \sigma\alpha^{-1}$ and $j := j+1$
            else
                begin
                    add σ to R_{j+1};
                    add σ to T_l, $l = k+1, \ldots, j$;
                    $k := j$;
                end ;
end (SIFT).

The main procedure is the following.

procedure PERMREP(Q, C)
INPUT: (Q, C) as above.
OUTPUT: $\{T_i\}, \{R_i\}$.
Initialize: $k := -1$, $T_{-1} := Q$,
        $T_i := \emptyset$ for $0 \leq i \leq m-1$, $R_i := \{1\}$, for $0 \leq i \leq m$.
    while $k \geq -1$ **do**
        begin

```
        while R_{k+1} × T_k not exhausted do
            begin
                select next (ρ, τ) in R_{k+1} × T_k;
                SIFT(ρτ, C, k, {T_i}, {R_i});
            end ;
        k := k − 1
        end
end (PERMREP).
```

Note that the intention is to put the elements of each $R_{k+1} \times T_k$ in a queue as such elements are created (by augmentation of R_{k+1} and/or T_k), ensuring that each $(\rho, \tau) \in R_{k+1} \times T_k$ is selected exactly once in the lifetime of the procedure.

The following proposition is just a reformulation of Sims's basic observations.

PROPOSITION 2.11. *When procedure PERMREP(Q, C) terminates, R_i is a complete set of coset representatives for G_{i-1} mod G_i and $\langle T_i \rangle^C = G_i^C$ for $0 \le i \le m$.*

Proof. Recall that N denotes the kernel of the G-action on C. As a result of having sifted $R_i T_{i-1}$, we know $R_i T_{i-1} \subseteq N R_m R_{m-1} \cdots R_i$ for $0 \le i \le m$. We have also maintained the properties $T_i \subseteq G_i$, $\bigcup_{j \ge i+1} R_j \subseteq \langle T_i \rangle$, and the elements of R_i represent distinct cosets mod G_i.

Since $Q = R_0 T_{-1} \subseteq N R_m \cdots R_1 \subseteq N\langle T_0 \rangle$, $G = N\langle T_0 \rangle$. Suppose, for any i that $G_i = N\langle T_i \rangle$; then $R_{i+1} T_i \subseteq N R_m R_{m-1} \cdots R_{i+1} \subseteq N\langle T_{i+1} \rangle R_{i+1}$, whence $G_i \subseteq N R_{i+1} \langle T_i \rangle \subseteq N\langle T_{i+1} \rangle R_{i+1}$. It follows that $G_i = N\langle T_i \rangle = N G_{i+1} R_{i+1}$ and $G_{i+1} = N\langle T_{i+1} \rangle$. □

We use the following easy facts about PERMREP(Q, C). We set $n = |A|$ and assume $n \ge m = |C|$. Therefore, the cost of each group operation is $O(n)$. Let $t = \max\{|G_{i-1}^C : G_i^C| : 1 \le i \le m\}$; note that $t \le m$. Also, $\log |G^C|$ is an upper bound on the length of subgroup chains in G^C so there are $\le \log |G^C|$ indices i such that $G_i \ne G_{i+1}$. In particular, the cost of each sift is $O(n \log |G^C|)$ and each T_i is increased $\le \log |G^C|$ times. From this, we obtain the following estimates.

THEOREM 2.12. (a) *Let $|Q| = q$. The running time of PERMREP(Q, C) is $O(n \log |G^C|(q + t \log^2 |G^C|))$; in particular, if $|G^C| \le \exp(\log^c n) = \exp(O^\sim(1))$ then the running time is $O^\sim(n(q + t))$.*

(b) *At any moment during the execution of the algorithm, $|T_{k-1}| \le 1 + \sum_{j=k}^m \log |R_j|$.*

Remark 2.13. The $O(n^5)$ bottleneck that is inherent to all versions of Sims's method [Si70], [FHL], [Je86], [Kn] can be appreciated in the context of PERMREP (take $C = A$ and $N = 1$). These methods rely on the construction of generators for the groups in the pointwise stabilizer chain, using Schreier's construction of subgroup generators. (In PERMREP, Schreier generators enter in the sifting of $R_{k+1} T_k$, since the sift of $\rho\tau$ begins with finding $\rho_1 \in R_{k+1}$ such that $\rho\tau\rho_1^{-1} \in G_{k+1}$.) In general, $|R_{k+1}|$, $|T_k|$ and the number of groups in the chain can each be $\Omega(n)$ so there may be $\Omega(n^3)$ elements to sift and a sift may cost $\Omega(n^2)$. In fact, Knuth discusses a class of groups in which the *average* behavior of such methods is $\Theta(n^5)$.

As in [FHL], a slight modification of PERMREP provides normal closures. The addition to the previous procedure is that we have to add conjugates of generators to the generating set until we get a group closed for conjugation. We again consider group actions.

The situation is the following: $G = \langle S \rangle \le \mathrm{Sym}(A)$ and $\langle Q \rangle \le \mathrm{Sym}(A)$ act on C. The output consists of sets of coset representatives $\{R_i\}$ and sets of generators of the stabilizer chain over C for $H := \langle Q^G \rangle$. For sets of permutations T and S, T^S denotes

the set of conjugates $\{\tau^\sigma : \tau \in T, \sigma \in S\}$.

procedure NORMCL(Q, C, S)
INPUT: (Q, C, S) as above.
OUTPUT: $\{T_i\}, \{R_i\}$.
    PERMREP(Q, C);
    $T^* := \emptyset$;
    repeat
        $k := -1$, $T_{-1} := (T_0 \setminus T^*)^S$, $T^* := T_0$;
        while $k \geq -1$ **do**
            begin
                while $R_{k+1} \times T_k$ not exhausted **do**
                    begin
                        select next $(\rho, \tau) \in R_{k+1} \times T_k$;
                        SIFT$(\rho\tau, C, k, \{T_i\}, \{R_i\})$;
                    end ;
                $k := k - 1$
            end
    until $T_0 = T^*$
end (NORMCL).

The proof of correctness and the timing of this algorithm is similar to that of PERMREP (with H playing the role of G in the estimates). Let $t = \max\{|H_{i-1}^C : H_i^C| : 1 \leq i \leq m\}$.

PROPOSITION 2.14. *When procedure NORMCL(Q, C, S) terminates, the R_i form complete sets of coset representatives for H, and $\langle T_i \rangle^C = H_i^C$.*

THEOREM 2.15. *Let $s = |S|$, $q = |Q|$. The running time of NORMCL(Q, C, S) is $O(n \log |H^C|(q + s \log |H^C| + t \log^2 |H^C|))$; in particular, if $|H^C| \leq \exp(\log^c n) = \exp(O^\sim(1))$ then the running time is $O^\sim(n(q + s + t))$.*

2.8. Structure forest, structure domain. It is natural in dealing with permutation groups, whether theoretically or in computational settings, to use the orbit structure in a problem decomposition. Further combinatorial divide-and-conquer is available in the imprimitivity structure of transitive groups. For computational purposes, it is convenient to provide an extension of the permutation domain that both reflects and guides the flow of control in such procedures. Specifically, a *structure forest* (SF) for a permutation group $G \leq \mathrm{Sym}(A)$ is a forest of rooted trees on which G acts as automorphisms fixing the roots, such that the leaves form the permutation domain A, and, denoting by $G(v)$ the permutation group induced on the children of node v by G_v (the stabilizer of v), each $G(v)$ is primitive. Thus, in particular, there is exactly one tree per orbit in A, and it is not possible to insert intermediate levels in that tree, with nontrivial branching, and remain consistent with the G action on the tree.

To reflect the flow of control in our procedure (e.g., treating orbits sequentially) we suppose that the trees of the SF are stacked vertically and enumerate the resulting "levels." Hence, the root of the first tree comprises level 0, its leaves comprise level h, where h is the height of this tree, the root of the second tree comprises level $h + 1$, etc.

The divide-and-conquer offered by the SF alone does not suffice for our methods. To achieve a finer decomposition, we need to delve into the primitive groups themselves; specifically, for "large" groups, we use the forced relations between the nature

of the socle and that of the permutation domain.

The first and principal stage creates an *extended structure forest* (ESF). For this, the SF is augmented at nodes v where $G(v)$ is a "large" group, i.e., of order $> \exp(\log^c n)$. At such places, we are assured that $G(v)$ is, in fact, a Cameron-type group with $\mathrm{Soc}(G(v)) \cong \mathrm{Alt}(k)^r$. Such $G(v)$ has a natural imprimitive representation on a set B of size kr, and we can build a structure forest (in fact, a tree) $T(v)$ on B for $G(v)$. Our algorithm constructs the trees $T(v)$ so that the leaves of $T(v)$ correspond to certain subsets of the children of $G(v)$. In particular, we need only do the work of constructing $T(v)$ at one node v at each level of the SF, using the action of G to copy the trees to other nodes at the same level. As a result, the permutation action of each element of G naturally extends to the ESF. We consider the trees $T(v)$ appended to the SF to be placed *entirely between levels of the initial forest*. Having so situated the $T(v)$, we delete the edges between v and its children in the original forest. Thus, edges, where they exist, in the ESF only traverse consecutive levels. It is important to note, however, that $G(v)$ acts faithfully on the leaves of $T(v)$, so that the subgroup of G that fixes all the leaves at this level also fixes all the nodes at the level of the children of v in the SF.

We now continue to use $G(v)$ to denote the (primitive) permutation group induced by G_v on the set of children of the node v of the ESF. (In context, it is clear which $G(v)$ is intended when we specify the ambient graph.) Thus, in the ESF, $G(v)$ is either a "small" group (of order $< \exp(\log^c n)$) or a giant. Furthermore, the groups at a given level are isomorphic, in fact, conjugate under the action of G; accordingly, we can talk about *small group levels* and *giant levels* in the ESF.

A second refinement is used to restrict the giant levels to be alternating. Consider a node v of the ESF where $G(v)$ is a full symmetric group $\mathrm{Sym}(C(v))$ on the children $C(v)$ of v. At each such node, we append a small tree consisting of the root v and two leaves, say v_L and v_R (for "left" and "right"), which are inserted at a new level between v and $C(v)$. Again, we sever the links from v to $C(v)$, but we now connect both v_L and v_R to all points in $C(v)$. We need to extend the action of G to the new intermediate level. This may be done by fixing any orderings of the $C(v)$, relative to which the actions of $\gamma \in G$ can be viewed as inducing even or odd permutations; if γ induces an "even" mapping of $C(v)$ to $C(w)$ then $v_L^\gamma = w_L$ and $v_R^\gamma = w_R$, else $v_L^\gamma = w_R$ and $v_R^\gamma = w_L$. We call the resulting structure D a *structure domain* (SD) for G.

For a node $v \in D$, we continue to denote the children of v, that is, the neighbors at next level by $C(v)$ and the action of G_v on $C(v)$ by $G(v)$.

We summarize some important properties of this structure. A structure domain for $G \leq \mathrm{Sym}(A)$ is a graph $D = (V, E)$ such that the following hold.

(i) $A \subseteq V$ and $|V| = O(n)$, where $n = |A|$.

(ii) The action of G extends to $\mathrm{Aut}(D)$.

(iii) The orbits of G in V, called "levels," are ordered, $L_0, L_1, \ldots, L_m$, and $E \subseteq \bigcup_{i=0}^{m-1}(L_i \times L_{i+1})$.

(iv) If $E \cap (L_i \times L_{i+1}) = \emptyset$, then $G_{L_i} \leq G_{L_{i+1}}$.

(v) If $E \cap (L_i \times L_{i+1}) \neq \emptyset$, then, letting $G(v)$ denote the action of G_v on the neighbors $C(v)$ in L_{i+1} of $v \in L_i$, it follows that $G(v)$ is either a "small" group or $\mathrm{Alt}(C(v))$.

Let $G_0 = G$ and, for $i \geq 1$, let G_i be the kernel of the action of G_{i-1} on L_i. Then the normal series

$$G = G_0 \geq G_1 \geq \cdots \geq G_m = 1$$

is the chain forecast in the introduction and in section 2.6. Instances of (iv) suggest that the chain is not strictly decreasing (and one can have equality of successive groups even when the induced bipartite graph is nontrivial), but it is convenient to allow this occasional duplication. Note, however, that (v) implies, when $G_{i+1} < G_i$, that G_i/G_{i+1} is either a product of isomorphic alternating groups or a subgroup of the product of isomorphic small primitive groups. The fact that, in the former case, G_{i-1}/G_i is actually isomorphic to a product of alternating groups (not only a subgroup) follows from Proposition 2.2.

Informally, we say the SD consists of *small group levels* and *alternating levels*.

3. Organization of the algorithm. In this section, we outline our main algorithm. Suppose that $G = \langle S \rangle \leq \mathrm{Sym}(A)$ is given, $|A| = n$. We construct a chain of normal subgroups $G = G_0 \geq G_1 \geq \cdots \geq G_m = 1$ and, for each $1 \leq i \leq m$, a permutation representation of G_{i-1} on a set L_i such that

(i) G_i is the kernel of the action of G_{i-1} on L_i;

(ii) G_{i-1}/G_i is either a subgroup of a direct product of small primitive groups ("small" in this context means of order $< \exp(9 \log^2 n \log\log n)$) or $G_{i-1}/G_i \cong \mathrm{Alt}(k)^r$ for some k, r.

The normal subgroup G_i is defined to be the pointwise stabilizer of the first i levels in a structure domain (see section 2.8). However, we have to *construct* generators for the G_i. We do this successively for $i = 0, 1, \ldots, m-1$. We construct an SGS T_i for G/G_i and *normal generators* N_i for G_i, i.e., group elements whose normal closure (in G) is G_i.

Suppose we have constructed T_{i-1} and N_{i-1}. We start to take the normal closure of N_{i-1} in G until the known part of the normal closure generates G_{i-1}/G_i. We confirm this by examining the action of G_{i-1} on L_i. Then we obtain T_i by appending an SGS for G_{i-1}/G_i to T_{i-1}. For this, if G_{i-1}/G_i corresponds to a small group level then we add complete sets of coset representatives from the point stabilizer chain on L_i to T_{i-1} (we do ensure that the total number of saved coset representatives in the entire subgroup chain is only $O(n \log^2 n)$); if G_{i-1}/G_i is the product of alternating groups then we add Jerrum-style compact SGS (cf. section 2.6) for each of these alternating groups to T_{i-1}. We also obtain a presentation for G_{i-1}/G_i, which, along with presentations for earlier quotients, facilitates a construction of normal generators N_i for G_i. Thus we proceed to the next value of i.

We emphasize that generators for G_i (not only normal generators) are available only when the entire algorithm is finished.

During the algorithm, we work with various permutation representations of subgroups of G. If a procedure performs group operations, we may either need the result in the current representation only (*local* operation) or in the original representation as well (*global* operation). An example of a purely local operation is the determination of whether the stabilizer of a node v in the SF acts as a Cameron-type group on its children. To this end, it is enough to perform group operations in $G(v)$ and the action of G_v on other nodes of the SF is irrelevant. All operations not explicitly labelled "local" are understood to be global. Since the sum of sizes of all the induced permutation representations remains $O(n)$, the cost of elementary group operations remains $O(n)$.

Main Algorithm.
INPUT: a set S of generators for $G \leq \mathrm{Sym}(A)$, $|S| = s$.
Step 1. Construct a structure forest and choose a representative v in each orbit of the SF. For all such v, construct Schreier generators for G_v.

Step 2. For these representatives, use NATURAL_ACTION to decide whether $G(v)$ is a "large group" and, if so, construct new action and corresponding structure tree $T(v)$.
Step 3. Append a copy of $T(v)$ to all nodes in the orbit v^G, deleting the connections of v to its children in the SF, thus obtaining an ESF. Extend the G-action of generators to the ESF. Inserting new levels at giant symmetric nodes, obtain the SD. Henceforth, compute the effect of any global operation on the entire SD. Compute the node stabilizers G_w as in Step 1 for representatives of G-orbits of the SD.
Step 4. For each node v representing an alternating level in the SD, construct an SGS for $G(v)$.

Step 5. **for** $i := 1$ to m **do**
construct SGS for G_{i-1}/G_i
store a *compatible* generating set S_{i-1} of size $O^\sim(|L_i|)$ for G_{i-1}/G_i
(* cf. section 2.6 *)
construct normal generators for G_i
end (MAIN ALGORITHM).

LEMMA 3.1. (a) *A structure forest can be computed in* $O(sn^2)$ *time.* (b) *Generators of* G_v *for representatives of the* G*-orbits of the SF can be constructed in* $O(sn^2)$ *time.*

Proof. (a) According to Atkinson [At], a structure forest can be computed as efficiently as orbits and minimal blocks of imprimitivity, i.e., in $O(sn^2)$ time.

(b) The action of the group generators on the orbit v^G of a node v in the SF naturally defines a graph on v^G. Choosing a spanning tree in this graph and computing the product of generators along the paths from v in this tree, group elements which carry v to the other nodes of its orbit can be computed in $O(|v^G|n + |v^G|s)$ time. We obtain generators for G_v (and, at the same time, for $G(v)$) via Lemma 2.1; thus G_v is generated by $O(|v^G|s)$ elements and the cost of computing each is $O(n)$. The result follows since the sum of the $|v^G|$ over orbit representatives v is the number of nodes in the SF. □

Steps 2 and 3 will be analyzed in section 4. We present a novel method for constructing an SGS for the giants in section 5. Section 6 relates group presentations (in terms of generators and relations) to the construction of normal generators. By the results of section 4, the factor groups G_{i-1}/G_i in Step 5 are either subgroups of products of "small" groups or products of alternating groups. We handle the first case in section 7, utilizing NORMCL (cf. section 2.7). For the second case, we give an efficient implementation of Luks's "noncommutative linear algebra" [Lu86] in section 8. Finally, in section 9, we present a version of the algorithm with decreased memory requirement and wrap up the proof of Theorem 1.1.

4. Reducing large to giant. The purpose of this section is to classify primitive groups as "large" and "small." Large groups turn out to be groups of Cameron type, and we construct their "natural" (often imprimitive) action with giants acting on each block and a small group permuting the blocks. Thereby most algorithmic problems are reduced to consideration of giants and small groups.

Our objective is achieved by the subroutine NATURAL_ACTION. This procedure is a slight refinement of the one under the same name in [BLS87]. The procedure involves a global variable n, the degree of the permutation group which is the input of the full algorithm. However, we execute group multiplications only on the set where the group under the current investigation acts primitively (local operations, cf.

section 3).

First, we describe a simple procedure to test whether or not a permutation group is a giant.

procedure TEST_GIANT(G)
INPUT: a 2-transitive group $G = \langle Q \rangle \leq \mathrm{Sym}(C)$, $|C| = m$.
Begin executing PERMREP(Q, C)
if $|\{i : |R_i| \neq 1\}| \geq 2\log m + \log^2 m$ ($*$ we use the notation of section 2.7. $*$)
then stop PERMREP(Q, C); **output** "giant" and **halt**
else output "small group" and **halt**
end (TEST_GIANT).

When reading the following pseudocode, it is useful to review the notation of sections 2.4 and 2.5 and keep in mind that NATURAL_ACTION was designed to handle Cameron-type groups, when $m = \binom{k}{s}^r$ and the underlying set A corresponds to s-transversals in rB for some set B of size k. In that scenario, Γ and Δ correspond to the sets of pairs of s-transversals with intersection pattern $(s-1, s, \ldots, s)$ and $(0, 0, \ldots, 0)$, respectively. The primary aim of the procedure is to construct a subset of A corresponding to the s-transversals containing a fixed point in rB. Such a set is constructed as $C(x, y)$ below, where x, y are s-transversals with intersection pattern $(s-1, s, \ldots, s)$ and $C(x, y)$ consists of all s-transversals containing the unique point in x that is not covered by y. The subset $C(x, y)$ has kr distinct images under G, corresponding to the points of rB. We compute this set D of images in two phases, first constructing in $D(x)$ only the rs images corresponding to the points of x. This and other checks on the sizes of newly constructed objects allow early termination in the case when G is not a large group.

procedure NATURAL_ACTION(Q)
INPUT: a primitive group $G = \langle Q \rangle \leq \mathrm{Sym}(A)$, where $m := |A| \leq n$.
Step 1. **if** $m \leq 3\log^2 n$
then output "small group" and **halt**
Step 2. **if** G is 2-transitive
then TEST_GIANT(G); ($*$ procedure will halt there $*$)
Step 3. Compute the orbitals (G-orbits on $A \times A$);
$\Gamma :=$ the second smallest orbital;
($*$ The smallest orbital is the diagonal. $*$)
$\Delta : D$ the largest orbital.
Fix $x \in A$;
if $|\Gamma(x)| > 2\sqrt{m}\log m$
then output "small group" and **halt**
For each $w \in A$ construct $\alpha(w) \in G$ such that $x^{\alpha(w)} = w$.
Construct Schreier generators for G_x.
Fix $y \in \Gamma(x)$. For each $y' \in \Gamma(x)$ compute some $\beta(y') \in G_x$ such that $y^{\beta(y')} = y'$.
Compute the sets

$$B(x, y) = \Delta(y) - \Delta(x);$$
$$C(x, y) = A - \bigcup_{z \in B(x,y)} \Delta(z).$$

Let $D(x) = \{C(x, y)^{\beta(y')} : y' \in \Gamma(x)\}$.
if $|D(x)| > \log m$
then output "small group" and **halt**

Let $D = \bigcup_{w \in A} D(x)^{\alpha(w)}$.
if $|D| > 2\sqrt{m} \log m$
then output "small group" and **halt**
Step 4. Consider G-action on D. ($*$ This action exists and it is transitive. $*$)
Select a system $\{E_1, \ldots, E_t\}$ of minimal-size (but nonsingleton) imprimitivity blocks
($* \bigcup_i E_i = D *$).
if $q := |E_i| > 4 \log\ n$ **and** $G(E_1)$:=the stabilizer of E_1 restricted to E_1 is 2-transitive
and TEST_GIANT($G(E_1)$) returns message "giant"
then output ("large group, faithfully acting on D";
the G-action on D;
a structure tree for the G-action on D)
else output "small group"
halt
end (NATURAL_ACTION).

We say that G *fails the large groups test* if output is "small group." Otherwise G is said to *pass the large groups test*.

4.1. Correctness of the subroutine NATURAL_ACTION.

LEMMA 4.1. *If TEST_GIANT(G) outputs "giant" then G is a giant. If the output is "small group" then $|G| < m^{2+\log m}$.*

Proof. It is proved by Theorem 2.4. □

The following result justifies the term "small groups" and provides additional information about large groups.

THEOREM 4.2. (1) *If NATURAL_ACTION outputs "giant" then G is a giant.*

(2) *If NATURAL_ACTION outputs "large group" then G acts faithfully on D and the stabilizer of each block E_i restricted to E_i contains* $\mathrm{Alt}(E_i)$.

(3) *If NATURAL_ACTION outputs "small group" then*

$$|G| < \exp(9 \log^2 n\ \log\log n).$$

Statement (1) is obviously correct. For (2) we need a lemma.

LEMMA 4.3. *For $p \neq r$ primes, the order of a Sylow r-subgroup of the linear group* $\mathrm{GL}(d, p)$ *is less than p^{2d}.*

Proof. This result is implicit in [Lu82, Lemma 3.6]. □

COROLLARY 4.4. *For $q \geq 4d \log p$, the order of* $\mathrm{Alt}(q)$ *does not divide the order of the affine linear group* $\mathrm{AGL}(d, p)$.

Proof. Let $r = 3$ if $p = 2$ and let $r = 2$ otherwise. The result follows from Lemma 4.3 (except for the two easy cases $p = 2, d \leq 3$). □

Proof of Theorem 4.2*, part* (2). We show that the alternating groups constructed by the procedure are in the socle of G and G has a unique minimal normal subgroup. These facts imply that the G-action on D has a trivial kernel. We say that the group H is *involved* in the group K if $H \cong L/M$ for some $M \triangleleft K, M \leq L \leq K$. If a simple group H is involved in K then clearly H is involved in a composition factor of K.

We may assume G is not a giant. Let K be the kernel of the G-action on D. The stabilizer of E_1 restricted to E_1 passed TEST_GIANT, whence it contains $\mathrm{Alt}(q)$, $q := |E_1|$. As the G-action on the set of blocks is transitive, the same holds for each E_i. Also, it follows that $\mathrm{Alt}(q)$ is involved in G/K.

If $\mathrm{Soc}(G)$ is abelian then, by Theorem 2.6, $m = p^d$ for some prime p and $G \leq \mathrm{AGL}(d, p)$. But, $d \log p = \log\ m \leq \log\ n \leq q/4$ and therefore, by Corollary 4.4, the order of $\mathrm{Alt}(q)$ does not divide $|G|$. Thus this case cannot occur. Hence $\mathrm{Soc}(G)$

is nonabelian and the results stated in Theorem 2.7 apply. We use the notation of Theorem 2.7 and refer to $N \triangleleft G$ established there.

First we show that $\mathrm{Alt}(q)$ is not involved in $G/\mathrm{Soc}(G)$. Indeed, otherwise $\mathrm{Alt}(q)$ must be involved either in G/N or in $N/\mathrm{Soc}(G)$. The first case is impossible because $G/N \leq S_d$ (Theorem 2.7(b)) and $d \leq \log\ m/\log\ 5 < q/8$ (Theorem 2.7(c)). In the second case, $\mathrm{Alt}(q)$ is involved in $N/\mathrm{Soc}(G) \leq \mathrm{Out}(T)^d$, a solvable group by the Schreier conjecture (Theorem 2.3), again a contradiction.

It follows now that $\mathrm{Alt}(q)$ is involved in $\mathrm{Soc}(G)$ and $K \not\geq \mathrm{Soc}(G)$. Now $\mathrm{Soc}(G)$ must be the unique minimal normal subgroup for otherwise, by Theorem 2.8, we have a contradiction:

$$n^2 \geq m^2 = |\mathrm{Soc}(G)| \geq |\mathrm{Alt}(q)| = q!/2 > 2^q \geq n^4.$$

It follows that K contains no minimal normal subgroup, whence $K = 1$. □

Proof of Theorem 4.2*, part* (3). Assume the order of $|G|$ exceeds the stated bound. We must show that at no point will "small group" be falsely announced. This would not happen in Step 1, for $m \leq 3\log^2 n$ implies $|G| \leq m! < (3\log^2 n)^{3\log^2 n}$. If G is 2-transitive then the Step 2 call to TEST_GIANT will correctly halt with that revelation (by Theorem 2.4).

By Theorem 2.9, G is of Cameron type and A can be identified with the set of points of a Cameron scheme $C(m, k, s, r)$, and we may assume that $rs > 1$; that is, G is not a giant. Of course, the parameters and the identification are not known a priori. We prove that Step 3 of NATURAL_ACTION correctly recovers this structure with D corresponding to rB, E_i to B_i (so $q = k$), and the parameter k satisfies $k > 4\log n$. (We use the letters r, k, B_i, $C = rB = B_1 \cup \cdots \cup B_r$ to mean what they do in section 2.5. We call the action of G on C "natural." Recall that each $a \in A$ corresponds to an s-transversal $T(a) \subset rB$.)

We take note of some inequalities satisfied by the parameters of this $C(m, k, s, r)$. Since $m = \binom{k}{s}^r \geq (k/s)^{rs} \geq 2^{rs}$,

$$rs \leq \log m. \tag{4.1}$$

Since $rs > 1$, we have $m \geq \binom{k}{2}$ which implies

$$k < 2\sqrt{m}. \tag{4.2}$$

Also,

$$k > 4\log n; \tag{4.3}$$

otherwise, $|G| \leq (k!)^r r! \leq k^{kr} r! \leq m^k r! \leq n^{4\log n}(\log n)! < \exp(9\log^2 n\ \log\log\ n)$. Finally,

$$k \geq 2rs^2; \tag{4.4}$$

otherwise, using (4.1), we have $|G| \leq (k!)^r r! < (2rs^2)^{2r^2s^2} r! < (2\log^2 n)^{2\log^2 n}(\log\ n)!$ $< \exp(9\log^2 n\ \log\log\ n)$.

We claim now that the G-action on D is similar to the natural G-action on C. For $b \in rB$, let $U(b) = \{u \in A | b \in T(u)\}$. We need to show that that $D = \{U(b) | b \in rB\}$. By (4.4) and Lemma 2.10, $\Gamma = \Sigma_1$ and $\Delta = \Phi$. Thus, for any $y \in \Gamma(x)$, the set $T(x) - T(y)$ is a singleton $\{b(x, y)\}$. Now, a simple inspection of the Cameron scheme,

using that $k > 3s$ (by (4.4) since $rs > 1$), shows that $C(x, y) = U(b(x, y))$. The result follows since G acts transitively on C.

The identification of the E_i with the B_i follows because the latter are the unique minimal-size blocks in the natural action of G. Finally, (4.1), (4.2), and (4.3) ensure that the cardinality tests on that $\Gamma(x), D(x), D, E_i$ in Steps 3 and 4 do not cause terminal output "small group." □

4.2. Time complexity of the subroutine NATURAL_ACTION.

LEMMA 4.5. *Suppose $G = \langle Q \rangle$ acts on an m-set, and $|Q| = q$. Then TEST_GIANT(G) runs in $O^{\sim}(m^2 + qm)$ time.*

Proof. We work with $O^{\sim}(1)$ coset representative sets R_i each of size $2 \leq |R_i| \leq m$, and, by Theorem 2.12(b), $O^{\sim}(1)$ sets of generators T_i of size $O^{\sim}(1)$. Hence the sifting of products of the form $\rho\tau$, $\rho \in R_i, \tau \in T_{i-1}$ for some i costs $O^{\sim}(m^2)$; in addition, we may have to sift the elements of Q. □

THEOREM 4.6. *Suppose $G = \langle Q \rangle$ acts on an m-set, and $|Q| = q$. Then NATURAL_ACTION(Q) runs in $O^{\sim}(qm^2)$ time.*

Proof. Testing 2-transitivity takes $O(m^2q)$ time. Hence, by Lemma 4.5, Step 2 can be executed in $O^{\sim}(m^2q)$. Finding the orbitals requires $O(m^2q)$ steps. The (local) computation of $\{\alpha(w) : w \in A\}$ requires $O^{\sim}(m^2 + qm)$ time. The number of Schreier generators is qm; they are found in $O(m^2q)$ time. $O^{\sim}(qm^{3/2})$ steps suffice to find the $\beta(y')$. $C(x, y)$ can be determined in $O(m^2)$ time. We compute $D(x)$ in $O^{\sim}(m^2)$. Finally, D is also obtained in $O^{\sim}(m^2)$. Thus Step 3 requires $O^{\sim}(m^2q)$ total time.

The action of any $\phi \in Q$ on D can be found in $O^{\sim}(m^{3/2})$. The structure tree is computed in $O^{\sim}(mq)$, and, since $r = O^{\sim}(1)$, generators for the stabilizers of orbit-representatives on the new tree can be computed in $O^{\sim}(qm)$ time. Finally, we call TEST_GIANT on a set of size $O^{\sim}(\sqrt{m})$, with $O^{\sim}(q)$ generators, requiring $O^{\sim}(m + q\sqrt{m})$ time. Thus the time complexity of Step 4 is $O^{\sim}(mq + m^{3/2})$. □

COROLLARY 4.7. *Step 2 of the main algorithm runs in $O^{\sim}(sn^2)$ time.*

Proof. We apply NATURAL_ACTION to the action $G(v)$ of the point stabilizer G_v on the children of v for certain nodes v of the structure forest (one node from each level of each tree). Denoting by q_v the number of (Schreier) generators for G_v and by m_v the number of children of v, $\sum_v (q_v m_v) = O(sn)$. □

4.3. Extending the structure forest.

PROPOSITION 4.8. *Step 3 of the main algorithm runs in $O^{\sim}(sn^2)$ time.*

Proof. Suppose that a node v is the representative of an orbit in the original SF, v has m children, and NATURAL_ACTION appended a tree $T(v)$ to v. The vertices of $T(v)$ are subsets of the children of v; hence the group elements carrying v to the other nodes of its orbit v^G, computed in Step 1, naturally define a copy of $T(v)$ appended to the other nodes in v^G. These copies of $T(v)$ can be obtained in $O^{\sim}(m^{3/2}|v^G|)$ and the action of any $\sigma \in G$ can be extended to the appended trees within the same time bound. Hence the action of σ on the entire ESF can be computed in $O^{\sim}(n^{3/2})$. The extension to the SD is straightforward and in time $O(n)$. Finally, as in Lemma 3.1(b), generators of G_v for representatives of G-orbits of the SD can be constructed in $O(sn^2)$ time. □

5. Constructing strong generators for a giant. The purpose of this section is to construct a Jerrum-style compact SGS for the giants. Recall that the "giants" are the symmetric and alternating groups in their natural action. The Jerrum-style compact SGS for $G = \mathrm{Sym}(C)$ acting on the set $C = \{x_1, x_2, \ldots, x_m\}$ consists of $m-1$ permutations $\pi_1, \pi_2, \ldots, \pi_{m-1}$ such that for all $1 \leq k \leq m-1$, π_k fixes $x_1, \ldots, x_{k-1}$

and $x_k^{\pi_k} = x_{k+1}$. For $G = \mathrm{Alt}(C)$, the SGS contains $\pi_1, \pi_2, \ldots, \pi_{m-2}$ as above while π_{m-1} fixes $x_1, \ldots, x_{m-3}$ and $x_{m-2}^{\pi_{m-1}} = x_m$.

We require that the strong generators are constructed from the given generators of the giant by the following *legal operations: multiplication, inversion, and taking powers of permutations*. The reason for this constraint is that the procedure is applied to the case when $G(v)$, the action of the stabilizer of node v on the children in the structure domain, is an alternating group. In this case, although we know a priori that some $\sigma \in G(v)$ acts on the children as, e.g., a given 3-cycle, no such permutation can be guaranteed to belong to the input group unless it has been constructed, by way of legal operations, from the generators of $G(v)$. In this application, all group operations are global; i.e., we perform them on all points in the SD.

5.1. Construction of a 3-cycle. The essence of the procedure is the construction of a 3-cycle. Once a 3-cycle ρ is constructed, an SGS can be obtained easily by taking appropriate conjugates of ρ.

We note that a byproduct of the procedure yields a simple, elementary proof of the old result, known to Jordan (1895) [Jo] (and vastly surpassed by Theorem 2.4) that the only $c \log^2 n / \log \log n$ -fold transitive permutation groups are the giants [BS87]. It also yields an $\exp(\sqrt{n \ln n}(1 + o(1)))$ upper bound on the diameter of any Cayley graph of the giants [BS88].

Our goal is achieved by the procedure THREE_CYCLE. As a preprocessing phase, we determine and store the first $\log n$ primes. (The global variable n is the degree of the permutation group which is the input of the entire algorithm; we assume that n is sufficiently large.) We denote the ith prime by p_i and the product of the first i primes by $p(i)$.

Also, we need the following definitions. For $\pi \in \mathrm{Sym}(C)$, let us call a subset B of $\mathrm{supp}(\pi)$ *independent* with respect to π if $B \cap B^\pi = \emptyset$. The *commutator* of $\pi, \tau \in \mathrm{Sym}(C)$ is $[\pi, \tau] = \pi\tau\pi^{-1}\tau^{-1}$.

The procedure THREE_CYCLE uses the subroutine ORBITALS. Given generators for some $G \leq \mathrm{Sym}(C)$, $|C| = m$, ORBITALS returns $O(\log m)$ generators for a subgroup $H \leq G$ with the same orbitals as G. In particular, if G is a giant then ORBITALS returns $O(\log m)$ generators for a 2-transitive subgroup. The idea is the following. Suppose that generators for some $H \leq G$ are already defined but the orbital structures of the two groups are different. We fix an ordering of the generators $P = \{\tau_1, \ldots, \tau_k\}$ of G, and, for each H-orbital Δ which is not an orbital of G, we find the last element of P which moves Δ. Then we add a product of the form $\tau_1^{\varepsilon_1}\tau_2^{\varepsilon_2}\cdots\tau_k^{\varepsilon_k}$ to the generators of H where each $\varepsilon_i \in \{0, 1\}$ and ε_j is chosen such that $\tau_1^{\varepsilon_1}\tau_2^{\varepsilon_2}\cdots\tau_j^{\varepsilon_j}$ moves at least half of the H-orbitals Δ for which τ_j was the last generator moving Δ. This is a deterministic version of the *random subproduct method*, which we describe in section 5.4.

procedure ORBITALS(P, C, R)
INPUT:$G = \langle P \rangle \leq \mathrm{Sym}(C)$, $|C| = m$, $P = \{\tau_1, \ldots, \tau_k\}$.
OUTPUT: Generators R for some $H \leq G$ with same orbitals as G, $|R| = O(\log m)$.
Initialize: compute orbitals $O_1, \ldots, O_p$ of G; $R = \emptyset$
repeat
    compute orbitals $\{\Delta_i : i \in I\}$ of $\langle R \rangle$
    for all $\Delta_i \notin \{O_1, \ldots, O_p\}$, compute
        $\mathrm{last}(\Delta_i) := \max\{j : \Delta_i^{\tau_j} \neq \Delta_i\}$
    $\sigma := 1$ ($*$ start constructing new element of R $*$)

for $j := 1$ to k **do**
if $|\{\Delta_i : \text{last}(\Delta_i) = j, \Delta_i^\sigma = \Delta_i\}| > |\{\Delta_i : \text{last}(\Delta_i) = j, \Delta_i^\sigma \neq \Delta_i\}|$
then $\sigma := \sigma\tau_j$
$R := R \cup \{\sigma\}$
until orbitals of G = orbitals of $\langle R\rangle$
end (ORBITALS).

Steps 1–3 in the next procedure can be viewed as a preprocessing phase in which we construct the first $\log^2 n$ coset representative sets in the stabilizer chain of a giant G. With these coset representative sets in hand, it is easy matter to construct a permutation $\tau \in G$ that has a prescribed effect on an arbitrary subset of size $\log^2 n$ (cf. Lemma 5.2). Such constructed elements are useful in a computation that replaces a given element λ by one with significantly smaller support. For an appropriately designed τ, $\lambda_1 = [\lambda, \tau]$ contains cycles of prime length for a lot of different primes. An underlying idea then is that one of these primes does not divide most of the cycle lengths in λ_1. Taking an appropriate power of λ_1, we can kill all cycles whose length was not divisible by that prime and we get a permutation with smaller support. Iterating the process, we obtain a 3-cycle.

procedure THREE_CYCLE(Q)
INPUT: $G = \langle Q\rangle$ acting on $C = \{x_1, x_2, \ldots, x_m\}$; $m > 3\log^2 n$, G^C is a giant.
OUTPUT: An SGS, constructed from Q (using legal operations only).
Step 1. Begin PERMREP(Q, C);
stop PERMREP(Q, C) when $|\{i : i > \log^2 n, |R_i| \neq 1\}| = 2\log n + \log^2 n$.
Let R be a collection of nontrivial coset representatives such that
$|R \cap R_i| = 1$ for all $i > \log^2 n$, $|R_i| \neq 1$.
Step 2. ORBITALS(Q, C, Q_0).
Step 3. **for** $i := 1$ to $\log^2 n$ **do**
Let $G(i-1) := \langle Q_{i-1} \cup R\rangle$.
Construct coset representatives D_i for $G(i-1)^C$ mod $G(i-1)^C_{x_i}$
($*$ $G(i-1)^C_{x_i}$ is the stabilizer of x_i in $G(i-1)^C$. $*$)
Construct Schreier generators Q_i^* for $G(i-1)^C_{x_i}$.
ORBITALS(Q_i^*, C, Q_i).
Step 4. Compute $f(m), g(m)$, where $f(m) := \min\{\{\}r : p(r) > m^4\}$, and $g(m) := \sum_{i=1}^{f(m)} p_i$.
Step 5. Let λ be any $\neq 1$ element of G.
while $\deg(\lambda) > \log^2 n$ **do**
Choose $B \subset \text{supp}(\lambda), |B| = g(m)$ such that B is independent.
Construct $\tau \in G$ such that τ fixes pointwise B^λ and
$\tau|_B$ consists of cycles of length $p_1, p_2, \ldots, p_{f(m)}$.
$\lambda_1 := [\lambda, \tau]$.
For all $i \leq f(m)$, compute $m(i) :=$ the product of all cycle lengths in λ_1
which are not divisible by p_i.
Choose $i \leq f(m)$ such that $2 \leq \deg(\lambda_1^{m(i)}) < \deg(\lambda)/2$.
Let $\lambda := \lambda_1^{m(i)}$ for this i.
end
Step 6. Construct $\rho \in G$ such that ρ fixes exactly $\deg(\lambda) - 1$ points in $\text{supp}(\lambda)$.
Let $\sigma = [\lambda, \rho]$. ($*$ σ is a 3-cycle $*$)
Step 7. Take conjugates of σ to obtain an SGS for $\text{Alt}(C)$.
If $G^C = \text{Sym}(C)$ then sift an odd permutation to obtain an SGS for G^C.

output the SGS for G^C.
end (THREE_CYCLE).

5.2. Correctness of the subroutine THREE_CYCLE.

LEMMA 5.1. *For each* $1 \le i \le \log^2 n$, $\langle Q_i^* \rangle^{C \setminus \{x_1,\ldots,x_i\}}$ *contains* $\mathrm{Alt}(C \setminus \{x_1,\ldots,x_i\})$.

Proof. It is proved by Theorem 2.4. □

LEMMA 5.2. *Given any* $D \subset C, |D| = d \le \log^2 n$, *and an injection* $f : D \to C$, *it is possible to construct* $\tau \in G$ *such that* $\tau|_D = f$, *and* τ *is a* $2d$*-long product of elements of* $\bigcup_{i=1}^{\log^2 n} D_i$.

Proof. By Lemma 5.1, for all $i \le \log^2 n$ $D_i = \{\alpha(i,j) | i \le j \le m\}$, where $\alpha(i,j)$ fixes $x_1, x_2, \ldots, x_{i-1}$ and moves x_i to x_j. For any distinct $a_1, \ldots, a_d \in C$, let us define recursively $\pi(a_1,\ldots,a_d) = \rho\alpha(d, a_d^\rho)^{-1}$, where $\rho = \pi(a_1,\ldots,a_{d-1})$. Then, for $i \le d$ we have $a_i^{\pi(a_1,\ldots,a_d)} = i$. Let now $D = \{l_1,\ldots,l_d\}$. Then $\tau = \pi(l_1,\ldots,l_d)\pi(f(l_1),\ldots,f(l_d))^{-1}$ is appropriate. □

PROPOSITION 5.3. $f(m) = O(\frac{\log\ m}{\log\ \log\ m})$ *and* $g(m) = O(\frac{\log^2 m}{\log\ \log\ m})$.

Proof. It is proved by the prime number theorem [HW]. □

COROLLARY 5.4. $f(m) = O(\log n)$ *and* $g(m) = O(log^2 n)$.

The following is easily verified.

LEMMA 5.5. *Let* $\pi, \tau \in \mathrm{Sym}(C)$. *Assume that* B *is an independent set with respect to* π *and* $\tau|_{B^\pi}$ *is the identity. Then* $[\pi,\tau]|_B = \tau^{-1}|_B$.

LEMMA 5.6. *Let* $\pi \in \mathrm{Sym}(m)$, $k = \deg(\pi)$. *Suppose* π *contains cycles of each prime length* p_i, $i \le r = f(m)$. *Let* $m(i)$ *be the product of all cycle lengths occurring in* π *which are not divisible by* p_i. *Then* $2 \le \deg(\pi^{m(i)}) < k/4$ *for some* $i \le r$.

Proof. Let $K = \mathrm{supp}(\pi)$. For each $x \in K$, let us consider the set $P(x)$ of those primes p_i dividing the length of the π-cycle through x. Clearly, the product of these primes is $\le k$.

Let $n(i)$ denote the number of points x such that $p_i \in P(x)$. Let us estimate the weighted average W of the $n(i)$ with weights $\log\ p_i$. Recall that the sum of the weights is $\sum \log\ p_i > \log(m^4) = 4 \log\ m$; therefore,

$$W < \sum_{x \in K} \sum_{p_i \in P(x)} \log\ p_i/(4 \log\ m)$$

$$\le (k \log\ k)/(4 \log\ m) \le k/4.$$

We thus infer that $n(i) < k/4$ for some $i \le r$. Clearly, $\pi^{m(i)}$ is not the identity and it fixes all but $n(i)$ points. □

THEOREM 5.7. *The output of THREE_CYCLE*(Q) *is an SGS for* G.

Proof. By Lemma 5.1, Step 3 constructs the first $\log^2 n$ coset-representative sets for a giant. By Corollary 5.4, we can choose independent sets of size $g(m)$ from permutations of degree $> \log^2 n$. By Lemma 5.2, we are able to construct the permutations τ, ρ required in Step 5 and 6. By Lemma 5.5, $\lambda_1|_B$ has the same cycle structure as $\tau|_B$; moreover, $\deg(\lambda_1) \le \deg(\lambda) + \deg(\tau\lambda^{-1}\tau^{-1}) = 2\deg(\lambda)$. Hence, by Lemma 5.6, we can choose $i \le r$ such that $2 \le \deg(\lambda_1^{m(i)}) < \deg(\lambda)/2$. In Step 6, we compute the commutator of two permutations whose supports intersect in exactly one point, whence the commutator is a 3-cycle. Finally, we can obtain permutations which conjugate σ into elements of an SGS by Lemma 5.2. □

5.3. Time complexity of THREE_CYCLE.

LEMMA 5.8. *ORBITALS(P, C, R) runs in $O^\sim(|P||C|^2 + |P|n)$ time.*

Proof. The orbitals of G can be computed in $O(|P||C|^2)$ time. One execution of the repeat loop costs $O^\sim(|R||C|^2)$ for the computation of the orbitals of $\langle R\rangle$, plus $O(|P||C|^2)$ for the computation of the function $\text{last}(\Delta_i)$, plus $O(|C|^2)$ for checking the images Δ_i^σ, plus $O(|P|n)$ for group multiplications to compute σ. The key observation is that we execute the repeat loop only $O(\log|C|)$ times since, at each execution, the new σ increases at least half of the orbitals Δ_i for which the function $\text{last}(\Delta_i)$ is defined. Therefore, after l executions of the repeat loop, the number of "bad" Δ_i's is $\le |C|^2(3/4)^l$. □

LEMMA 5.9. *Suppose that the sum of the different positive integers b_i is $\le m$. Then $\prod b_i \le \exp(O^\sim(\sqrt{m}))$.*

Proof. Choose $b_1 < b_2 < \cdots$ such that $\prod b_i$ is maximal. Then $b_1 \le 4$, for otherwise substituting b_1 by 2 and $b_1 - 2$ the product would increase. Also, for any i, $b_{i+1} - b_i \le 2$; otherwise the product would increase by substituting $b_i + 1$ and $b_{i+1} - 1$ for b_i and b_{i+1}. Moreover, $b_{i+1} = b_i + 2$ for at most one i: if $b_{i+1} \ge b_i + 2$ and $b_{j+1} \ge b_j + 2$ for some $i < j$ then by substituting b_i by $b_i + 1$ and b_{j+1} by $b_{j+1} - 1$ the product would increase. Thus the b_i comprise an initial segment of the natural numbers with the possible omission of 1, 2, 3 and one other number. If $\max\{b_i\} = x$, then $m \ge \sum b_i \ge x(x+1) - 1 - 2 - 3 - (x-1)$, which implies $x \le 2 + \sqrt{2m}$. We have then $\prod b_i < x! = \exp(O^\sim(\sqrt{m}))$. □

THEOREM 5.10. *Suppose that $|Q| = q$ and $\langle Q\rangle^C$ is a giant. Then THREE_CYCLE(Q) constructs an SGS for $\langle Q\rangle^C$ in $O^\sim(qn + mn + m^2q + m^3)$ time.*

Proof. By the prime number theorem, the $\log n$th prime is $O(\log n \log\log n)$; hence the preprocessing phase requires $O^\sim(1)$ time. By Theorem 2.12(b) and the argument already used at the analysis of TEST_GIANT (cf. Lemma 4.5), Step 1 runs in $O^\sim(qn + mn)$. By Lemma 5.8, Step 2 requires $O^\sim(m^2q + qn)$ time and the output Q_0 satisfies $|Q_0| = O(\log m)$. We execute the loop of Step 3 $O^\sim(1)$ times. The coset representative set D_i is obtained in $O(mn)$. The Schreier generators are constructed in $O^\sim(mn)$ time and their number is $|Q_i^*| \le m|Q_{i-1}| = O^\sim(m)$. Using Lemma 5.8 again, Q_i is computed in $O^\sim(m^3 + mn)$; hence the total time requirement of Step 3 is $O^\sim(mn + m^3)$. Step 4 runs in $O^\sim(1)$. Since we decrease the degree of λ at least by a factor 2, the loop of Step 5 is executed $O^\sim(1)$ times. By Lemma 5.2, τ, whence λ_1, is obtained in $O^\sim(n)$. By Lemma 5.9, $m(i)$ is a $\le O^\sim(\sqrt{m})$-digit number; thus, for all $i \le r$, $m(i)$ can be computed in $O^\sim(m)$ time [SS], and $\lambda_1^{m(i)}$ can be constructed in $O^\sim(n\sqrt{m})$. (For all x in the permutation domain, we have to divide $m(i)$ by the length of the cycle through x.) Hence Step 5 requires $O^\sim(n\sqrt{m})$ time. Step 6 runs in $O^\sim(n)$. Finally, by Lemma 5.2, Step 7 requires $O^\sim(mn)$ time. □

COROLLARY 5.11. *Step 4 of the main algorithm runs in $O^\sim(n^3 + sn^2)$ total time.*

Proof. We apply THREE_CYCLE to the action of the stabilizer of some nodes v on the children of v in the SD. As in the proof of Corollary 4.7, denoting by q_v the number of (Schreier) generators for G_v and by m_v the number of children of v, $\sum_v (q_v m_v) = O(sn)$. □

5.4. Las Vegas speedup of THREE_CYCLE. In this section we present a randomized version of THREE_CYCLE with $O^\sim((q+m)n)$ running time. As indicated in the proof of Theorem 5.10, calls to the subroutine ORBITALS were the only parts of the procedure THREE_CYCLE not executable within this tighter time bound. ORBITALS is accelerated by using random subproducts of generators.

DEFINITION 5.12. *Let $G = \langle \tau_1, \tau_2, \ldots, \tau_k \rangle$. A random subproduct of the generators $\tau_1, \ldots, \tau_k$ is an instance of the product $\tau_1^{\varepsilon_1}\tau_2^{\varepsilon_2}\cdots\tau_k^{\varepsilon_k}$ where the ε_i are independent,* 0-1 *valued random variables with* $\mathrm{Prob}(\varepsilon_i = 0) = \mathrm{Prob}(\varepsilon_i = 1) = 1/2$.

The key observation is that a random subproduct of the generators is just as likely to increase an orbital of a subgroup $H \leq G$ as the deterministically constructed element σ in ORBITALS. We make this observation more precise in the following lemma.

LEMMA 5.13. *Let $G = \langle \tau_1, \tau_2, \ldots, \tau_k \rangle \leq \mathrm{Sym}(m)$. Then the expected number of random subproducts of the generators $\tau_1, \ldots, \tau_k$ which generate a subgroup H with the same orbitals as G is $c \log m$.*

Proof. Let H_t be the subgroup generated by the first t random subproducts and let $\sigma = \tau_1^{\varepsilon_1}\tau_2^{\varepsilon_2}\cdots\tau_k^{\varepsilon_k}$ be the $(t+1)$st random subproduct. Let $\{\Delta_i : i \in I\}$ be the orbitals of H_t which are not orbitals in G. For an arbitrary Δ_i, let $l = \mathrm{last}(\Delta_i) = \max\{j : \Delta_i^{\tau_j} \neq \Delta_i\}$. Then

$$\begin{aligned}
\mathrm{Prob}(\Delta_i^{\sigma} \neq \Delta_i) &= \mathrm{Prob}(\Delta_i^{\tau_1^{\varepsilon_1}\cdots\tau_l^{\varepsilon_l}} \neq \Delta_i) \\
&\geq \mathrm{Prob}(\varepsilon_l = 1 | \Delta_i^{\tau_1^{\varepsilon_1}\cdots\tau_l^{\varepsilon_{l-1}}} = \Delta_i)\mathrm{Prob}(\Delta_i^{\tau_1^{\varepsilon_1}\cdots\tau_l^{\varepsilon_{l-1}}} = \Delta_i) \\
&\quad + \mathrm{Prob}(\varepsilon_l = 0 | \Delta_i^{\tau_1^{\varepsilon_1}\cdots\tau_l^{\varepsilon_{l-1}}} \neq \Delta_i)\mathrm{Prob}(\Delta_i^{\tau_1^{\varepsilon_1}\cdots\tau_l^{\varepsilon_{l-1}}} \neq \Delta_i) \\
&= 1/2.
\end{aligned}$$

Hence, with probability $\geq 1/2$, σ enlarges each "bad" orbital of H_t. A standard argument shows that after taking t random subproducts the expected number of "bad" orbitals is $\leq m^2(3/4)^t$. □

The speedup of THREE_CYCLE is straightforward: instead of calling ORBITALS, we take $O(\log m)$ random subproducts of generators. The procedure is Las Vegas since we can check in $O^{\sim}(m^2)$ time whether these random subproducts generate a 2-transitive group.

Chronologically, the idea of random subproducts preceded the subroutine ORBITALS (cf. [BLS88]). Random subproducts are useful far beyond the scope of this paper; for example, in [BCFLS91], [BCFLS95], augmented with other ideas, they provide a $O^{\sim}(n^3)$ *elementary* Monte Carlo SGS construction.

6. Descending the structure domain: Traversing levels. In the previous sections, we discussed the first four (preparatory) steps of the main algorithm. We constructed an extension of the original permutation domain, called the SD, and an ordered partition of the SD such that the pointwise stabilizer G_i of the first i sets is normal in G. The algorithm proceeds by constructing an SGS for successive G_i mod G_{i+1} and finding *normal generators* for G_{i+1} (that is, generators of subgroup whose normal closure in G is G_{i+1}). We describe the construction of these elements through a process of *normal sifting*, which relies on knowledge of presentations for the quotients G_i/G_{i+1}. Our time bounds depend critically on the number of normal generators obtained and, to that end, we indicate how we form concise presentations.

Recall that a *presentation* of a group G is a pair $\langle X \mid \mathcal{R} \rangle$, in which X is a set and $\mathcal{R} \subseteq \mathcal{F}(X)$ ($\mathcal{F}(X)$ denotes the free group on X) such that there is an epimorphism $\phi : \mathcal{F}(X) \to G$ with kernel $\langle \mathcal{R}^{\mathcal{F}(X)} \rangle$. We shall say that the presentation is *induced by* ϕ; in the algorithmic application of presentations, it is typically necessary to specify ϕ along with X and $\mathcal{R}$. The elements of $\mathcal{R}$ are called *relators*.

6.1. Normal sifting. Let

$$G = G_0 \geq G_1 \geq \cdots \geq G_m = N \tag{6.1}$$

be a chain of normal subgroups of G. Let $S_i \subset G_i$ generate G_i mod G_{i+1}, i.e., $G_i = \langle S_i \rangle G_{i+1}$ $(i \leq m-1)$. We call the collection $\{S_i : 0 \leq i \leq m-1\}$ a system of *chain generators* of the series (6.1).

Suppose that

$$G_i/G_{i+1} = \langle X_i \mid \mathcal{R}_i \rangle \tag{6.2}$$

is a presentation of G_i/G_{i+1} induced by $\phi : X_i \to G_i/G_{i+1}$. We say that $S_i \subseteq G_i$ *corresponds* to this presentation if the natural map $G_i \to G_i/G_{i+1}$ yields a bijection $S_i \to \phi(X_i)$. Then, for $w(X_i) \in \mathcal{R}_i$, substitution of S_i for X_i yields an element $w(S_i) \in G_{i+1}$.

Assume that the subgroup chain

$$G = H_0 \geq H_1 \geq \cdots \geq H_f = N \tag{6.3}$$

is a refinement of (6.1): $G_i = H_{j_i}$ $(0 = j_0 < j_1 < \cdots < j_m = f)$. Assume further that a set C_j of right coset representatives of H_{j-1} mod H_j is given for each j, $1 \leq j \leq f$ such that for $j_i + 1 \leq j \leq j_{i+1}$, we have $C_j \subset \langle S_i \rangle$. Such a system will be called *compatible* with the given system $\{S_i\}$ of chain generators of (6.1). Given an element of $g \in G$, we can sift it down along the chain $\{H_j\}$ to obtain a siftee, a member of N. This defines the map sift: $G \to N$.

THEOREM 6.1 (normal sift theorem). *Assume a series of normal subgroups* (6.1) *of the group $G = \langle S \rangle$ is given along with chain generators $\{S_i \mid 0 \leq i \leq m-1\}$ which correspond to presentations* (6.2) *of the factors. Assume a refinement* (6.3) *of* (6.1) *is given along with coset representatives, compatible with the given chain generators. Let Q denote the set of the following elements:*

(a) *S (the set of generators of G);*

(b) *$g^{-1}hg$ for $g \in S$ and $h \in S_i$, $1 \leq i \leq m-1$;*

(c) *$w_i(S_i)$ for all $w_i \in \mathcal{R}_i$, $0 \leq i \leq m-1$.*

Then $N = \langle \mathrm{sift}(Q)^G \rangle$.

Proof. Let $H = \langle \mathrm{sift}(Q)^G \rangle$. Set $\bar{G} = G/H$ and let $\phi : G \to \bar{G}$ be the natural homomorphism. Clearly, $H \leq N$, and therefore $|\bar{G}| \geq |G/N|$. We have to prove that equality holds here. For any subset $U \subset G$, we use $\bar{U}$ to denote $\phi(U)$.

Let $H_i = \langle S_i, S_{i+1}, \ldots, S_{m-1} \rangle$. $(H_m = 1.)$

1. $\bar{H}_0 = \bar{G}$, because $\mathrm{sift}(S) \subset H$ by (a).

2. $\bar{H}_i \triangleleft \bar{G}$, because $\mathrm{sift}(S_i^S) \subset H$ by (b).

3. $|\bar{H}_i/\bar{H}_{i+1}| \leq |G_i/G_{i+1}|$, because $w_i(\bar{S}_i) \in \bar{H}_{i+1}$ for $w_i \in \mathcal{R}_i$ by (c).

It follows that $|\bar{G}| = |\bar{H}_0/\bar{H}_1| \cdots |\bar{H}_{m-1}/\bar{H}_m| \leq |G_0/G_1| \cdots |G_{m-1}/G_m| = |G/N|$. □

6.2. Presentations. The normal sift theorem is applied each time our descent of the structure domain finishes a level. There, we are dealing with quotients G_i/G_{i+1} that act faithfully on L_{i+1}, the $(i+1)$st level of the SD. For our time bounds, we need to ensure that $|\mathcal{R}_i| = O^{\sim}(|L_{i+1}|^2)$.

For a full alternating group, $\mathrm{Alt}(q)$, there is a concise set of at most q relations [Car], cf. [CM, p. 67]. We quote Carmichael's presentation of $\mathrm{Alt}(q)$.

THEOREM 6.2 (see [Car]). *Fix $q \geq 4$. Let $X = \{x, y\}$. Let*

$$\mathcal{R}_{\mathrm{Car}} = \{y^{q-2}, x^3, (yx)^q\} \cup \{(xy^{-k}xy^k)^2 \mid 1 \leq k \leq (q-3)/2\}$$

if q is odd, and

$$\mathcal{R}_{\mathrm{Car}} = \{y^{q-2}, x^3, (yx)^{q-1}\} \cup \{(x^{(-1)^k} y^{-k} x y^k)^2 \mid 1 \le k \le (q-2)/2\}$$

if q is even. Then $\langle X \mid \mathcal{R}_{\mathrm{Car}} \rangle$ is a presentation of $\mathrm{Alt}(q)$.

This extends easily to a presentation of direct products of alternating groups, the situation we uncover at alternating levels of the SD. We use a Carmichael presentation, with a pair of generators, for each factor and enter the relators (commutators) that ensure that the pairs of generators commute.

For the small-group levels, we recall an elementary construction of presentations. Suppose that, for $1 \le j \le f$, C_j is a complete set of right coset representatives for $H_{j-1} \bmod H_j$, where

$$G = H_0 \ge H_1 \ge \cdots \ge H_f = 1.$$

For each $\gamma \in \bigcup_{j=1}^{f} C_j$, associate a symbol x_γ and let X be the collection of these symbols. For any $j \ge k$ and $1 \ne \sigma \in C_j$, $1 \ne \tau \in C_k$,

$$\sigma\tau = \gamma_f \cdots \gamma_{j+1}\gamma_j, \text{ for unique } \gamma_p \in C_p,\ j \le p \le f.$$

Let $w_{\sigma,\tau}$ be the word $x_\tau^{-1} x_\sigma^{-1} x_{\gamma_f} \cdots x_{\gamma_{j+1}} x_{\gamma_j}$ and let $\mathcal{R}$ be the collection of all such words. Then $\langle X \mid \mathcal{R} \rangle$ is a presentation of H.

Let $H = G_i/G_{i+1}$ be a small-level group acting on L_{i+1}, $|L_{i+1}| = m$. Coset representatives in the point stabilizer chain for H are available via PERMREP (Proposition 2.11). We know, however, that H is contained in a direct product of isomorphic primitive groups, this direct product acting as a "small" group on each of its, say r, orbits each of size m/r. Any such orbit includes at most $O^\sim(1)$ points where the point stabilizer chain for H decreases, i.e., where $|C_i| \ne 1$. Furthermore $|C_i| \le m/r$ for all i. It follows that $|X| = O^\sim(m)$ and $|\mathcal{R}| = O^\sim(m^2)$.

7. Descending the structure domain: Small group levels. By the results of section 4, the group $G(w)$ (the action of the stabilizer of the node w in the structure domain on the children of w), is either an alternating or a small group. (A small group is of order $< \exp(9 \log^2 n \log\log n)$.) Moreover, for $w, w' \in L_i$ these groups are isomorphic. We call L_i an *alternating level* if $G(w)$ is alternating for $w \in L_i$, and a *small group level* in the other case. Our objective in this section is to get past a small group level L_{i-1}. Suppose that we have constructed an SGS for G/G_{i-1} and normal generators Q^{i-1} for G_{i-1}. We proceed to constructing an SGS for G_{i-1}/G_i and normal generators Q^i for G_i.

The routine NORMCL(Q^{i-1}, L_i, S) gives us the SGS. A presentation for G_{i-1}/G_i is obtained according to section 6.2, and then normal generators for G_i are constructed according to Theorem 6.1.

Timing analysis for NORMCL. Let $L_{i-1} := \{w_1, w_2, \ldots, w_r\}$, and denote by B_j the children of w_j. Then $L_i = \bigcup_{1 \le j \le r} B_j$ and $|(G_{i-1})^{L_i}| = \exp(O^\sim(|L_{i-1}|))$. Moreover, since G_{i-1} stabilizes L_{i-1} pointwise, $t := \max(|(G_{i-1})_j^{L_i} : (G_{i-1})_{j+1}^{L_i}|) \le |L_i|/r$. (Recall that $(G_{i-1})_j^{L_i}$ denotes the jth subgroup in the pointwise stabilizer chain in the group G_{i-1} acting on the set L_i.) Therefore, by Theorem 2.15, the running time of NORMCL(Q^{i-1}, L_i, S) is $O^\sim\big(|L_{i-1}|n(|Q^{i-1}| + s|L_{i-1}| + |L_i||L_{i-1}|)\big)$.

Number of normal generators obtained. There are $O^\sim(|L_i|)$ coset representatives, so $|Q^i| \le |Q^{i-1}| + O^\sim(s|L_i| + |L_i|^2)$.

Finally we observe that the time to sift each normal generator into G_i is $O^\sim(n|L_{i-1}|)$.

Remark 7.1. If $s > n$ then we may apply NORMCL(Q^{i-1}, L_i, S^*) with $S^* := \bigcup_{j<i-1} S^j$. Since S^* is a set of compatible generators for G/G_{i-1} and Q^{i-1} contains the siftees of S into G_{i-1}, $\langle S\rangle = \langle S^*, Q^{i-1}\rangle$ and $\langle Q^{i-1}\rangle^{\langle S^*\rangle} = \langle Q^{i-1}\rangle^G$. This change improves the timing and the bound on the number of generators, replacing s by n in both expressions.

8. Descending the structure domain: Alternating levels. Suppose that we have constructed an SGS for G/G_{i-1} in Step 5 of the main algorithm and L_{i-1} is an alternating level. In this section, we describe a method to obtain an SGS for G_{i-1}/G_i and normal generators for G_i.

First, we introduce some notation. Let v be a representative node at level L_{i-1}. Level L_i can be partitioned into $L_i = B_1 \dot\cup B_2 \dot\cup \cdots \dot\cup B_r$, $|B_j| = m > 3\log^2 n$ for all j such that for each $w \in L_{i-1}$ the children of w comprise one of the B_j and the point stabilizer G_w acts as $\mathrm{Alt}(B_j)$ on this B_j. We may suppose that B_1 contains the children of v. While computing Schreier generators for G_v, the algorithm constructed $\alpha_2, \ldots, \alpha_r \in G$ such that $B_1^{\alpha_j} = B_j$. We denote by Q^{i-1} the set of normal generators for G_{i-1} constructed by the algorithm and by S the generators of G. Finally, for $\pi \in G_{i-1}$, $\mathrm{length}(\pi) := |\{j : \pi|_{B_j} \neq 1\}|$.

If all elements of Q^{i-1} act trivially on L_i, then $G_i = G_{i-1}$ and there is nothing to do. If there exists $\rho \in Q^{i-1}$ acting nontrivially on $B_{j'}$ for some $j' \leq r$, then, since G_{i-1} contains all conjugates of ρ, G_{i-1} acts as $\mathrm{Alt}(B_{j'})$ on $B_{j'}$; moreover, conjugating by $\alpha_2, \ldots, \alpha_r$ we see that $G_{i-1}^{B_j} = \mathrm{Alt}(B_j)$ for all $j \leq r$. Hence, by Proposition 2.2, G_{i-1}/G_i is isomorphic to $\mathrm{Alt}(m)^k$ for some k. We give an efficient version of Luks's "noncommutative linear algebra" to determine which coordinates of $\prod_{j\leq r} \mathrm{Alt}(B_j)$ are *linked* in the diagonal subgroups. We note that because of the transitive G-action on $\{B_1, \ldots, B_r\}$, the number of $\mathrm{Alt}(B_j)$'s is the same in each linked collection.

8.1. The procedure GIANT_CLOSURE. In Step 4 of the main algorithm, we computed an SGS $P_v \subseteq G$ for $G(v)$, the action of G_v on B_1. However, the elements of P_v are not necessarily in G_{i-1} (that is, they do not necessarily fix *all* nodes at level $i-1$). Here we describe a subroutine which computes an SGS $R \subseteq G_{i-1}$ for $\mathrm{Alt}(B_1)$ given P_v and given an element of G_{i-1} acting nontrivially on B_1.

More precisely, with additional applications in mind, we consider the following situation. The setwise stabilizer $G_{\{C\}}$ of a group G acts on a set C, $|C| \geq 8$, as $\mathrm{Alt}(C)$. The input to GIANT_CLOSURE is $P \subseteq G_{\{C\}}$ such that P is an SGS in this action, and $\rho \in G_{\{C\}}$. The output is R, an SGS for $\mathrm{Alt}(C)$, such that $R \subseteq \langle \rho^{\langle P\rangle}\rangle$; i.e., R is generated by conjugates of ρ by the elements of $\langle P\rangle$. Moreover, we require that there are $\tau, \sigma \in \langle \rho^{\langle P\rangle}\rangle$ such that $w(\tau, \sigma) = 1$ for $w(x, y) \in \mathcal{R}_{\mathrm{Car}}$ (see Theorem 6.2), and $R \subseteq \langle \tau, \sigma\rangle$.

If $C = \{1, 2, \ldots, m\}$, m even then $\tau = (1\ 2\ 3)$, $\sigma = (1\ 2)(3\ 4\ \ldots\ m-1\ m)$ satisfy the relations in Theorem 6.2. If m is odd then we can choose $\tau = (1\ 2\ 3)$, $\sigma = (3\ 4\ \ldots\ m-1\ m)$.

procedure GIANT_CLOSURE$(G, C, \rho, P, R, \tau, \sigma)$
INPUT:C, P, ρ as specified above.
OUTPUT: R, τ, σ.
Step 1. Let $\gamma_1 \in \langle P\rangle$ such that $\gamma_1|_C$ is a 3-cycle not commuting with $\rho|_C$. Compute $\rho_1 = [\rho, \gamma_1]$. $(*\ \deg(\rho_1|_C) \leq 6\ *)$ Take $\gamma_2 \in \langle P\rangle$ such that $|\mathrm{supp}(\rho_1|_C) \cap \mathrm{supp}(\gamma_2|_C)| = 1$. Compute $\rho_2 = [\rho_1, \gamma_2]$. $(*\ \rho_2|_C$ is a 3-cycle $*)$
Step 2. Conjugating ρ_2 with appropriate elements of $\langle P\rangle$, obtain permutations

$\pi_1, \pi_2, \ldots, \pi_{m-2}$ such that $\pi_i|_C = (i\ i+1\ i+2)$. $(*\ \pi_1, \pi_2, \ldots, \pi_{m-2}, \pi_{m-2}^2$ is an SGS for $\mathrm{Alt}(C)\ *)$.
Step 3. Compute τ, σ as specified before the procedure as a product of the π_i's.
Step 4. If m is odd then compute $\sigma\tau$. $(*\ \sigma\tau|_C = (1\ 2\ \ldots\ m-1\ m)\ *)$ R consists of τ and its conjugates by the powers of $\sigma\tau$.

If m is even then compute $\sigma\tau^2$ and $\tau\sigma^{-1}\tau\sigma$. $(*\ \sigma\tau^2|_C = (2\ 3\ \ldots\ m-1\ m)$ and $\tau\sigma^{-1}\tau\sigma|_C = (2\ 3\ 4)\ *)$ R consists of τ, $\tau\sigma^{-1}\tau\sigma$, and the conjugates of $\tau\sigma^{-1}\tau\sigma$ by the powers of $\sigma\tau^2$.
end (GIANT_CLOSURE).

PROPOSITION 8.1. *If group operations in G require $O(n)$ time then GIANT_CLOSURE$(G, C, \rho, P, R, \tau, \sigma)$ computes an SGS for* $\mathrm{Alt}(C)$ *in $O(mn)$.*

Proof. The correctness of the procedure is obvious. Given an SGS for $\mathrm{Alt}(C)$, any element of $\mathrm{Alt}(C)$ can be constructed from it by $O(m)$ group multiplications. Therefore, Steps 1 and 3 require $O(mn)$ time. By Lemma 5.2, a permutation with three prescribed positions can be constructed in $O(n)$ time so Step 2 also runs in $O(mn)$. Finally, we notice that using the result of the conjugation by the previous power of $\sigma\tau$ (or $\sigma\tau^2$), all conjugates in Step 4 can be computed with $O(m)$ group operations. □

8.2. The procedure GIANT_LINK. We obtain an SGS for G_{i-1}/G_i by applying the procedure GIANT_LINK. We use the notation introduced at the beginning of section 8 for the input; the output will be an SGS T and a set S^{i-1} of compatible generators for G_{i-1}/G_i and a set Q^i of normal generators for G_i.

If two coordinates $j, j' \le r$ are not linked in a diagonal action then there exists $\pi \in G_{i-1}$ such that $\pi|_{B_j} \neq 1$ and $\pi|_{B_{j'}} = 1$. In this case, we say that π *witnesses the separation of j from j'*. Note that possession of a witness to the separation of j from j' does not imply possession of a witness to the reverse separation, even though we know that one exists.

GIANT_LINK uses the subroutine GIANT_SEPARATE. The input is an SGS $R_j \subset G_{i-1}$ for $G_{i-1}^{B_j} \cong \mathrm{Alt}(B_j)$ and $\pi_1, \pi_2 \in G_{i-1}$ such that $\pi_l|_{B_j} \neq 1$ for $l = 1, 2$. The output is a single $\pi \in G_{i-1}$ such that, for any coordinate j', if either π_1 or π_2 witnesses the separation of j from j', then π also witnesses this separation.

procedure GIANT_SEPARATE(R_j, π_1, π_2, π)
INPUT:R_j, π_1, π_2 as specified above.
OUTPUT: π.
  if $\pi_1|_{B_j}, \pi_2|_{B_j}$ do not commute
    then $\pi := [\pi_1, \pi_2]$
    else take $\rho \in \langle R_j \rangle$ such that $\pi_1|_{B_j}, \rho^{-1}\pi_2\rho|_{B_j}$ do not commute
      $\pi := [\pi_1, \rho^{-1}\pi_2\rho]$
end (GIANT_SEPARATE).

PROPOSITION 8.2. *GIANT_SEPARATE computes the witness π in $O(n)$ time.*

Proof. The only nontrivial point is that an appropriate $\rho \in \langle R_j \rangle$ can be constructed in $O(n)$ time. If $\pi_1|_{B_j}$ has a fixed point, say $x^{\pi_1} = x$, then conjugate π_2 such that $x^{\rho^{-1}\pi_2\rho} = y$ for some y with $y^{\pi_1} \neq y$. If $\pi_1|_{B_j}$ does not have a fixed point then choose four different points $x, y, z, u \in B_j$ such that $x^{\pi_1} = y$ and $z^{\pi_1} = u$. Conjugate π_2 such that $y^{\rho^{-1}\pi_2\rho} = u$ and $x^{\rho^{-1}\pi_2\rho} \neq z$. Since we described the value of ρ at ≤ 4 points, such ρ can be obtained in $O(n)$ time by Lemma 5.2. Note that $\pi_2|_{B_{j'}} = 1$ implies $\rho^{-1}\pi_2\rho|_{B_{j'}} = 1$. □

In the first three steps of GIANT_LINK, we compute a subgroup of G_{i-1} which acts as the full alternating group on each B_j. In Step 4, we obtain witnesses for all pairs not linked by this subgroup and then compute a single witness for each B_j. In the loop described in Steps 5–7, we obtain additional elements of the subgroup G_{i-1} (until we have a collection that fully generates G_{i-1}/G_i). First, in Step 5, until the linked collections have the same length, we conjugate the shortest collection into all positions, necessarily breaking up some links in the longer collections. Step 7 ensures that we have done all the link breaking that is implied by the subgroup at hand and, if so, that the subgroup is normalized (mod G_i) by G; failure of either test produces a new witness to separation in Step 7, and the loop is repeated. If the tests are passed, we can specify Q^i (Step 8).

procedure GIANT_LINK$(L_i, Q^{i-1}, S, P_v, \{\alpha_2, \ldots, \alpha_r\}, Q^i, S^{i-1}, T)$
INPUT: $L_i = B_1 \dot\cup \cdots \dot\cup B_r, Q^{i-1}, S, P_v, \{\alpha_2, \ldots, \alpha_r\}$ as specified above.
OUTPUT: Q^i, S^{i-1}, T.
Step 1. take $\rho \in Q^{i-1}$, $\rho|_{B_j} \neq 1$ for some j; Compute $\rho_1 := \alpha_j \rho \alpha_j^{-1}$.
Step 2. GIANT_CLOSURE$(G, B_1, \rho_1, P_v, R_1, \tau_1, \sigma_1)$.
Step 3. **for** $j := 2$ to r **do**
compute $R_j := \alpha_j^{-1} R_1 \alpha_j, \sigma_j := \alpha_j^{-1}\sigma_1\alpha_j$, $\tau_j := \alpha_j^{-1}\tau_1\alpha_j$, and $\rho_j := \alpha_j^{-1}\rho_1\alpha_j$.
Step 4. **for** $j := 1$ to r **do**
Collect the following elements of G_{i-1} in a set Σ:
the siftees of $Q^{i-1} \cup \{\sigma_{j'}, \tau_{j'} : 1 \le j' \le r\}$ through R_j
$w(\tau_j, \sigma_j)$ for all $w(x, y) \in \mathcal{R}_{\mathrm{Car}}$ ($*$ see Theorem 6.2 $*$)
for $\sigma \in \Sigma$ **do**
for all coordinates j' for which σ witnesses the separation of j' from j but this separation is not witnessed by the current $\rho_{j'}$ **do**
GIANT_SEPARATE$(R_{j'}, \rho_{j'}, \sigma, \rho_{j'})$.
Step 5. **while** there exist j, j' with length$(\rho_j) \neq$ length$(\rho_{j'})$ **do**
take ρ_j with minimal length
for $j' := 1$ to r **do**
GIANT_SEPARATE$(R_{j'}, \rho_{j'}, \alpha_{j'}^{-1}\alpha_j\rho_j\alpha_j^{-1}\alpha_{j'}, \rho_{j'})$
if the lengths of all $\rho_{j'}$, $1 \le j' \le r$, are equal **and**
there exist j', j'' such that $\rho_{j''}$ witnesses a separation of j' (from some j''') that is *not* witnessed by $\rho_{j'}$
then for one such pair j', j''
GIANT_SEPARATE$(R_{j'}, \rho_{j'}, \rho_{j''}, \rho_{j'})$.
Step 6. **for** $j := 1$ to r **do**
GIANT_CLOSURE$(G, B_j, \rho_j, R_j, R_j, \tau_j, \sigma_j)$.
Step 7. **for** $j := 1$ to r **do**
Collect the following elements of G_{i-1} in a set Σ:
the siftees of $Q^{i-1} \cup \{\sigma_{j'}, \tau_{j'} : 1 \le j' \le r\}$ through R_j
$w(\tau_j, \sigma_j)$ for all $w(x, y) \in \mathcal{R}_{\mathrm{Car}}$
the siftees of $\{\sigma_{j'}^{\alpha}, \tau_{j'}^{\alpha} : \alpha \in S, 1 \le j' \le r\}$ through R_j
for $\sigma \in \Sigma$ **do**
for all coordinates j' for which σ witnesses the separation of j' from j but this separation is not witnessed by the current $\rho_{j'}$ **do**
GIANT_SEPARATE$(R_{j'}, \rho_{j'}, \sigma, \rho_{j'})$
if any of the ρ_j were changed in this step **then goto** Step 5.

Step 8. Let $J \subseteq \{1, 2, \ldots, r\}$ consist of a representative j from each linked collection of coordinates.
output $S^{i-1} := \{\tau_j, \sigma_j : j \in J\}$;
output $T := \bigcup\{R_j : j \in J\}$;
collect in Q^i the following elements of G_i:
the siftees of Q^{i-1} through the SGS T
for all distinct $j, j' \in J$, the commutators $[\sigma_j, \tau_{j'}], [\sigma_j, \sigma_{j'}], [\tau_j, \tau_{j'}]$, $[\tau_j, \sigma_{j'}]$
for all $j \in J$, $w(\tau_j, \sigma_j)$ for all $w(x, y) \in \mathcal{R}_{\mathrm{Car}}$
the siftees of $\{\alpha^{-1}\tau_j\alpha, \alpha^{-1}\sigma_j\alpha : \tau_j, \sigma_j \in S^{i-1}, \alpha \in S\}$ through the SGS T;
output Q^i.
end (GIANT_LINK).

8.3. Correctness and time requirement of GIANT_LINK.

THEOREM 8.3. *The outputs T, S^{i-1} of GIANT_LINK are, respectively, an SGS and a set of compatible generators for G_{i-1}/G_i. The collection Q^i is a set of normal generators for G_i.*

Proof. We first claim that after the execution of Step 4, for all $1 \le j \le r$, ρ_j witnesses the separation of j form any j' that is implied by the group $H = \langle Q^{i-1} \cup \{\tau_j, \sigma_j : 1 \le j \le r\}\rangle$. The claim follows from Theorem 6.1 with $G := H, N := H_{B_j}$ and $m := 1$ (because normal generators for the kernel of the action on B_j suffice to witness possible separations of any j' from j). Thus, in particular, the distinct classes $\mathcal{C}_j = \{j' \mid \rho_j|_{B_{j'}} \neq 1\}$, $1 \le j \le r$ partition $\{1, \ldots, r\}$.

When we emerge from Step 5, the distinct classes among the $\mathcal{C}_j$ are again disjoint (a nontrivial intersection would be picked up by the last **if** statement, which would reduce the length of $\rho_{j'}$ for some j' in the intersection) and they now have the same size.

In Step 7, if the sifting of Q^{i-1}, $\sigma_{j'}, \tau_{j'}$ and the elements $w(\tau_j, \sigma_j)$ witness no new separations, then we know that the ρ_j, $1 \le j \le r$, witness all separations implied by elements of the group $H = \langle Q^{i-1} \cup \{\tau_j, \sigma_j : 1 \le j \le r\}\rangle$ (by the argument for Step 4). Furthermore, we know that H acts on L_i as a direct product of alternating groups, exactly one alternating group in each still-linked class of coordinates. If so, the successful sifting of the collection of $\sigma_{j'}^{\alpha}, \tau_{j'}^{\alpha}$ guarantees that H is invariant (mod G_i) under the action of G.

The claims about T and S^{i-1} are now clear. The fact that Q^i is a set of normal generators of G_i then follows from Theorem 6.1 (with the chain of normal subgroups $G = G_0 \ge G_1 \ge \cdots \ge G_{i-1} \ge G_i = N$). □

THEOREM 8.4. *Let $s = |S|$. Then GIANT_LINK$(L_i, Q^{i-1}, S, P_v, \{\alpha_2, \ldots, \alpha_r\}, Q^i, S^{i-1}, T)$ runs in time $O^{\sim}(|Q^{i-1}||L_i|n + s|L_i|^2 n)$ and $|Q^i| \le |Q^{i-1}| + O(s|L_i| + |L_i|^2)$.*

Proof. In Step 1, we can pick an appropriate ρ in $O(|Q^{i-1}|rm)$ and ρ_1 is computed in $O(n)$ time. By Proposition 8.1, Step 2 requires $O(mn)$ time and Step 3 can be executed in $O(rmn)$.

In Step 4, we sift $|Q^{i-1}| + 2r$ elements through r SGS's, requiring $O(|Q^{r-1}|rmn + r^2mn)$ total time; moreover, we compute $O(rm)$ defining relations, in $O(rmn)$ total time. Altogether for all $1 \le j \le r$ we place $O(|Q^{i-1}|r + r^2)$ elements into Σ; for each of these, $O(mr)$ time suffices to check whether it breaks some new links. The total cost of calls of the subroutine GIANT_SEPARATE in Step 4 is at most $O(r^2n)$ since for each pair j, j' we call GIANT_SEPARATE at most once. Hence the total cost of

Step 4 (using that $r \leq n$) is $O(|Q^{r-1}|rmn + r^2mn)$.

We enter the **while** loop of Step 5 at most r times since the minimum length of ρ_j decreases at each call. (Within the **while** loop, the length of each $\rho_{j'}$ decreases at least to the previous minimum.) Hence calls of GIANT_SEPARATE in Step 5 cost $O(r^2n)$. The **if** statement can also be executed within this time bound, since all we have to check is whether the sets $\mathcal{C}_j$ (see the proof of Theorem 8.3) define a partition of $\{1, 2, \ldots, r\}$.

Each time Step 6 is executed, all the ρ_j are of the same length. The length in any round is necessarily a divisor of the length in the previous round, so Step 6 is executed $\leq \log r$ times. By Proposition 8.1, one execution costs $O(mrn)$.

Step 7 is executed always after Step 6, i.e., $\leq \log r$ times, and one execution is similar to Step 4 with the additional sifting of $O(sr)$ conjugates (by S) through the r SGS's for a total timing of $O(|Q^{r-1}|rmn + sr^2mn)$.

Finally, Step 8 runs in $O(|Q^{i-1}|mrn + r^2n + srmn)$. Only the term $O(srmn)$ (instead of $O(sr^2mn)$) requires additional explanation: each conjugate $\alpha^{-1}\tau_j\alpha$, $\alpha \in S$ acts nontrivially in only one of the linked collections of alternating groups so sifting costs only $O(mn)$. Noting that $|L_i| = mr$, the proof is complete. □

Remark 8.5. If $s > n$ then we may use the set $S^* = \bigcup_{j<i-1} S^j$ instead of S as input of GIANT_LINK, replacing the term s by n in both the running time and number of generators created. Correctness is proved by the argument in Remark 7.1.

9. Proof of the main results. In this section, we finish the proof of Theorem 1.1 and sketch two other versions of the algorithm: one with reduced memory requirement (and same time efficiency as the original) and an elementary version with $O^{\sim}(n^{4.5})$ running time.

9.1. Proof of Theorem 1.1. The algorithm described in sections 3–8 computed an SGS for the input group $G = \langle S \rangle \leq \mathrm{Sym}(n)$, $|S| = s$; we have to analyze the running time.

By Lemma 3.1, Corollary 4.7, Proposition 4.8, and Corollary 5.11, the first four steps of the main algorithm run within $O^{\sim}(n^3 + sn^2)$. By the analysis in section 7 and Theorem 8.4, the number of normal generators created while processing level L_{i-1} is $O^{\sim}(s|L_i| + |L_i|^2)$ (in addition to the $|Q^{i-1}|$ normal generators for G_{i-1}). Hence $|Q^i| = O^{\sim}(n^2 + sn)$ for all i and, by section 7 and Theorem 8.4, Step 5 runs in $O^{\sim}(n^4 + sn^3)$.

If the $O^{\sim}(sn^3)$ term becomes dominant, i.e., $s > n$, then we modify the procedure according to Remarks 7.1, 8.5, and the running time drops down to $O^{\sim}(n^4 + sn^2)$. Finally, if sn^2 dominates n^4, i.e., $s > n^2$, then we begin the algorithm by reducing the number of generators to $O(n^2)$ in $O(sn^2)$ time. This can be achieved by sifting the elements of S into (the originally empty) coset representative table with respect to the point stabilizer chain of the permutation domain (cf. procedure SIFT in section 2.7). In any case, we can achieve the claimed $O(n^4 \log^c n + sn^2)$ running time with no logarithmic factors multiplying sn^2.

We turn to the proof of claims (b)–(e) in Theorem 1.1. The order of G is easily computed as the product of sizes of coset representative sets. Although the SGS constructed by the algorithm can be used directly for membership testing by extending the action of a candidate permutation to the SD and there is a method to compute pointwise set stabilizers from it (developed for the parallel procedure in [BLS87]), it is easier to use a result of Brown, Finkelstein, and Purdom [BFP]. They provide an $O(n^3)$ base-change algorithm for converting strong generating sets with respect

to point stabilizer chains along different orderings of the permutation domain. The base-change algorithm outputs the SGS in Jerrum's compact format.

Finally, we observe that the normal sift theorem (Theorem 6.1) essentially provides the scheme for proving (e). Note that, with the descent of the SD complete, the chain generators generate G, so we may assume $S = \bigcup_i S_i$. We associate a symbol x_π to every element π of S and let X denote the collection of these. Each coset representative $\rho \in R_j$ (see notation and discussion preceding the theorem) is representable as a word in S and there is a corresponding word $w(\rho)$ in X. The elements to be sifted in Theorem 6.1 (a),(b),(c) are given as words in S, and so each τ corresponds naturally to a word $w'(\tau)$ in X. Sifting τ can be interpreted as expressing τ canonically as a word $\rho_1 \cdots \rho_l$. From each such sift we derive a relation $w'(\tau)^{-1} w(\rho_1) \cdots w(\rho_l)$ and denote the collection of these by $\mathcal{R}$. Then $\langle X \mid \mathcal{R} \rangle$ is a presentation of G.

9.2. Reducing the memory requirement. The algorithm, as presented in sections 3–8, requires $O^\sim(n^3 + sn^2)$ space. Here we indicate how to reduce the memory requirement to $O^\sim(n^2 + sn)$.

The first four steps of the main algorithm run within this tighter bound. The problem arises because of the top-down approach in Step 5 since, eventually, we accumulate $O^\sim(n^2 + sn)$ normal generators. On the other hand, the SGS we build occupies only $O^\sim(n^2)$ space. The solution is to build the output SGS T simultaneously on all levels. We call T *up-to-date* on level i if $\langle T \cap G_j \rangle$ is a normal subgroup of G for all $j \geq i$. We work always at the lowest level (i.e., greatest index i) which is not up-to-date.

We start executing Step 5 at level 0 as before. The difference is that working on level i, whenever the algorithm produces a normal generator ψ for G_{i+1}, we sift ψ *immediately* into the SGS already constructed. If ψ factors completely then it can be discarded; if it has a nontrivial siftee on some lower level j then we suspend the execution on level i and jump down to level j.

On small levels, we execute exactly the same steps as in the original algorithm (possibly interrupted by some computations on lower levels). On alternating levels, we may execute GIANT_LINK $O(\log n)$ times, discovering smaller and smaller linked collections of subgroups. There are no more than $O(\log n)$ executions since the lengths of linked collections are divisors of each other. This extra work may add a $\log n$ factor to a lower-order term in the running time.

9.3. An elementary version. Two elementary estimates on the order of primitive groups enable us to break the $O(n^5)$ barrier by an elementary, $O^\sim(n^{4.5})$ algorithm. One of them is Pyber's estimate (cf. Theorem 2.5) on the order of nongiant 2-transitive groups and the other one is due to Babai.

THEOREM 9.1 (see [Ba]). *Let $G \leq \mathrm{Sym}(n)$ be primitive; G is not a giant. Then $|G| \leq \exp(O^\sim(\sqrt{n}))$.*

ELEMENTARY ALGORITHM.
INPUT: a set S of generators for $G \leq \mathrm{Sym}(A)$, $|S| = s$.
Step 1. Construct an SF and choose a representative v in each orbit of the SF. For all such v, construct Schreier generators for G_v.
Step 2. For these representatives, use TEST_GIANT to decide whether $G(v)$ is a giant.
By inserting new levels after symmetric levels, obtain the SD. Compute the node stabilizers G_w as in Step 1 for representatives of G-orbits of the SD. Let $(L_0, L_1, \ldots, L_m)$ be the levels of the SD.

Step 3. For each node v representing an alternating level in the SD, construct an SGS for $G(v)$.

Step 4. **for** $i := 1$ to m **do**
if L_{i-1} is an alternating level
then construct SGS for G_{i-1}/G_i, normal generators for G_i as in section 8
else construct SGS for G_{i-1}/G_i, normal generators for G_i as in section 7

end (ELEMENTARY ALGORITHM).

We have to modify the stopping condition in TEST_GIANT and in the first step of THREE_CYCLE to accommodate the weaker bound in Theorem 2.5. This change adds only a logarithmic factor to a low-order term in the running time. Since NATURAL_ACTION is eliminated from this algorithm, correctness is elementary. However, primitive groups on small levels may be of the size allowed in Theorem 9.1, adding a factor $\sqrt{n}$ in the analysis of section 7.

Acknowledgment. We are indebted to the referees for their careful work and for suggestions to improve the presentation.

REFERENCES

[At] M. D. Atkinson, *An algorithm for finding the blocks of a permutation group*, Math. Comp., 29 (1975), pp. 911–913.

[Ba] L. Babai, *On the order of uniprimitive permutation groups*, Ann. of Math., 113 (1981), pp. 553–568.

[BCFLS91] L. Babai, G. Cooperman, L. Finkelstein, E. M. Luks, and Á. Seress, *Fast Monte Carlo algorithms for permutation groups*, in Proc. 23rd ACM Symposium on the Theory of Computing, 1991, pp. 90–100.

[BCFLS95] L. Babai, G. Cooperman, L. Finkelstein, E. M. Luks, and Á. Seress, *Fast Monte Carlo algorithms for permutation groups*, J. Comput. System Sci., 50 (1995), pp. 296–308.

[BLS87] L. Babai, E. M. Luks, and Á. Seress, *Permutation groups in NC*, in Proc. 19th ACM STOC, 1987, pp. 409–420.

[BLS88] L. Babai, E. M. Luks, and Á. Seress, *Fast management of permutation groups*, in Proc. 29th IEEE Foundations of Computer Science, 1988, pp. 272–282.

[BLS93] L. Babai, E. M. Luks, and Á. Seress, *Computing composition series in primitive groups*, in Groups and Computation, DIMACS Series in Discrete Mathematics 11, 1993, pp. 1–15.

[BLS] L. Babai, E. M. Luks, and Á. Seress, *Fast Management of Permutation Groups* II, in preparation.

[BS87] L. Babai and Á. Seress, *On the degree of transitivity of permutation groups: A short proof*, J. Combinatorial Theory Ser. A, 45 (1987), pp. 310–315.

[BS88] L. Babai and Á. Seress, *On the diameter of Cayley graphs of the symmetric group*, J. Combin. Theory Ser. A, 49 (1988), pp. 175–179.

[Bos] R. C. Bose, *Strongly regular graphs, partial geometries, and partially balanced designs*, Pacific J. Math., 13 (1963), pp. 389–419.

[BFP] C. Brown, L. Finkelstein, and P. Purdom, *A new base change algorithm for permutation groups*, SIAM J. Comput., 18 (1989), pp. 1037–1047.

[Cam] P. J. Cameron, *Finite permutation groups and finite simple groups*, Bull. London Math. Soc., 13 (1981), pp. 1–22.

[Car] R. Carter, *Simple Groups of Lie Type*, Wiley, London, 1972.

[CKS] C. W. Curtis, W. M. Kantor, and G. L. Seitz, *The* 2*-transitive permutation representations of the finite Chevalley groups*, Trans. Amer. Math. Soc., 218 (1976), pp. 1–57.

[CM] H. S. M. Coxeter and W. O. J. Moser, *Generators and Relations for Discrete Groups*, 3rd ed., Springer-Verlag, New York, 1972.

[Del] P. Delsarte, *An algebraic approach to the association schemes of coding theory*, Philips Research Report Supplement, 10 (1973).

[FHL] M. L. Furst, J. Hopcroft, and E. M. Luks, *Polynomial time algorithms for permutation groups*, in Proc. 21st IEEE Foundations of Computer Science, 1980, pp. 36–41.

[Go] D. Gorenstein, *Finite Simple Groups and Their Classification*, Academic Press, New York, 1986.

[Ha] M. Hall, Jr., *The Theory of Groups*, Macmillan, New York, 1959.

[HW] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 5th ed., Clarendon Press, Oxford, 1979.

[Je82] M. Jerrum, *A compact representation for permutation groups*, in Proc. 23rd IEEE Foundations of Computer Science, 1982, pp. 126–133.

[Je86] M. Jerrum, *A compact representation for permutation groups*, J. Algorithms, 7 (1986), pp. 60–78.

[Jo] C. Jordan, *Nouvelles recherches sur la limite de transitivité des groupes qui ne contiennent pas le groupe alterné*, Journ. de Mathématiques, 1 (1895), pp. 35–60.

[Kn] D. E. Knuth, *Efficient representation of perm groups*, Combinatorica, 11 (1991), pp. 33–44.

[Li] M. W. Liebeck, *On minimal degrees and base sizes of primitive groups*, Arch. Math., 43 (1984), pp. 11–15.

[Lu82] E. M. Luks, *Isomorphism of graphs of bounded valence can be tested in polynomial time*, J. Comput. System Sci., 25 (1982), pp. 42–65.

[Lu86] E. M. Luks, *Parallel algorithms for permutation groups and graph isomorphism*, in Proc. 27th IEEE Foundations of Computer Science, 1986, pp. 292–302.

[Lu87] E. M. Luks, *Computing the composition factors of a permutation group in polynomial time*, Combinatorica, 7 (1987), pp. 87–99.

[MS] F. J. Macwilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, North–Holland, Amsterdam, 1978.

[Py] L. Pyber, *On the orders of doubly transitive permutation groups, elementary estimates*, J. Combin. Theory Ser. A, 62 (1993), pp. 361–366.

[SS] A. Schönhage and V. Strassen, *Schnelle Multiplikation Großer Zahlen*, Computing, 7 (1971), pp. 281–292.

[Sc] L. L. Scott, *Representations in characteristic p*, in Proc. Santa Cruz Conf. on Finite Groups, AMS, Providence, RI, 1980, pp. 319–322.

[Si67] C. C. Sims, *Graphs and finite permutation groups*, Math. Z., 95 (1967), pp. 76–86.

[Si70] C. C. Sims, *Computational methods in the study of permutation groups*, in Computational Problems in Abstract Algebra, J. Leech, ed., Pergamon Press, Elmsford, NY, 1970, pp. 169–183.

[Wi] H. Wielandt, *Finite Permutation Groups*, Academic Press, New York, 1964.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1343–1362, October 1997

004

PARAMETERIZED DUPLICATION IN STRINGS: ALGORITHMS AND AN APPLICATION TO SOFTWARE MAINTENANCE*

BRENDA S. BAKER†

Abstract. As an aid in software maintenance, it would be useful to be able to track down duplication in large software systems efficiently. Duplication in code is often in the form of sections of code that are the same except for a systematic change of parameters such as identifiers and constants. To model such parameterized duplication in code, this paper introduces the notions of parameterized strings and parameterized matches of parameterized strings. A data structure called a parameterized suffix tree is defined to aid in searching for parameterized matches. For fixed alphabets, algorithms are given to construct a parameterized suffix tree in linear time and to find all maximal parameterized matches over a threshold length in a parameterized p-string in time linear in the size of the input plus the number of matches reported. The algorithms have been implemented, and experimental results show that they perform well on C code.

Key words. string matching, pattern matching, duplication

AMS subject classifications. 68Q25, 68Q20, 68R15

PII. S0097539793246707

1. Introduction.

In a large ongoing systems project, introduction of new features and code maintenance by large staffs of programmers may result in code that includes many duplicated sections. Such duplication still occurs even though it has long been known that copying code may make the code larger, more complex, and more difficult to maintain. For example, when a new feature is introduced, rather than risk breaking a working feature by making a major revision, a programmer might choose to leave the old section of code untouched, and to add another slightly modified copy of it for the new feature. A bug fix might also be handled by copying and modifying the code, for example, if the original programmer omitted handling of special cases. The copies might be further copied and modified as time goes on. With time, the amount of duplication in a system can become substantial and can complicate maintenance.

While some of the duplication in a software system may involve sections of code that are identical, much of the duplication involves sections of code that are not identical, but are the same except for a systematic change of parameters such as identifiers and constants. For example, each occurrence of *first*, *last*, 0, and *fun* in one section may be replaced by *init*, *final*, 1, and *g*, respectively, in the other section; this kind of correspondence between sections of code is called a *parameterized match* [Bak1]. [Bak1] describes a program *dup* that finds all maximal parameterized matches over a threshold length in C code; application of *dup* to a million-line subsystem of a production system revealed that 21% of the lines were involved in parameterized matches of at least 30 lines (excluding comments and white space).

This paper formalizes the notion of parameterized matches for code in terms of *parameterized strings*, or *p-strings*, which are strings over two alphabets: an alphabet of constant symbols and an alphabet of parameter symbols. Two parameterized

*Received by the editors April 2, 1993; accepted for publication (in revised form) September 29, 1995. Some of the results in this paper appear in the *Proc. of the 25th Annual ACM Symposium on Theory of Computing*, ACM, New York, 1993, pp. 71–80.

http://www.siam.org/journals/sicomp/26-5/24670.html

†Bell Laboratories, Lucent Technologies, 700 Mountain Avenue, Room 2C-457, Murray Hill, NJ 07974 (bsb@research.bell-labs.com).

strings are a *parameterized match*, or *p-match*, if they are the same except for a one-to-one correspondence between the parameter symbols occurring in them. For example, *axbxyazyx* and *aubuvaxvu* are a p-match where the one-to-one correspondence maps the x, y, and z of the first p-string into the u, v, and x, respectively, of the second p-string.

For use in searching p-strings, we define a new data structure called a *parameterized suffix tree*, or *p-suffix tree*. We show that a p-suffix tree can be built in time and space $O(n)$, where n is the length of the input, if the alphabets are fixed. Given a pattern p-string P of length m and a text p-string T of length n over fixed alphabets, a p-suffix tree for T can be used to determine in $O(n+m)$ time and $O(n)$ space whether P has a p-match in T. An algorithm is given that finds all maximal p-matches over a threshold length in a p-string S in time $O(n+r)$, where n is the length of the input and r is the number of matches reported, by searching a p-suffix tree constructed for S. The algorithms for constructing p-suffix trees and for reporting all maximal p-matches over a threshold length have been implemented. Experiments show that the program performs well on C code input.

The algorithms in this paper improve upon the somewhat ad hoc method of finding parameterized matches in *dup*. The parameterized-matching algorithm implemented in *dup* operates as follows: it transforms all parameter candidates such as identifiers and constants to the same symbol P, finds exact matches in the transformed code, and then checks the exact matches for possible parameterized matches; the worst-case running time is not a function of the size of the input and number of p-matches, or even of the size of the input and the total length of the p-matches. In some cases, as many as 99% of the exact matches found do not correspond to parameterized matches.

P-suffix trees are a generalization of suffix trees for strings [McC, Ukk, We], but are more dynamic in that each access to an input symbol requires a transformation based on the depth in the tree. The algorithm described here for building p-suffix trees is based on the McCreight algorithm for building suffix trees [McC]; however, the original algorithm and the concept of suffix links used in it must be modified to allow for the dynamic way in which p-strings are handled and the failure of a key property of strings to generalize for p-strings. The algorithm for finding all maximal p-matches over a threshold length is a generalization of the suffix-tree-based algorithm implemented in *dup* for finding all exact matches over threshold length in strings [Bak1, Bak2]; again, the generalization is not straightforward because of the dynamic nature of p-suffix trees.

The generalization of suffix trees to p-strings is related to Giancarlo's generalization of suffix trees to L-strings [G] in that both have to deal with the failure of the same key property of strings to generalize. In other respects, L-strings and p-strings behave differently, because L-strings do not obey the restricted form of this property that can be proved for p-strings. Consequently, the linear bounds obtainable for constructing p-suffix trees with fixed alphabets do not appear to be obtainable for constructing L-suffix trees for L-strings.

Four other methods have been attempted for finding duplication in code: (1) string pattern matching has been applied to strings encoding the call graph and statistics about characteristics such as use of operators to detect student plagiarism [Ja]; (2) signal processing techniques combined with a graphical user interface have been used to find approximate duplication by eye [CH]; (3) exhaustive search was used on parse trees to identify identical subtrees or subtrees related by change of

parameter, but was found to be unsuccessful because of time and space usage [Jo]; and (4) data flow analysis and safe approximation techniques have been proposed as a basis for comparing program components in a restricted programming language [Ho].

Parameterized matching is reminiscent of unification [GeN], where the goal is to determine whether two expressions can be made equivalent via substitutions for variables, but unification differs in three ways from our problem: the domain is expressions (terms) rather than strings, terms (rather than just variables) are substituted for variables, and there is no notion of matching just parts of terms.

A parameterized match is a kind of approximate match, but is very different from the standard definition in which two strings are an approximate match if they are within a specified edit distance (number of insertions, deletions, or substitutions) from each other, as studied, for example, in [CL, GG]. Even exact or approximate matches to regular expressions [Aho, MM, WM] do not involve any notion of relating repeated occurrences of corresponding (but different) symbols.

The UNIX grep pattern-matching program and ed editor [KP] allow a pattern to be a restricted regular expression with backreferencing to refer to parts of the text matching earlier parts of the pattern, as described in [Aho]. This problem is NP-complete [Aho], and the algorithms implemented are undocumented, but are based on backtracking and do not correctly implement the definitions [Hu92]. The grep/ed usage of backreferencing is not comparable with our definitions: in our terminology, they have no way of requiring that distinct parameters should match different substrings, while we have no way of allowing them to match the same substrings. These programs also do not address the problem of finding all duplication over a threshold length.

The paper is organized as follows. Section 2 defines parameterized strings, parameterized matches, and parameterized suffix trees, and shows how parameterized suffix trees can be used for parameterized pattern matching. The algorithm for constructing a parameterized suffix tree is given in section 3. Section 4 gives the algorithm for reporting all parameterized matches over a threshold length in a p-string. An implementation of the algorithms and some experimental results from applying them to C code are described in section 5. Section 6 discusses time bounds for the algorithms for variable alphabets and directions for further research.

2. Parameterized strings, parameterized matches, and parameterized suffix trees. In this section, we introduce parameterized strings, parameterized matches, and parameterized suffix trees (p-suffix trees), and show how parameterized suffix trees can be used for parameterized pattern matching. We assume a RAM model of computation with the uniform cost criterion [AHU].

Throughout, Σ will be a fixed finite alphabet of constant symbols and Π will be a fixed finite alphabet of parameter symbols; i.e., the sizes of Σ and Π are $O(1)$. We assume that Σ and Π are disjoint from each other and the set of nonnegative integers, that symbols are ordered and can be compared in constant time, and that symbols of Π can be used to index into an array in constant time.

DEFINITION. *A string of symbols in* $(\Sigma \cup \Pi)^*$ *is called a* parameterized string *or* p-string. *Two p-strings are a* parameterized match, *or* p-match, *if one p-string can be transformed into the other by renaming the parameters via a one-to-one function whose domain is the set of parameter symbols occurring in one p-string and whose range is the set of parameter symbols occurring in the other p-string.*

For example, if x, y, and v are parameter symbols and a, b, and c are constant symbols, then $S_1 = axaybxycby$ and $S_2 = ayavbyvcbv$ are a p-match, where x and y of S_1 and renamed as y and v, respectively, in S_2.

Determining whether two entire p-strings are a p-match is straightforward, as follows. Scan the two p-strings left to right, while constructing a table giving the one-to-one correspondence, to see if any mismatches are found between symbols. In addition to mismatches in length, mismatches can be between different non-parameters, between a parameter and a non-parameter, or between two parameters, at least one of which has already been made to correspond to a different parameter. Given our definitions, checking for mismatches can be done in time linear in input length n and space $O(|\Pi|)$, by constructing a table for the one-to-one correspondence. Unfortunately, this approach does not conveniently generalize to pattern matching.

Instead, we use a procedure *prev*, that chains together occurrences of the same parameter, to obtain a string in $(\Sigma \cup \mathrm{N})^*$, where N is the set of nonnegative integers. For each parameter, the leftmost occurrence is represented by a 0, and each successive occurrence is represented by the difference in position compared to the previous occurrence. A number representing such a difference in position is called a *parameter pointer*. For example, if u, v, x, and y are parameter symbols and a and b are constant symbols, then $prev(abuvabuvu) = ab00ab442 = prev(abxyabxyx)$. (Each parameter pointer is a single digit here.)

Since symbols of Π can be used to index into a table, computation of *prev* can be done in time linear in input length and space linear in $|\Pi|$ by means of a table containing the position of last occurrence of each parameter symbol encountered. Proving the following proposition is straightforward from the definitions.

PROPOSITION 1. *P-strings S and $\overline{S}$ are a p-match if and only if $prev(S) = prev(\overline{S})$.*

DEFINITION. *Define the ith p-suffix of a p-string $S = b_1 b_2 \dots b_n$ to be $psuffix(S, i) = prev(b_i b_{i+1} \dots b_n)$. Define $prefix(S, i, j) = prev(b_i b_{i+1} \dots b_j)$, for $i \le j$. Define $prefix(S, i, j)$ to be the empty string if $j < i$.*

Note that a symbol of $prev(S)$ corresponds to a different value in $\mathrm{psuffix}(S, i)$ if it is a parameter pointer that points to a position before i. For example, if $prev(S) = a0a2ab3$, then $psuffix(S, 3) = a0ab3$. It is easily seen that $prefix(S, i, j)$ is the prefix of length $j - i + 1$ of $psuffix(S, i)$.

The following proposition follows directly from the definitions and Proposition 1.

PROPOSITION 2. *If P is a p-string pattern and T is a p-string text, P has a p-match starting at position i of T iff $prev(P) = prefix(T, i, i + |P| - 1)$.*

We also note that the value of the jth symbol of $psuffix(S, i)$ can be computed in constant time from j and the corresponding $(j + i - 1)$st symbol b of $prev(S)$. We let f be this function; the value of f is as follows.

DEFINITION. *For $b \in \Sigma \cup \mathrm{N}$, if b if a nonnegative integer larger than $j - 1$, $f(b, j) = 0$; otherwise $f(b, j) = b$.*

Our strategy is to generalize suffix trees in p-strings based on Proposition 2 and the function f. First, we briefly review suffix trees.

Suppose that Σ is an alphabet and $S = a_1 a_2 \dots a_n$ is a string, where each $a_i \in \Sigma$. For each i, $1 \le i \le n$, the substring $a_i \dots a_n$ is a *suffix* of the input. Without loss of generality, we assume that the last symbol a_n is a unique endmarker; consequently, no suffix is the prefix of another suffix. A suffix tree is a compacted trie (multiway Patricia trie) over the alphabet $\Sigma \cup N$ representing the suffixes of the input string [McC]. A suffix tree is shown in Figure 1. Each arc of the tree is labelled with a nonempty substring of the input, each internal (nonleaf) vertex has degree at least two, and, for each internal vertex, the arcs to its children have labels beginning with distinct symbols. For each leaf, the concatenation of the labels on the path from the

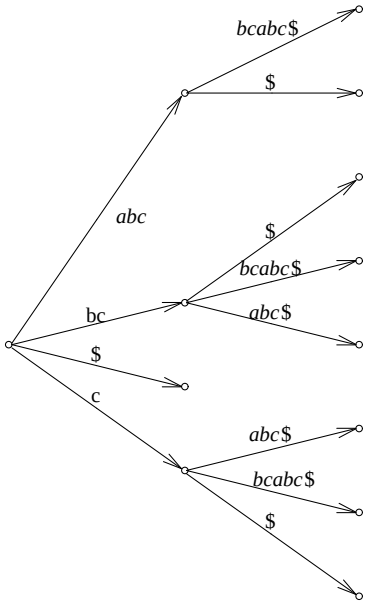


FIG. 1. *A suffix tree for the string abcbcabc* $.

root to the leaf is a distinct suffix of S. Each vertex other than a leaf has at least two children. Since the number of internal vertices is less than the number of leaves, the number of vertices is at most $2n$. Because no suffix is a prefix of another suffix, there is a one-to-one correspondence between the leaves and the suffixes.

The generalization of suffix trees to p-suffixes of a p-string gives us p-suffix trees, defined as follows.

DEFINITION. *If S is a p-string that ends with a unique endmarker in Σ, a* parameterized suffix tree*, or* p-suffix tree*, for S is a compacted trie* (*multiway Patricia trie*) *that stores the p-suffixes of S.*

Each are in a p-suffix tree for S represents a nonempty substring of a p-suffix of S; each internal vertex has degree at least two; and the arcs from an internal vertex to its children have labels beginning with distinct first symbols. For each leaf, the concatenation of the labels on the path from the root to the leaf is a p-suffix of S. Since S ends with a unique endmarker, no p-suffix of S is the prefix of another p-suffix of S, and consequently, each p-suffix of S is the concatenation of the labels on a path from the root to a leaf. Thus, there is a one-to-one correspondence between leaves and p-suffixes of S, and the number of vertices in S is linear in $|S|$. Arcs are oriented from the root toward the leaves, so that arcs leaving a vertex point toward its children.

Example. Let $S = xbyyxbx\$$, where x and y are parameter symbols and b and \$ are constant symbols, so that $prev(S) = 0b014b2\$$. (All parameter pointers are single digits here.) The p-suffixes to be encoded in the tree are $0b014b2\$$, $b010b2\$$, $010b2\$$, $00b2\$$, $0b2\$$, $b0\$$, $0\$$, and \$. Notice that the parameter pointers change to 0 as the preceding part of the string is shortened. The p-suffix tree for S is shown in Figure 2.

DEFINITION. *For each vertex ν, the* pathstring *of ν is the concatenation of the labels on the path from the root to ν, and ν is the* locus *of its pathstring. The length of the pathstring of ν is the* pathlength *of ν.*

In order for the p-suffix tree to be stored in space linear in input length, arc labels

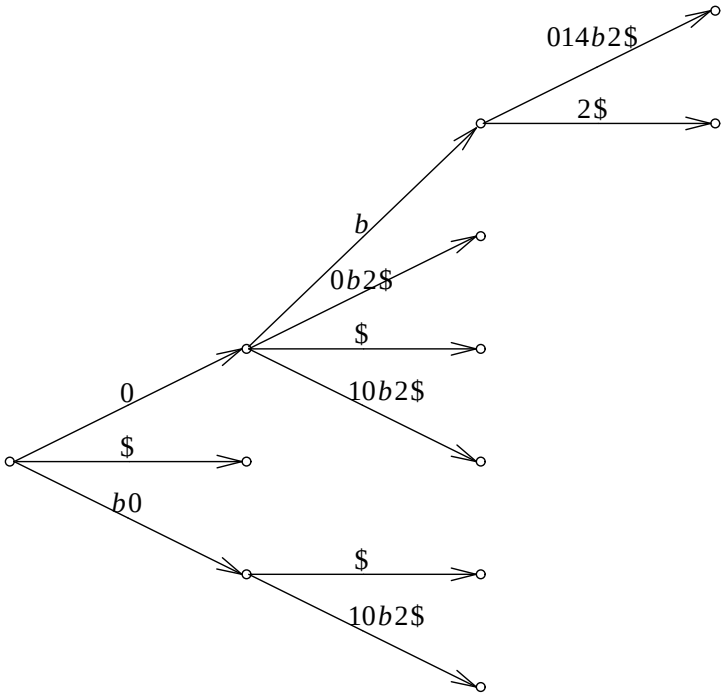


FIG. 2. *A* p-*suffix tree for the* p-*string* $S = xbyyxbx\$$, *where* $\Sigma = \{b, \$\}$ *and* $\Pi = \{x, y\}$.

are calculated dynamically as follows. We assume that $prev(S)$ has been computed and stored in an array. For each vertex ν other than the root, we store the pathlength $plen(\nu)$, an index $firstpos(\nu)$ into the input to specify the starting position of the label of the arc from its parent, and the length $arclen(\nu)$ of this arc. If a label symbol is at pathlength j from the root and corresponds to index k into the input, its value in the label is $f(b, j)$, where b is the kth symbol of $prev(S)$, and f is as defined above; extracting this value takes constant time.

Because up to n symbols of N can be used even when Π is fixed, it might seem that the number of children of each internal vertex is $O(n)$. However, this number is bounded by $|\Sigma| + |\Pi|$, for the following reason. The pathstring of any vertex ν contains parameter pointers representing at most $|\Pi|$ distinct chains of parameters, and the first symbol of an arc to a child must either be a symbol of Σ, a 0 (implying that at most $|\Pi| - 1$ chains appear in the pathstring of ν), or a nonnegative integer pointing to the last element of one of these chains.

For fixed alphabets, linked lists can be used to store the arcs, as far as the theoretical time bounds are concerned; in practice, for large alphabets, hashing would be used, as suggested by McCreight [McC].

Thanks to Proposition 2, searching the p-suffix tree of a text p-string $T\$$ for a pattern p-string P is straightforward: follow the path determined by successive symbols of $prev(P)$ from the root downward in the p-suffix tree for $T\$$ to see if $prev(P)$ is identical to the first part of some p-suffix of T. This search can be accomplished in time $O(|P|)$. The actual matching positions can be calculated from the descendant leaves. Thus, we obtain the following result.

THEOREM 1. *Given p-strings* P *and* T *over fixed alphabets,* $prev(T)$, *and a p-*

suffix tree for T\$, where \$ is a unique endmarker, whether P has a p-match with a substring of T can be determined in time $O(|P|)$ and space $O(|T|)$. All positions of T at which P has a p-match can be found in time $O(|P| + k)$ and space $O(|T|)$, if there are k such positions.

In the next section, we will show that for fixed alphabets, a p-suffix tree can be built in time and space linear in the input length. Consequently, given p-strings P and T, whether P has a p-match with a substring of T can be determined in time $O(|P| + |T|)$ and space $O(|T|)$, and all positions of T at which P has a p-match can be found in time $O(|P| + |T| + k) = O(|P| + |T|)$ and space $O(|T|)$.

3. Building a p-suffix tree. In this section, an algorithm is given for constructing a p-suffix tree. We would like to imitate McCreight's algorithm for building suffix trees as much as possible; however, some basic changes must be made because of the difference between strings and p-strings.

Some useful properties. Strings have the following two trivial properties:

(1) Common Prefix Property. For $a, b \in \Sigma$ and $S, T \in \Sigma^*$, if $aS = bT$, then $S = T$.
(2) Distinct Right Context Property. Suppose $aS = bT$ and $aSc \neq bTd$, where $a, b, c, d \in \Sigma$ and $S, T \in \Sigma^*$. Then $Sc \neq Td$.

These two properties make it possible to augment a suffix tree with pointers called *suffix links*. If an internal vertex has pathstring $a\alpha$, where a is a symbol and α is a string, its suffix link points to an internal vertex with pathstring α; in addition, the suffix link for the root points to the root. The definition of suffix links depends on the two properties in the following way. The existence of an internal vertex with pathstring $a\alpha$ implies there are two distinct strings sharing prefix $a\alpha$. The Common Prefix Property guarantees that stripping off the initial a from the two strings results in strings sharing an initial prefix α, and the Distinct Right Context Property implies that no longer prefix is shared, which in turn guarantees the existence of an internal vertex with pathstring α. Suffix links are useful both for building a suffix tree [McC] and for pattern matching in space proportional to the size of the pattern [CL].

The Common Prefix Property generalizes to p-strings, but unfortunately the Distinct Right Context Property does not.

LEMMA 1 (Common Prefix Property for p-strings). *If $a, b \in \Sigma \cup \Pi$ and S and T are p-strings such that $prev(aS) = prev(bT)$, then $prev(S) = prev(T)$.*

Proof. Observe that $prev(S)$ is different from $prev(aS)$ only in the deletion of the first symbol and in the changing of a parameter pointer pointing to the first position in $prev(aS)$ to a 0, if such a parameter pointer exists, and similarly for $prev(T)$ and $prev(bT)$. By equality of $prev(aS)$ and $prev(bT)$, such parameter pointers, if they exist, must be in the same position in these two p-strings. □

We would like to be able to generalize the Distinct Right Context Property to p-strings as follows: if $prev(aS) = prev(bT)$ and $prev(aSc) \neq prev(bTd)$, then $prev(Sc) \neq prev(Td)$, where $a, b, c, d \in \Sigma \cup \Pi$ and S and T are p-strings. Unfortunately, this is false, because *prev* turns nonnegative integers into 0's as the front end of the string is chopped off. For example, suppose $S = xabxyabz$, with $\Sigma = \{a, b\}$ and $\Pi = \{x, y, z\}$. Then $prev(xabx) = 0ab3$ and $prev(yabz) = 0ab0$, which have a common prefix of $0ab$, but $prev(abx) = ab0 = prev(abz)$, and the distinctness of the right contexts of $0ab$ is lost.

The best we can do is the following restricted form of the Distinct Right Context Property.

LEMMA 2 (Restricted Distinct Right Context Property for p-strings). *Suppose* $prev(aS) = prev(bT)$ *and* $prev(aSc) \neq prev(bTd)$, *where* S *and* T *are p-strings of length* k *and* $a, b, c, d \in \Sigma \cup \Pi$. *If* $prev(Sc) = prev(Td)$, *then the last symbol of one of* $prev(aSc)$, $prev(bTd)$ *is* $k+1$, *while the last symbol of the other is* 0.

Proof. Obviously, $prev(aSc)$ and $prev(bTd)$ differ only at their last symbol. Suppose $prev(Sc) = prev(Td)$. If the common last symbol is a nonzero parameter pointer or is in Σ, then the corresponding symbols in $prev(aSc)$ and $prev(bTd)$ also have a common value ν, implying that $prev(aSc) = prev(aS)\nu = prev(aT)\nu = prev(aTd)$, a contradiction. The only other possibility is that the last symbols of $prev(Sc)$ and $prev(Td)$ are zero, and the last symbols of $prev(aSc)$ and $prev(bTd)$ are parameter pointers in $\{0, k+1\}$. They can't both be zero or both be $k+1$, since $prev(aSc) \neq prev(bTd)$. □

Overview of McCreight's algorithm. McCreight's algorithm for building suffix trees inserts suffixes in stages, where Stage i inserts suffix i (the suffix starting at position i of the input), for $i = 1, 2, \ldots, n$ (from left to right). Define $head_i$ to be the longest prefix of the ith suffix of S that is also a prefix of the jth suffix of S for some $j < i$. The path for suffix i in the tree coincides with an existing path up to $|head_i|$ symbols; in Stage i, the algorithm inserts a new vertex hd_i at that point, if no vertex exists there already, and gives it a child that is a leaf whose pathstring is suffix i.

Suffix links are McCreight's key to turning this idea into an efficient algorithm. A suffix link for a vertex with pathstring $a\alpha$, where a is a symbol and α is a string, points to the vertex with pathstring α (which must exist because of the Common Prefix Property and Distinct Right Context Property); the suffix link for the root points to the root. Suppose Stage $i-1$ found that $head_{i-1}$ was $a\alpha$, where a is a symbol and α a p-string. Then $head_i$ will be at least as long as $|\alpha|$ because of the Common Prefix Property. In Stage i, if the suffix link is already defined for the vertex with pathstring $a\alpha$, there is no need to trace the common prefix α from the root; the processing can jump directly from the vertex with pathstring $a\alpha$ to the vertex with pathstring α via a suffix link. From that point, scanning can continue downward to find the desired location of hd_i.

Unfortunately, the suffix links are constructed dynamically, and the desired suffix link may not actually be defined until after it would be most useful in Stage i. McCreight's algorithm gets around this obstacle as follows. In Stage i, it uses the best suffix link available, namely that of the parent of the desired vertex, and follows a path downward in the tree while *rescanning* part of the input, up to a pathlength of $|\alpha|$. Fortunately, we know what symbols of the input were already scanned, just not where to go in the tree; thus, only the first symbol of each arc needs to be rescanned. If no vertex exists at this point (with pathlength $|\alpha|$), a new vertex hd_i is inserted, the missing suffix link is set to point to hd_i, and a leaf is created to represent suffix i. If a vertex does exist already at this point, the missing suffix link is set to it, and the *scanning* phase continues along the path corresponding to suffix i until the next symbol is not available in the tree; at this point, the algorithm creates an internal vertex hd_i if necessary, and a leaf to represent suffix i. (This explanation skips over some of the details of the algorithm.)

Our algorithm. We would like to generalize McCreight's algorithm to p-suffix trees. Unfortunately, we cannot generally define a suffix link for a vertex with pathstring $prev(aS)$ to point to a vertex with pathstring $prev(S)$, because that vertex may not exist due to the failure of the Distinct Right Context Property for p-strings.

We can, however, define a modified suffix link, called a *contracted suffix link*. For a vertex with pathstring $prev(aS)$, the contracted suffix link points to the best available vertex, namely the one whose pathstring is the *contracted locus* of $prev(S)$, by the following definition.

DEFINITION. *For $\alpha \in (\Sigma \cup \mathrm{N})^*$, the* contracted locus *of α is the vertex whose pathstring is the longest prefix of α of all vertices in the tree.*

The contracted locus of α must exist because the empty string is a prefix of every string, and the root is the locus of the empty string.

The contracted locus of a pathstring may change as vertices are added to the p-suffix tree. Thus many contracted suffix links may need to be reset. But the following algorithm uses lazy evaluation; i.e., a contracted suffix link is reset only when it needs to be evaluated.

DEFINITION. *For a p-string S and $i \geq 1$, define $head_i(S)$ as the longest prefix of $psuffix(S, i)$ that is also a prefix of $psuffix(S, j)$ for some $j < i$. Define $head_0$ to be the empty string.*

LEMMA 3. *If $head_{i-1}(S) = prefix(S, i-1, s)$, where $i - 1 \leq s$, then $head_i(S) = prefix(S, i, t)$ for some $t \geq s$.*

Proof. By definition of $head_{i-1}$, there is some $j < i - 1$ such that $head_{i-1}(S)$ is a prefix of $psuffix(S, j)$ as well as of $psuffix(S, i-1)$. Since $|head_{i-1}(S)| = s - i + 2$, the first $s - i + 1$ symbols of $psuffix(S, j+1)$ and $psuffix(S, i)$ must be the same by the Common Prefix Property. The result follows by definition of *head*. □

The construction of p-suffix trees follows the organization of McCreight's original algorithm [McC] as much as possible; modifications are needed to allow for updating out-of-date contracted suffix links and the extra searching resulting from out-of-date contracted suffix links. The values of *plen*, *firstpos*, and *arclen* are stored for the vertices as described earlier. In addition, for each vertex ν, the contracted suffix link $CSL(\nu)$ is stored, and if ν is not the root, a pointer to its parent $parent(\nu)$ is stored.

Let S be the p-string for which the p-suffix tree is to be constructed; we assume that S ends in a unique endmarker in Σ, and $P = prev(S)$ has already been constructed in linear time and space as described in the previous section. The main procedure of the algorithm, called *lazy*, is given in Figure 3; additional procedures, *prescan*, *rescan*, and *scan*, called by *lazy* are described in the text. The ith iteration of the main loop of *lazy* will be referred to as Stage i and inserts the ith p-suffix into the tree.

The tree is initialized to a root, with $CSL(root) = root$ and $oldhd = root$. We will prove the algorithm correct inductively by means of the following properties, which we will show must hold for Stage i, $i \geq 1$.

P1: At the beginning of stage i, $CSL(\nu)$ has been set for the root and for every internal vertex ν except possibly for *oldhd*, which is the locus of $head_{i-1}$; $CSL(root)$ points to *root*, and for a vertex ν other than the root, if $CSL(\nu)$ is defined and the pathstring of ν is $prefix(S, j, k)$, where $j \leq k$, $CSL(\nu)$ points to a vertex whose pathstring is $prefix(S, j+1, t)$ for some t, $j \leq t \leq k$.

P2: At the end of Stage i, the tree is a compacted trie for the first i p-suffixes.

The goals in Stage i are to set $CSL(oldhd)$, to find or create hd, the locus of $head_i$, and to create a new leaf as the locus for $psuffix(S, i)$. Along the way, the contracted suffix link for $parent(oldhd)$ may be updated. In the following discussion, we assume that P1 and P2 held up to the start of Stage i.

If *oldhd* is the root, $CSL(oldhd)$ is already set to the root, and is up-to-date; the algorithm proceeds to call *scan*, described below.

```
/*index(i,len) translates a tree pathlength len into an index
into prev(S) for Stage i */
#define index(i,len) i-1+len
#define FP(i,j)          f(P[i],j)

lazy() {
   int i,short;
   VERTEX c,d,hd,oldhd,start;
   create a tree consisting of a root;
   oldhd=root;
   CSL(oldhd)=root;
   for (i=1;i<=n;++i) { /* Stage i */
      if (oldhd == root) hd=scan(root,child(root,S(0,i)),0,i,i);
      else {
         if (CSL(oldhd) is defined)
           start = CSL(oldhd);
         else
           start = CSL(parent(oldhd)) = prescan(parent(oldhd),i);
         pgoal=plen(oldhd)-1;   /* pgoal = |prefix(S,i,s)| */
         c=rescan(start,pgoal,i); /*contracted locus of prefix(S,i,s) */
         short = pgoal-plen(c);
         /*compare next transformed symbol of prev(S) to corresponding
                symbol on the appropriate arc out of c */
         d=child(c,f(plen(c),index(i,plen(c)+1)));
         if ((d is defined) and (short>0) and
            (FP(index(i,pgoal+1),pgoal)!= FP(firstpos(d)+short,pgoal) {
            create a new vertex hd between c and d
              with arclen(hd) = short and firstpos(hd)=i-1+pgoal;
            CSL(oldhd) = hd;
            }
         else  {
              CSL(oldhd) = c;
              hd = scan(c,d,short,index(i,pgoal+1),i);
            }
         }
      }
      add a new leaf lf as a child of hd, with firstpos(lf)=i+plen(hd),
          arclen(hd)=n-i-plen(hd)-1; /*  locus of psuffix(i) */
      oldhd=hd;
   }
}
```

FIG. 3. *The main procedure for the algorithm for constructing p-suffix trees.*

So suppose *oldhd* is not the root. For $s = plen(oldhd) + i - 2$, $head_{i-1} = prefix(S, i-1, s)$. $CSL(oldhd)$ must be set to the contracted locus of $prefix(S, i, s)$. Fortunately, $prefix(S, i, s)$ is guaranteed to be a prefix of some pathstring already existing in the tree. This follows from the definition of $head_{i-1}$ and the Common Prefix Property.

Initially, *lazy* looks for a vertex *start* that is an ancestor of the contracted locus of $prefix(S, i, s)$. If $CSL(oldhd)$ is already defined (although possibly out of date), by property P1, $CSL(oldhd)$ can be used as *start*. Otherwise *lazy* begins by updating $CSL(parent(oldhd))$ and sets *start* to its updated value. This updated value is found by calling $prescan(parent(oldhd), i)$ and will be the contracted locus of $prefix(S, i, r)$ for some $r < s$.

If $parent(oldhd)$ is the root, $prescan(parent(oldhd), i)$ returns the root. Otherwise, it follows the path of $prefix(S, i, s)$ downward in the tree from the vertex pointed to by $CSL(parent(oldhd))$. It needs to check only the first symbol of each arc label since $prefix(S, i, r)$ is a prefix of $prefix(S, i, s)$, which we showed to be a prefix of some pathstring in the tree. It finds the contracted locus by not exceeding the desired pathlength, $plen(parent(oldhd)) - 1$.

Once *start* has been set, *lazy* finds the current contracted locus c of $prefix(S, i, s)$ by calling a function $rescan(start, pgoal, i)$, where $pgoal = plen(oldhd) - 1$. Now, *rescan* scans downward from *start* following the path of $prefix(S, i, s)$. Like *prescan*, *rescan* checks only the first symbol of each arc label because $prefix(S, i, s)$ is known to be a prefix of some pathstring in the tree. The contracted locus of $prefix(S, i, s)$ is found by not exceeding the desired pathlength, *pgoal*; the contracted locus is *short* symbols above where the locus of $prefix(S, i, s)$ would be (if it existed), for some $short \geq 0$.

While c is currently the contracted locus of $prefix(S, i, s)$, it may no longer be so at the end of the stage if a new vertex is created as the locus of $head_i$. By Lemma 3, for some $t \geq s$, $head_i = prefix(S, i, t)$, but the value of t is not yet known. The only case in which c will not be the contracted locus of $prefix(S, i, s)$ at the end of the stage is when $short > 0$ and $t = s$. If $short > 0$, whether $t = s$ is determined by checking the $(short + 1)$st transformed symbol on the arc from c to the appropriate child d.

Thus the algorithm proceeds as follows. If $short > 0$ and $t = s$, a new vertex is created as the locus hd of $head_i$ and $CSL(oldhd)$ is made to point to it. Otherwise, the algorithm sets $CSL(oldhd)$ to c and sets hd to the vertex returned by $scan(c, d, short, i + pgoal, i)$.

$Scan(c, d, short, j, i)$ begins just after the $(short)$th symbol on the arc into d (or at c if $short = 0$) and the jth input symbol and scans downward in the tree along the path determined by $psuffix(S, i)$ until the next transformed input symbol is not available in the tree. At this point, *scan* creates a new vertex as the locus of $head_i$, if none exists already, and returns it.

Finally, a new arc is added from the locus hd of $head_i = prefix(S, i, t)$ to a new leaf, which is the locus of $psuffix(S, i)$.

Property P1 holds initially when the tree is initialized to just the root. In Stage i, the algorithm sets $CSL(oldhd)$ to the contracted locus for $prefix(S, i, s)$, implying P1 holds for $oldhd$, and care is taken to ensure that P1 still holds for $parent(oldhd)$ if its contracted suffix link was reset. No other contracted suffix links were changed. Hence, if P1 held at the beginning of this stage, it still holds at the end of the stage.

Property P2 holds at the beginning when the tree is initialized to just the root, and for $i > 1$, by the induction hypothesis, property P2 holds at the end of Stage $i - 1$. Either the locus of $head_i$ is created in Stage i by *lazy* because $short > 0$ and $t = s$, or it is created by *scan*. In either case, it is made the child of the old contracted locus of $head_i$ in the tree. Therefore, property P2 holds at the end of Stage i.

By induction, properties P1 and P2 hold for all stages. For $i = n$, P2 implies that the tree is a compacted trie for all the p-suffixes of S. Thus, the above algorithm constructs a p-suffix tree for S.

Analysis.

THEOREM 2. *Let Σ and Π be fixed finite disjoint alphabets. Given a p-string $S \in (\Sigma \cup \Pi)^*$ ending in a unique endmarker in Σ, a p-suffix tree can be constructed for S in time $O(n)$ and space $O(n)$, where n is the length of S.*

Proof. Since correctness of the algorithm was shown above and linearity of space was shown when p-suffix trees were defined in section 2, it only remains to analyze the running time. The proof is more complicated than that for McCreight's algorithm because of the parameter pointers and the use of contracted loci. In McCreight's algorithm, of the input symbols rescanned in a single stage, only the last can be rescanned again later, implying that the time for rescanning is linear in the length of the string. In our case, the failure of the Distinct Right Context Property and the resulting use of contracted loci mean that from one stage to the next, rescanning can back up in the input and rescan again a sequence of symbols already rescanned, but at most a number proportional to $|\Pi|$ in any stage, for a total of $O(|\Pi|n)$ rescanning steps. Prescanning, not needed in McCreight's algorithm, also can recheck symbols already checked previously, but again at most a number proportional to $|\Pi|n$. The result will follow from our assumption that $|\Pi|$ is $O(1)$.

First we observe that *scan* uses $O(n)$ time over all stages, because in each stage, *scan* is called at most once and scans at most one symbol scanned in an earlier stage, and $|\Sigma|$ and $|\Pi|$ are $O(1)$.

Next, we analyze the work required for prescanning.

Call a contracted suffix link *good* if it is for the root or if it is from ν to $\overline{\nu}$, where the pathlength of $\overline{\nu}$ is one less than the pathlength of ν. Otherwise, it is *bad*. By the Restricted Distinct Right Context Property, when a vertex is first given a bad suffix link, it has exactly two arcs, one whose label begins with 0 and one whose label begins with the pathlength of the vertex.

Let $BAD(y)$ be the set of vertices ν that have had contracted suffix links pointing to proper ancestors of y at the start of Stage i and whose contracted suffix links are reset to point to y or a descendant of y after y is created. Every prescanning step that checks the first symbol on an outarc of y is due to a distinct member of $BAD(y)$.

At the start of Stage i, every vertex in $BAD(y)$ still has exactly two outarcs, one whose label begins with 0 and one whose label begins with the pathlength of the vertex, since otherwise the vertex would have been given a good contracted suffix link already. We claim that all vertices of $BAD(y)$ lie in a single path in the tree at the start of Stage i and that the path corresponding to $BAD(y)$ goes along the arcs whose labels begin with a parameter pointer 0.

For suppose that vertices of $BAD(y)$ include w_b and w_c, where neither is an ancestor of the other. Let z be their lowest common ancestor. Since any pathstring can have at most one parameter pointer to the first symbol, by the definition of *prev*, z has two arcs whose labels begin with symbols we will call b and c, respectively, where b and c are not parameter pointers to the first symbols in the pathstrings and $b \neq c$. Without loss of generality, suppose the arc whose label begins with c was created second. By the Restricted Distinct Right Context Property, in the stage after the second arc was created, z received a good contracted suffix link to a vertex u, with arcs whose labels begin with b and c, respectively. Since all internal vertices descended from the arc of z whose label begins with c get their contracted suffix links created after u is created (by construction), w_c has a bad contracted suffix link pointing to u or to a descendant of u through the arc whose label begins with c. By the definition of $BAD(y)$, the contracted suffix link of w_c points strictly above y at the start of Stage i but to y or below y after y is created, and consequently y is a descendant of u through the arc whose label begins with c. But then the $(|u|+1)$st symbol of the pathstring of y is c, whereas by the definition of w_b, it must be b, contradicting the membership of w_b in $BAD(y)$. Therefore, either w_b or w_c must be an ancestor of the other.

Moreover, at each vertex ν (other than the one farthest from the root) in $BAD(y)$, the path follows the outarc whose label begins with 0. The path cannot follow the arc whose label begins with a parameter pointer to the first symbol in the pathstring, because each vertex of $BAD(y)$ has an outarc whose label begins with a parameter pointer to the first symbol in the pathstring, and a pathstring can have at most one such parameter pointer by definition of *prev*.

Since the number of 0's in a pathstring is at most $|\Pi|$, $BAD(y)$ contains at most $|\Pi|$ vertices. Therefore, vertices created before y account for at most $|\Pi|$ symbols prescanned because of y. The only other prescanning steps involving y are those in which prescanning begins at y for vertices whose bad contracted suffix links point directly at y; there are at most n such steps.

There are at most n prescanning steps that can be allocated to the first symbol prescanned in each stage. Over all vertices y, there are at most $|\Pi|n$ additional prescanning steps. Since $|\Sigma|$ and $|\Pi|$ are $O(1)$, each step takes $O(1)$ time, and the total time for prescanning is $O(n)$.

Finally, we analyze the time required for rescanning.

An argument similar to the prescanning argument shows that the number of symbols rescanned in resetting an out-of-date *CSL*(*oldhd*) is $O(|\Pi|n) = O(n)$ over all stages.

Next, we consider rescanning for stages in which *CSL*(*oldhd*) is initially undefined. We will show that, in successive stages, rescanning can back up in the input and rescan sections of input already rescanned, but it cannot back up past symbols already rescanned whose transformed value is not 0. More precisely, suppose that the kth symbol of $prev(S)$ is rescanned in Stage i after having been previously rescanned, $k > i$ (the kth symbol is not the first symbol in the label of an outarc from the root), and the transformed value in this rescanning is not 0. We will show that this symbol must be the first symbol rescanned in Stage i.

The transformed value is the first symbol in the label of an outarc of the locus of $prefix(S, i, k-1)$ and is either a symbol in Σ or an integer between 1 and $k-i$. Also, $k \leq s$, where *oldhd* is the locus of $prefix(S, i-1, s)$, or the kth symbol of $prev(S)$ would not be rescanned. Let j be the number of the last stage in which this symbol was rescanned or scanned. In Stage j, the transformed value of the kth symbol was the same, because of the definition of f and the fact that this symbol was deeper in the tree in Stage j than in Stage i. In Stage j, the kth symbol of $prev(S)$ corresponds to the first symbol on an arc of an internal vertex. Therefore, for some q, $prefix(S, j, k-1) = prefix(S, q, q+k-j-1)$ but $prefix(S, j, k) \neq prefix(S, q, q+k-j)$, and the last symbol of $prefix(S, j, k)$ is either a symbol in Σ or an integer between 1 and $k-i < k-j$. Therefore, by the Common Prefix Property and Restricted Distinct Right Context Property, the locus of $prefix(S, j+1, k-1) = prefix(S, q+1, q+k-j-1)$ exists at the end of Stage $j+1$. Moreover, this argument can be applied inductively to show that the locus of $prefix(S, i-1, k-1)$ exists at the end of Stage $i-1$. Consequently, $k-1 \leq r < s$, where $parent(oldhd)$ is the locus of $prefix(S, i-1, r)$. But the contracted locus of $prefix(S, i, r)$ cannot be any closer to the root than the locus of $prefix(S, i, k-1)$. We conclude that the kth symbol of $prev(S)$ must be the first symbol rescanned in Stage i.

We have shown that other than the first symbol rescanned in each stage, only symbols transformed into 0's can have been rescanned previously. Since the number of 0's in any pathstring is at most $|\Pi|$, each stage rescans at most $|\Pi| + 1$ symbols already rescanned previously. Over all stages, there are at most n rescannings of

symbols for the first time. Since $|\Pi|$ and $|\Sigma|$ are $O(1)$, $O(|\Pi|n) = O(n)$ symbols are rescanned and the time for rescanning each symbol is $O(1)$. Thus, the total time spent on rescanning over all stages is $O(n)$.

From above, scanning, prescanning, and rescanning use $O(n)$ time over all stages. Since the time used by *lazy* outside of these procedures is also $O(n)$, the result follows. □

4. An algorithm for finding all maximal p-matches over a threshold length. This section defines maximal p-matches and gives algorithms for finding all maximal p-matches over threshold length in a p-string.

DEFINITION. *For a p-string S, define S_i to be its ith symbol, for $1 \le i \le |S|$, and $S_{i,j} = S_i \dots S_j$, for $1 \le i \le j \le |S|$.*

Let S be a p-string of length n. If $S_{i,i+k}$ and $S_{j,j+k}$ are a p-match, where $1 \le k < n$ and $1 \le i,\ j \le n-k$, we denote it by $(i, j, k+1)$ or (equivalently) $(j, i, k+1)$, i.e., by the two starting positions and the length of the p-match. Define S_{n+1} to be an endmarker \$ that does not occur in S, and S_0 to be a beginning marker that also does not occur in S, with $S_0 \neq \$$.

DEFINITION. *Suppose $S_{i,i+k}$ and $S_{j,j+k}$ are a p-match, where $1 \le i \le i+k \le n$ and $1 \le j \le j+k \le n$. We say this p-match is* left-extensible *if $S_{i-1,i+k}$ and $S_{j-1,j+k}$ are a p-match, and* right-extensible *if $S_{i,i+k+1}$ and $S_{j,j+k+1}$ are a p-match. If it is neither left-extensible nor right-extensible and is not the trivial p-match $(1, 1, n)$, we say it is a* maximal *p-match.*

LEMMA 4. *If (i, j, k) is a maximal p-match, then this p-match cannot be extended in any amount in either or both directions; i.e., there are no $r, s \ge 0$ with at least one of r, s nonzero such that $(i-r, j-r, r+k+s)$ is a p-match.*

Proof. If two p-strings p-match, and they are truncated on the right (left) by the same amount, the resulting p-strings will still p-match. Therefore, if there is a p-match that is more than one symbol longer in either direction, there will also be a p-match that is exactly one symbol longer, contradicting the nonextensibility of the p-match. □

Maximal p-matches for p-strings include as a subcase maximal matches for strings. It was shown in [Bak2] that the maximal match relation is not an equivalence relation, because it is not transitive. For example, consider the string *adbdadb*. The triple (2, 4, 1) represents the maximal match between the first two *d*'s. Similarly, the triple (4, 6, 1) represents the maximal match between the last two *d*'s. However, the first and last *d*'s are not a maximal match; they are part of the longer maximal match (1, 5, 3). Thus, maximal p-matches are also not an equivalence relation, and an algorithm to report all maximal p-matches over a threshold length must report pairs of p-substrings rather than equivalence classes of p-strings.

In [Bak2], a suffix-tree-based algorithm was given to find all maximal matching substrings over a threshold length in a string. We would like to generalize that algorithm to p-suffix trees and p-strings. The algorithm for strings was based on two facts: each pathstring in a suffix tree represents one or more matches that are not right-extensible, and whether the p-matches are left-extensible can be determined by checking the symbol to the left of the matching substrings.

For p-suffix trees, it is also true that each pathstring represents one or more p-matches that are not right-extensible. However, checking whether the p-matches are left-extensible is more complicated.

Suppose $S_{i,i+k}$ and $S_{j,j+k}$ are a p-match. If S_{i-1} and S_{j-1} are both in Π, the first symbols of $prev(S_{i-1,i+k})$ and $prev(S_{j-1,j+k})$ will both be 0. Nevertheless, it may

happen that $S_{i-1,i+k}$ and $S_{j-1,j+k}$ are not a p-match. The reason is that for some $r \leq k$, the rth symbols of $prev(S_{i,i+k})$ and $prev(S_{j,j+k})$ may both be 0, but S_{i+r-1} may be the next occurrence of S_{i-1}, while S_{j+r-1} may be the first occurrence of some parameter other than S_{j-1}, so these symbols cannot correspond under renaming in $S_{i-1,i+k}$ and $S_{j,j+k}$. For example, consider $S_{i-1,i+k} = xabcx$ and $S_{j-1,j+k} = yabcz$, where x, y, and z are parameters. Then $prev(abcx) = abc0 = prev(abcz)$, but $prev(xabcx) = 0abc3$ while $prev(yabcz) = 0abc0$. This is the failure of the Distinct Right Context Property in a different guise.

It would be convenient to have a way to determine left-extensibility just by checking the positions to the left. Our solution is to construct another string, $A = (prev(S^r))^r$, where the superscript r represents reversal; in A, the parameters are turned into forward references rather than back references as before. Note that A can be constructed in time and space linear in $|S|$ by scanning $prev(S)$, replacing each parameter by a 0, and then replacing each such 0 pointed to by a parameter pointer by the value of the parameter pointer. Let $A_0 = S_0$ (the unique beginning-marker). The following proposition shows that by applying a transform function to each left context, we need check only the left context positions for equality.

LEMMA 5. *Let $i, j, k > 0$ and $i \neq j$. A p-match (i, j, k) is left-extensible iff $f(A_{i-1}, k+1) = f(A_{j-1}, k+1)$.*

Proof. The proof is trivial if at least one of the symbols A_{i-1} and A_{j-1} is in Σ. So, consider the case where both are parameters.

If (i, j, k) is left-extensible, $(i-1, j-1, k+1)$ is a p-match and S_{i-1} occurs in $S_{i,i+k-1}$ iff S_{j-1} occurs in the corresponding positions of $S_{j,j+k-1}$. If these symbols do occur, then

$$f(A_{i-1}, k+1) = f(A_{j-1}, k+1) > 0,$$

while if they don't occur, then

$$f(A_{i-1}, k+1) = f(A_{j-1}, k+1) = 0.$$

Now, suppose (i, j, k) is a p-match and $f(A_{i-1}, k+1) = f(A_{j-1}, k+1) = r$. If $r = 0$, then the initial symbols are both parameter symbols that don't occur in the rest of the p-strings, and the p-match is left-extensible. If $r > 0$, then the initial symbols are the same as the parameter r symbols to the right, and the one to one correspondence between parameter symbols for (i, j, k) also implies that $(i-1, j-1, k+1)$ is a p-match and (i, j, k) is left-extensible. □

The algorithm for finding all maximal p-matches over a threshold length will be based on p-suffix trees augmented by lists of the following forms.

DEFINITION. *A* plist *is a list of integers, and a* clist *is a list of plists.*

An integer i in a plist will represent the ith p-suffix. Each plist will be constructed so that all of its member elements correspond to p-suffixes with the same transformed left context. The intent is to construct a clist for each vertex ν to represent the descendant leaves of ν, sorted by transformed left context in A.

For strings, the algorithm recurses over a suffix tree as follows [Bak2]. For each leaf L, it creates a clist containing a single plist with one index corresponding to the suffix represented by the leaf. At each internal vertex, after constructing the clists for the children, the algorithm sorts the information represented by the clists of the children into a new clist. The sorting is accomplished by processing the children from left to right and merging their information into a new clist. At the same time, any longest p-matches that are found are reported.

The following example illustrates the operation of the string algorithm at a vertex ν. Suppose we represent a plist with positions $p_1, p_2, \ldots, p_k$ and left context σ by $\sigma : p_1, p_2, \ldots, p_k$. If the first child of ν has a clist containing plists a:35,72,46, b:43,7, and c:25, the second child has a clist consisting of plists a:66,2 and c:56, and the third child has a clist consisting of plists b:64,31 and c:82,13,59, where a, b, $c \in \Sigma$, then the clist constructed for the parent, ν, will be a:35,72,46,66,2, b:43,7,64,31, and c:25,56,82,13,59.

For strings, this algorithmic structure is adequate [Bak2]. For p-strings, an extra step must be performed, because the transformed left context of a plist may change from nonzero to zero when it is transformed with respect to a smaller number. For example, if the transformed left context is 35 when evaluated with respect to the pathlength 38 of a child, and the parent has pathlength 32, then the transformed left context will be 0 when transformed with respect to the pathlength of the parent.

Thus, a clist that is sorted by left context for a child may contain more than one plist with left context 0 when the left contexts are evaluated with respect to the parent. Thus, after a clist is constructed for a child c of a vertex ν, the clist is scanned for any plists corresponding to left contexts of 0 when transformed with respect to the pathlength of ν, and such plists are merged into a single plist.

Figure 4 gives the three procedures needed to perform this algorithm for a threshold t: *pdup*, *concatz*, and *pcombine*. The main procedure is *pdup*, which recurses over the p-suffix tree. For each internal vertex, *pdup* calls *pcombine*, which sorts and combines the plists produced for the children, and applies *concatz* to handle the special case where a nonzero value of a transformed left context changes to a zero value. The concatenations of plists, which are not described explicitly, are done via pointers rather than by copying; by maintaining pointers to the beginning and end of each plist, each concatenation is done in $O(1)$ time. For conciseness, the following definitions are assumed in the pseudocode.

DEFINITION. *For a leaf L, the starting position of the p-suffix corresponding to L is $start(L) = firstpos(L) + arclen(L) - plen(L)$. For an integer i, define $LCA(i)$ to be A_{i-1}. For a plist of pl, define $LCA(pl)$ to be A_{i-1}, where the first element of pl is i.*

Thus, for a vertex ν, $f(LCA(pl), plen(\nu)+1)$ is the transformed left context used for pl while processing vertex ν.

THEOREM 3. *Given a p-string S of length n over fixed alphabets Σ and Π and a positive integer t, all p-matches of length at least t in S can be found in time $O(n+r)$ and space $O(n)$, where r is the number of matches reported.*

Proof. Construct the p-suffix tree T for $S\$$, where \$ is an endmarker that doesn't occur in S, using linear time by Theorem 2. Then, call $pdup(root(T), t)$, where *pdup* is given in Figure 4.

We claim that the algorithm correctly merges the clists of the children at each vertex ν, transforming the left context values as required by the pathlength of ν, and reporting exactly the longest p-matches over threshold length. A formal proof would be by induction on the depth of ν; details are left to the interested reader.

Now, we need to analyze the time and space bounds for the algorithm. The time spent on linear searches of clists and plists in *pcombine* and *concatz* is dominated by the number of steps spent on cross-products. Two cross-products of clists are performed in *pcombine*. We partition the work into *same-work*, namely, the steps required when the transformed left contexts are the same, and *different-work*, namely, the steps required when the transformed left contexts are different. We bound the amount of same-work and different-work as follows.

```
/*f is defined in Section 2 */
clist pcombine(clist cl1, clist cl2, int len, int t) {
   plist pl1, pl2;
   clist outputlist;
   if (len < t) return (NULL);
   for each plist pl1 of cl1 and each plist pl2 of cl2
      if (f(LCA(pl1),len+1) ≠ f(LCA(pl2),len+1))
         for every p1 in pl1 and every p2 in pl2
           report a maximal match (p1, p2, len);
   /*construct outputlist */
   for each plist pl1 of cl1 and each plist pl2 of cl2
      if (f(LCA(pl1), len+1) == f(LCA(pl2),len+1)) {
         include in outputlist the plist obtained by concatenating
           pl2 to pl1;
        mark pl1 and pl2 as used;
      }
   for each plist pl1 of cl1 that is not marked used
      include pl1 in outputlist;
   for each plist pl2 of cl2 that is not marked used
      include pl2 in outputlist;
   remove marks from all plists in outputlist;
   return outputlist;
}
clist concatz(clist cl, int len) {
   scan cl and concatenate all plists pl for which f(LCA(pl), len+1)=0
      into one plist while leaving others unchanged
}

clist pdup(vertex v, int t) {
   clist cl;
   if (v is a leaf) return a clist containing one plist
      consisting of start(ν);
   cl = NULL;
   for each child s of v
      cl = pcombine(cl, concatz(pdup(s,t),plen(v)),plen(v),t);
   return cl;
}
```

FIG. 4. *The algorithm pdup for reporting all maximal p-matches of length at least t.*

By examining the second cross-product of clists, where plists are merged, it is easy to see that the same-work done at vertex ν is linear in the number of new links created in combining plists, and consequently, the amount of same-work performed over all vertices is linear in the sum of the lengths of the plists for the root, and is consequently linear in n.

At each step in the first cross-product with distinct transformed left contexts, a maximal p-match is reported for each pair of positions in the cross-product of the plists. Therefore, the amount of time spent over all vertices on different-work in the two cross-products of clists is linear in r. □

5. Implementation and experiments. This section describes an application of the p-suffix tree to finding parameterized duplication in C code. The algorithms *lazy* and *pdup* have been implemented in a C program running under UNIX. The sets Σ and Π are disjoint sets of integers, varying with the input. For each line of C code, the lexical analyzer turns tokens such as identifiers and constants (but not keywords or operators) into integer parameter symbols, generates a new version of

TABLE 1
Data for lazy *on three subsystems.*

Number of lines	Number of symbols	$\|\Sigma\|$	$\|\Pi\|$	Number of symbols prescanned	Number of symbols rescanned	Time for *lazy* in seconds
35233	112146	2406	6761	60730	27371	4.97
101874	320947	3992	11718	176690	74149	15.06
158579	517415	8808	24168	271069	121202	25.91

the line in which each such token is replaced by a "P," and obtains an integer in Σ corresponding to the transformed line. Thus, each line of input corresponds to one symbol of Σ and zero or more symbols of Π (in the order in which the corresponding tokens occur in the line). Since Σ and Π can be very large, accessing the children of each vertex is accomplished by hashing rather than by linked lists in *lazy*. How to do this was suggested by McCreight [McC]. However, since *pdup* requires being able to scan successive children of each vertex, which cannot easily be done with the hashing method, the outarcs of each vertex are converted into linked lists for running *pdup*; the transformation is done in time linear in input length.

The program was applied to code from three different subsystems of a production system. The experiments were run on one processor of an SGI machine with eight 33 MHz R3000 processors, data and instruction caches of 64 Kbytes, a secondary data cache size of 256 Kbytes, and main memory size of 256 Mbytes.

Table 1 gives the data related to *lazy*. The first four columns give the data for the input subsystems: the number of lines (including comments and white space), the length of the resulting p-string, the size of Σ, and the size of Π. The last three columns give the running time for *lazy*, (not including the lexical analysis), the number of symbols prescanned, and the number of symbols rescanned.

The proof of Theorem 2 suggests that, in the worst case, the number of steps executed for prescanning and rescanning might be proportional to $|\Pi|$ times the length of the input. Since $|\Pi|$ is not constant in our application, and could be in principle proportional to n, the running time could be quadratic rather than linear in n. Table 1 shows that this blowup was not observed in the experiments: in each case, the number of symbols prescanned and the number of symbols rescanned were less than the length of the input. The question of why this blowup was not observed is partially answered by looking at the maximum number of prescannings and rescannings in any stage. For the largest subsystem, the maximum number of prescannings in any stage was 6, and for the other two, it was 3; for the largest subsystem, the maximum number of rescannings in any stage was 99, for the second largest, it was 33, and for the smallest, it was 16. Either the bound of $|\Pi|n$ steps is not tight, or the input from code does not generate worst-case behavior. In any case, we conclude that *lazy* runs fast even with large alphabets and large input.

Table 2 gives data regarding running times of *pdup* and running times of the whole program (including lexical analysis and *lazy*) on the three subsystems, with a threshold length of 75, which is a value which would be reasonable for the application. The times given for *pdup* include the time for transforming the arc representation from hashing into linked lists and the time for running *pdup*. Note that with a threshold of 75, the number of p-matches reported is much less than input length; hence the running time of *pdup* is dominated by the part of the algorithm linear in input length. With a threshold of 10 for the largest subsystem, with p-string length of 517415, *pdup*

TABLE 2
Running times for pdup *and for the whole program.*

Threshold length	Number of lines	Number of symbols	Number of matches	*pdup* time in seconds	Total time in seconds
75	35233	112146	512	4.28	27.48
75	101874	320947	3837	15.14	69.64
75	158579	517415	16088	26.06	111.33

reports 5017926 maximal p-matches and takes 467 seconds, and the whole program takes 555 seconds.

6. Discussion. The theorems in this paper were stated in terms of fixed alphabets, but in the application, as described in the last section, the alphabets are not fixed. It is interesting to examine further the worst-case time bounds in the case of variable alphabets.

The construction of A from $prev(S)$ takes time linear in $|S|$ even for variable alphabets, and in *pdup*, arcs are only accessed sequentially. Thus, for variable alphabets, once the p-suffix tree is constructed for a p-string S, the time to report all maximal p-matches over a threshold length in S is $O(n + r)$, where n is the input length and r is the number of p-matches reported.

On the other hand, as mentioned in the previous section, the bounds proved for constructing a p-suffix tree do depend on alphabet size: the proof of Theorem 2 shows that the number of symbols prescanned and rescanned when running *lazy* on input of length n is bounded by $O(n|\Pi|)$. If arcs are stored in a balanced tree scheme with $O(\log(|\Sigma| + |\Pi|))$ access time, then *lazy* constructs a p-suffix tree in worst-case time $O(n(|\Pi| \log(|\Sigma| + |\Pi|)))$. As discussed in the last section, these bounds may not be tight.

The author is continuing work on p-suffix trees, some of which will be described in [Bak3]. As a step in improving the worst-case bounds for variable alphabets, [Bak3] will show that a p-suffix tree for a p-string S of length n can be constructed in $O(n \log n)$ time and $O(n)$ space if Σ and Π can vary. However, that algorithm is not practical, since it relies on complicated auxiliary data structures.

An issue regarding the choice of problem statement is that reporting all maximal p-matches over a threshold length does not necessarily result in a natural analysis of certain kinds of duplication. For example, a structure of $(abc)^4$ will be reported as one p-match of length 3, one of length 6, and one of length 9 (with the matching p-strings overlapping). It could be that some other set of definitions would enable a more natural analysis of the structure of duplication.

A related issue regarding the choice of definitions is the issue of how the amount of output relates to the amount of information it conveys. Tufte has stressed in his beautiful book [Tu] on the visual display of data that it is important to maximize the data-ink ratio. The same principle should apply to how much output is reported by algorithms. In the case of duplication, if there are c copies of the same p-string, then a natural way to report them would be by listing c starting positions and the p-string; however, because the p-match relationship is not an equivalence relation, they could in general be part of $c(c-1)/2$ distinct maximal p-matches, and under the maximal p-match definition, these must be reported separately. It is not obvious what definitions would enable reporting a minimal amount of information in each case, in a way easily understandable to the user of the program.

Acknowledgments. The author would like to thank Raffaele Giancarlo for helpful discussions relating to this work, and William Chang for providing an implementation of McCreight's algorithm that was modified to implement `lazy`.

REFERENCES

[Aho] A. V. Aho, *Algorithms for finding patterns in strings*, Handbook of Theoretical Computer Science, J. Van Leeuwen, ed., Elsevier Science Publishers, Amsterdam, 1990, pp. 255–300.

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[Bak1] B. S. Baker, *A program for identifying duplicated code*, Comput. Sci. Stat., 24 (1992), pp. 49–57.

[Bak2] B. S. Baker, *On finding duplication in strings and software*, Algorithmica, to appear.

[Bak3] B. S. Baker, *Parameterized pattern matching: Algorithms and applications*, in Proc. of 25th Annual ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 71–80.

[CL] W. I. Chang and E. L. Lawler, *Sublinear approximate string matching and biological applications*, Algorithmica, 12 (1995), pp. 327–344.

[CH] K. W. Church and J. I. Helfman, *Dotplot: A program for exploring self-similarity in millions of lines of text and code*, J. Comput. Graph. Statist., 2 (1993), pp. 153–174.

[GG] Z. Galil and R. Giancarlo, *Data structures and algorithms for approximate string matching*, J. Complexity, 4 (1988), pp. 33–72.

[GeN] M. R. Genesereth and N. J. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufman, Los Altos, CA, 1987.

[G] R. Giancarlo, *The suffix tree of a square matrix, with applications*, in Proc. Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, TX, 1993, SIAM, Philadelphia, pp. 402–411.

[Ho] S. Horwitz, *Identifying the semantic and textual differences between two versions of a program*, Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1990, ACM, New York, pp. 234–245.

[Hu92] A. G. Hume, personal communication, November 1992.

[Ja] H. T. Jankowitz, *Detecting plagiarism in student PASCAL programs*, Comput. J., 31 (1988), pp. 1–8.

[Jo] R. Johnson, personal communication, October, 1991.

[KP] B. W. Kernigan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[McC] E. M. McCreight, *A space-economical suffix-tree construction algorithm*, J. ACM, 23 (1976), pp. 262–272.

[MM] E. W. Myers and W. Miller, *Approximate matching of regular expressions*, Bull. Math. Biol., 51 (1989), pp. 5–37.

[Tu] E. R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1983.

[Ukk] E. Ukkonen, *On-line construction of suffix trees*, Algorithmica, 14 (1995), pp. 249–260.

[We] P. Weiner, *Linear pattern matching algorithms*, in Proc. 14th Annual IEEE Symp. on Switching and Automata Theory, 1973, IEEE, Piscataway, NJ, pp. 1–11.

[WM] W. Wu and U. Mamber, *Fast text searching allowing errors*, Comm. ACM, 35 (1992), pp. 83–91.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1363–1383, October 1997

005

THE MAXIMUM LATENCY AND IDENTIFICATION OF POSITIVE BOOLEAN FUNCTIONS*

KAZUHISA MAKINO† AND TOSHIHIDE IBARAKI‡

Abstract. Consider the problem of identifying $\min T(f)$ and $\max F(f)$ of a positive (i.e., monotone) Boolean function f by using membership queries only, where $\min T(f)$ ($\max F(f)$) denotes the set of minimal true vectors (maximal false vectors) of f. It is known that an incrementally polynomial algorithm exists if and only if there is a polynomial time algorithm to check the existence of an unknown vector u for given sets $MT \subseteq \min T(f)$ and $MF \subseteq \max F(f)$; that is, $u \in \{0,1\}^n \setminus (\{v \mid v \geq w \text{ for some } w \in MT\} \cup \{v \mid v \leq w \text{ for some } w \in MF\})$. This paper introduces a measure for the difficulty to find an unknown vector, which is called the maximum latency. If the maximum latency is constant, then an unknown vector can be found in polynomial time and there is an incrementally polynomial algorithm for identification. Several subclasses of positive functions are shown to have constant maximum latency, e.g., 2-monotonic positive functions, Δ-partial positive threshold functions, and matroid functions, while the class of general positive functions has $\lfloor n/4 \rfloor + 1$ maximum latency and the class of positive k-DNF functions has $\Omega(\sqrt{n})$ maximum latency.

Key words. identification of Boolean functions, positive Boolean function, partial function, unknown vector, maximum latency, dualization, transversal

AMS subject classifications. 06E30, 68Q25, 68T05, 68R05, 94C10

PII. S0097539794276324

1. Introduction.

Consider the problem of identifying $T(f)$ (set of true vectors) and $F(f)$ (set of false vectors) of a given Boolean function (or a function in short) f by asking membership queries to an oracle whether $f(u) = 0$ or 1 holds for some selected vectors u [6]. In the terminology of computational learning theory [1, 2, 28], this is the exact learning of a Boolean theory f by membership queries only. It is also a process of forming a theory that explains a certain phenomenon by collecting positive and negative data (in the sense of causing and not causing that phenomenon) [12]. In particular, we are interested in the case where f is known to be positive, i.e., monotone. If f is a positive function, $T(f)$ and $F(f)$ can be compactly represented by $\min T(f)$ (set of minimal true vectors) and $\max F(f)$ (set of maximal false vectors). Therefore our problem is stated as follows.

PROBLEM IDENTIFICATION.

Input: A membership oracle for a positive function f.

Output: $\min T(f)$ and $\max F(f)$.

The complexity of this type of enumeration algorithm is usually measured in its length of input and output. An algorithm to enumerate items $a_1, a_2, \ldots, a_p$ is called *incrementally polynomial* [19, 21] (i) if it iterates the following procedure for $i = 1, 2, \ldots, p$: output the ith item a_i from the knowledge of its input and items $a_1, a_2, \ldots, a_{i-1}$ generated by then and (ii) if the time required for the ith iteration

* Received by the editors November 1, 1994; accepted for publication (in revised form) October 12, 1995. This research was partially supported by the Scientific Grant-in-Aid from Ministry of Education, Science and Culture of Japan. A preliminary version of this paper appeared in the *Proceedings of the 5th International Symposium on Symbolic and Algebraic Computation* (ISSAC'94), D. Z. Du and X. S. Zhang, eds., Lecture Notes in Comput. Sci. 834, Springer-Verlag, Berlin, 1994, pp. 324–332 [23].

http://www.siam.org/journals/sicomp/26-5/27632.html

† Department of Systems and Human Science, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 560, Japan (makino@sys.es.osaka-u.ac.jp).

‡ Department of Applied Mathematics and Physics, Graduate School of Engineering, Kyoto University, Kyoto 606, Japan (ibaraki@kuamp.kyoto-u.ac.jp).

is polynomial in the input length and the sizes of $a_1, a_2, \ldots, a_{i-1}$. If an algorithm is incrementally polynomial, it also satisfies the criterion of *polynomial total time* [19] (i.e., polynomial time in the length of input and output).

Now let MT and MF respectively denote the partial knowledge of $\min T(f)$ and $\max F(f)$ currently at hand, i.e.,

$$MT \subseteq \min T(f) \text{ and } MF \subseteq \max F(f). \tag{1}$$

Define

$$T(MT) = \{v \mid v \geq w \text{ for some } w \in MT\},$$
$$F(MF) = \{v \mid v \leq w \text{ for some } w \in MF\}.$$

By assumption (1), $T(MT) \subseteq T(f)$ and $F(MF) \subseteq F(f)$, and hence

$$T(MT) \cap F(MF) = \emptyset$$

holds. A vector u is called *unknown* if

$$u \in \{0,1\}^n \setminus (T(MT) \cup F(MF)),$$

since it is not known at the current stage whether u is a true vector or a false vector of f. If there is no unknown vector, then $T(MT) \cup F(MF) = \{0,1\}^n$ holds, i.e., $MT = \min T(f)$ and $MF = \max F(f)$ hold for some positive function f.

The general procedure of identifying a positive function f can be described as follows.

Algorithm IDENTIFY.

Input: A membership oracle for a positive function f.

Output: $\min T(f)$ and $\max F(f)$.

1. Start with appropriate sets MT ($\subseteq \min T(f)$) and MF ($\subseteq \max F(f)$).

2. Test if $T(MT) \cup F(MF) = \{0,1\}^n$ holds. If so, output MT and MF, and halt. Otherwise, find an unknown vector u and go to 3.

3. Ask an oracle if $f(u) = 1$ or $f(u) = 0$. If $f(u) = 1$, then compute a new minimal true vector y such that $y \leq u$ and let $MT := MT \cup \{y\}$. On the other hand, if $f(u) = 0$, compute a new maximal false vector y such that $y \geq u$ and let $MF := MF \cup \{y\}$. Return to 2.

The crucial part of this algorithm is in step 2, i.e., to solve the following problem, where a set of vectors M is *incomparable* if any pair of vectors v and w in M satisfies $v \not\geq w$ and $w \not\geq v$.

Problem EQ.

Input: Incomparable sets MT, MF ($\subseteq \{0,1\}^n$) such that $T(MT) \cap F(MF) = \emptyset$.

Question: Does $T(MT) \cup F(MF) = \{0,1\}^n$ (i.e., no unknown vector) hold?

In case of $MT, MF = \emptyset$, problem EQ is obviously polynomially solvable, since all vectors u are unknown. Hence in the following, we assume $MT \cup MF \neq \emptyset$ in step 1. If problem EQ can be solved in polynomial time, it is known that an unknown vector in step 2 can be found in polynomial time [6] and that computing a minimal true vector or a maximal false vector y from an unknown vector u in step 3 can also be done in polynomial time [1, 6, 15, 28]. Therefore, an incrementally polynomial algorithm exists if problem EQ can be solved in polynomial time. The converse direction is also known [6], and hence an incrementally polynomial algorithm exists if and only if problem EQ can be solved in polynomial time. As noted in [6], problem

EQ is polynomially equivalent to many other interesting problems encountered in various fields such as hypergraph theory [13], theory of coteries (used in distributed systems) [16, 18], artificial intelligence [27], and Boolean theory [5]. Unfortunately, the complexity of these problems is still open [6, 13, 19], though the recent result by M. Fredman and L. Khachiyan [14] shows that it is unlikely that the problem is NP-hard.

We comment here that there is a wide spectrum of research on the exact learning of Boolean functions. Many of these studies, however, are based on the model of using both membership and equivalence queries. Using such a model, N. H. Bshouty [9] showed that any Boolean function (not necessarily positive) is polynomially learnable either as DNF (disjunctive normal form) or CNF (conjunctive normal form).

In order to investigate the complexity of problem EQ, we introduce in this paper the concept of maximum latency, which is a complexity measure for finding an unknown vector. If the maximum latency is constant, then problem EQ can be solved in polynomial time, though the converse is not generally true. Based on the result by V. K. Korobkov and T. L. Reznik [20], it is noted in section 3 that the maximum latency of general positive functions is $\lfloor n/4 \rfloor + 1$. However, several special classes of positive functions are found in section 4 to have constant maximum latency, e.g., classes of (i) 2-monotonic positive functions [4, 7, 8, 11, 25, 26] (ii) Δ-partial positive threshold functions [25], (iii) k-degree positive threshold functions, (iv) matroid functions [3, 10], and (v) k-tight positive functions. For these classes of positive functions, therefore, there are incrementally polynomial identification algorithms. Finally, it is shown in section 5 that the class of positive k-DNF functions has maximum latency not less than $\Omega(\sqrt{n})$, even though it is known [13] that problem EQ can be solved in polynomial time for this class of functions.

The last result indicates that the concept of maximum latency is not always sufficient to distinguish polynomially solvable cases from those not solvable in polynomial time. However, it is also evident that the maximum latency is a powerful tool to find polynomially solvable special cases.

2. Definitions and basic properties. A *Boolean function*, or a *function* in short, is a mapping $f : \{0,1\}^n \mapsto \{0,1\}$, where $v \in \{0,1\}^n$ is called a *Boolean vector* (a *vector* in short). If $f(v) = 1$ (resp., 0), then v is called a *true* (resp., *false*) vector of f. The set of all true vectors (resp., false vectors) is denoted by $T(f)$ (resp., $F(f)$). Two special functions with $T(f) = \emptyset$ and $F(f) = \emptyset$ are, respectively, denoted by $f = \bot$ and $f = \top$. A function f is *positive* if $v \leq w$ always implies $f(v) \leq f(w)$. A positive function is also called *monotone*.[1] A true vector v of f is *minimal* if there is no other true vector w such that $w < v$, and let $\min T(f)$ denote the set of all minimal true vectors of f. A *maximal* false vector is symmetrically defined and $\max F(f)$ denotes the set of all maximal false vectors of f.

If functions f and h satisfy $h(a) \leq f(a)$ for all $a \in \{0,1\}^n$, then we denote $h \leq f$. If $h \leq f$ and there exists a vector a satisfying $h(a) = 0$ and $f(a) = 1$, we denote $h < f$. The variables $x_1, x_2, \ldots, x_n$ and their complements $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ are called *literals*. A *term* (resp., *clause*) is a conjunction (resp., disjunction) of literals such that at most one of x_i and $\bar{x}_i$ appears for each variable. A term t (resp., clause C) is called an *implicant* (resp., *implicate*) of a function f if $t \leq f$ (resp. $C \geq f$). An implicant t (resp., implicate C) of a function is called *prime* if there is no implicant

[1] The name "monotone" may be more familiar to people in theoretical computer science than "positive." However, in this paper we use "positive" to avoid confusion, since such names as 1-monomonic (or unate) and 2-monotonic used in threshold logic do not exclude nonpositive functions.

$t' > t$ (resp., no implicate $C' < C$). A DNF (resp., CNF) is a disjunction of terms (resp., conjunction of clauses).

If f is positive, it is known that f has (i) the unique DNF consisting of all prime implicants and (ii) the unique CNF consisting of all prime implicates. There is one-to-one correspondence between prime implicants (resp., prime implicates) and minimal true vectors (resp., maximal false vectors). For example, a positive function $f = x_1x_2 \vee x_2x_3x_4 = x_2(x_1 \vee x_3)(x_1 \vee x_4)$ has prime implicants x_1x_2 and $x_2x_3x_4$, which respectively correspond to minimal true vectors (1100) and (0111), and prime implicates x_2, $(x_1 \vee x_3)$, and $(x_1 \vee x_4)$, which respectively correspond to maximal false vectors (1011), (0101), and (0110). In other words, the input length to describe a positive function f is $O(n|\min T(f)|)$ (resp., $O(n|\max F(f)|)$) if it is represented by DNF (resp., CNF).

Sets $\min T(f)$ and $\max F(f)$, respectively, define $T(f)$ and $F(f)$ by

$$T(f) = \{v \mid v \geq a \text{ for some } a \in \min T(f)\},$$
$$F(f) = \{v \mid v \leq b \text{ for some } b \in \max F(f)\}.$$

DEFINITION 2.1. *Given incomparable sets MT, MF ($\subseteq \{0,1\}^n$) such that $MT \cup MF \neq \emptyset$ and $T(MT) \cap F(MF) = \emptyset$, the partial function g is defined by*

$$g(v) = \begin{cases} 1, & v \in T(MT), \\ 0, & v \in F(MF), \\ \textit{unknown}, & \textit{otherwise}. \end{cases}$$

If MT and MF of g satisfy $MT \subseteq \min T(f)$ and $MF \subseteq \max F(f)$ for some (complete) positive function f, then g is called a partial function of f. *The set of unknown vectors of g is denoted by $U(g)$, i.e.,*

$$U(g) = \{0,1\}^n \setminus (T(MT) \cup F(MF)).$$

The k-neighborhood of g (defined by MT and MF) is given by

$$N_k(g) = \{v \mid \| v - a \| \leq k \text{ for some } a \in MT \cup MF\},$$

where $\| w \|$ denotes $\sum_{i=1}^{n} |w_i|$; i.e., $N_k(g)$ is the set of all vectors within Hamming distance k of $MT \cup MF$. The latency *of g, $\lambda(g)$, is defined by the integer k satisfying*

$$N_{k-1}(g) \cap U(g) = \emptyset \quad \textit{and} \quad N_k(g) \cap U(g) \neq \emptyset.$$

As a special case, if $U(g) = \emptyset$, i.e., $g = f$, then $\lambda(g)$ is defined to be 0. $\lambda(g)$ is equivalently given by

$$\lambda(g) = \min\{\| u - a \| \mid a \in MT \cup MF, u \in U(g)\}.$$

Example 2.2. Let $f = x_1x_3 \vee x_2x_4$, i.e.,

$$\min T(f) = \{1010, 0101\},$$
$$\max F(f) = \{0011, 0110, 1001, 1100\},$$

and let g be a partial function defined by

$$MT = \{1010, 0101\},$$
$$MF = \{0011, 0110, 1001\}.$$

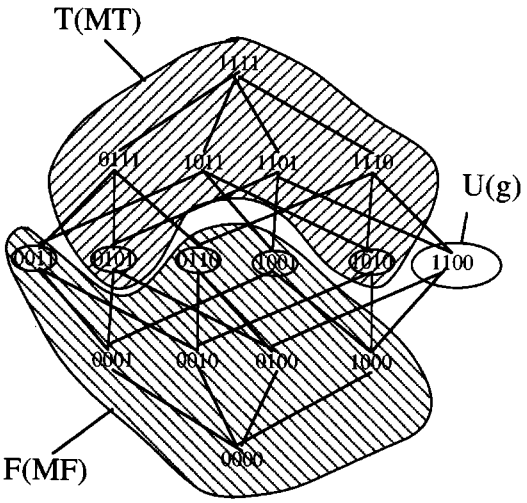


FIG. 1. *The partial function g of Example* 2.2.

Then the set of unknown vectors is

$$U(g) = \{1100\},$$

and we obtain $\lambda(g) = 2$ (see Figure 1).

DEFINITION 2.3. *Let C_X be a subclass of positive functions. $C_X(n)$ denotes the set of functions in C_X with n variables. For $C_X(n)$, the* maximum latency *is defined by*

$$\Lambda_X(n) = \max\{\lambda(g) \mid g \text{ is a partial function of } f \in C_X(n)\}.$$

If g is a partial function of $f \in C_X(n)$, then by definition there is no unknown vector if $N_{\Lambda_X(n)}(g) \cap U(g) = \emptyset$. That is, in order to find an unknown vector, we need only to search $\Lambda_X(n)$-neighborhood of g. Therefore, if a positive function f of n variables is known to belong to class $C_X(n)$, step 2 of Algorithm IDENTIFY can be executed as follows.

2. Test if $N_{\Lambda_X(n)}(g) \subseteq T(MT) \cup F(MF)$, where g is the partial function defined by MT and MF. If so, output MT and MF, and halt. Otherwise, find an unknown vector $u \in N_{\Lambda_X(n)}(g) \setminus (T(MT) \cup F(MF))$ and go to 3.

The test of $N_{\Lambda_X(n)}(g) \subseteq T(MT) \cup F(MF)$ can be accomplished by checking if $v \geq a$ for some $a \in MT$ or $v \leq b$ for some $b \in MF$, for every $v \in N_{\Lambda_X(n)}(g)$. This computation takes at most

$$n(|MT| + |MF|)\,|N_{\Lambda_X(n)}(g)| \leq n(|MT| + |MF|)^2\, n^{\Lambda_X(n)}$$

time. Therefore, we have the next result.

THEOREM 2.4. *Let g, defined by MT and MF, be a partial function of $f \in C_X(n)$. If $\Lambda_X(n)$ is constant, then an unknown vector can be found in polynomial time in n and $|MT| + |MF|$. (Therefore, in this case, there is an incrementally polynomial algorithm to identify $f \in C_X(n)$.)*

3. Maximum latency of general positive functions. Let

C_P: class of all positive functions.

It was shown by V. K. Korobkov and T. L. Reznik [20] that $\Lambda_P(n) \leq \lfloor n/4 \rfloor + 1$ for all n, and that $\Lambda_P(n) \geq \lfloor n/4 \rfloor + 1$ for $n = 4k$. Similar results were independently obtained in [22]. We complement these results by showing the next lemma. For a vector $v \in \{0,1\}^n$, use the notations $ON(v) = \{j \mid v_j = 1,\ j = 1,2,\ldots,n\}$ and $OFF(v) = \{j \mid v_j = 0,\ j = 1,2,\ldots,n\}$.

LEMMA 3.1. *$\Lambda_P(n) \geq \lfloor n/4 \rfloor + 1$ for all positive integers n.*

Proof. For $n = 1,2,3$, it is clear that $\Lambda_P(n) = \lfloor n/4 \rfloor + 1 = 1$. We show $\Lambda_P(n) \geq \lfloor n/4 \rfloor + 1$ for $n \geq 4$ by providing an example of g with $\lambda(g) = \lfloor n/4 \rfloor + 1$. In this example (MT, MF), there is exactly one unknown vector u. Hence, there is a vector $v \in MT$ (resp., $v \in MF$) such that $v \leq u + e_i$ (resp., $v \geq u - e_i$) for every $i \in OFF(u)$ (resp., $i \in ON(u)$), where e_i denotes the unit vector whose ith component is 1. MT and MF are constructed in such a way that all vectors $v \in MT \cup MF$ are used for the above purpose and satisfy $\| v - u \| \geq \lfloor n/4 \rfloor + 1$.

(1) $n = 4k$ $(k \geq 1)$: Define a $2k \times 4k$ matrix

$$X = \left(\begin{array}{cc|c} J_k & O & \\ \hline O & J_k & I_{2k} \end{array}\right)$$

and a $(k^2 + 2k) \times 4k$ matrix

$$Y = \left(\begin{array}{c|cc} Q & \multicolumn{2}{c}{J_{k^2 \times 2k}} \\ \hline J_{2k} - I_{2k} & \begin{array}{c|c} J_k & O \\ \hline O & J_k \end{array} & \end{array}\right),$$

where J_i is the $i \times i$ matrix whose elements are all 1's, $J_{i \times j}$ is the $i \times j$ matrix whose elements are all 1's, I_i is the $i \times i$ identity matrix, O is the $k \times k$ zero matrix, and Q is the $k^2 \times 2k$ matrix defined by

$$Q = \left(\begin{array}{ccccc|c} 0 & 1 & \ldots & \ldots & 1 & \\ 0 & 1 & \ldots & \ldots & 1 & \\ \multicolumn{5}{c|}{\ldots\ldots\ldots\ldots\ldots} & J_k - I_k \\ 0 & 1 & \ldots & \ldots & 1 & \\ \hline 1 & 0 & 1 & \ldots & 1 & \\ \multicolumn{5}{c|}{\ldots\ldots\ldots\ldots\ldots} & J_k - I_k \\ 1 & 0 & 1 & \ldots & 1 & \\ \hline \multicolumn{5}{c|}{\ldots\ldots\ldots\ldots\ldots} & \ldots \\ \hline 1 & \ldots & \ldots & 1 & 0 & \\ \multicolumn{5}{c|}{\ldots\ldots\ldots\ldots\ldots} & J_k - I_k \\ 1 & \ldots & \ldots & 1 & 0 & \end{array}\right) \begin{array}{l} \left.\vphantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} k \\ \left.\vphantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} k. \\ \vdots \\ \left.\vphantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} k \end{array}$$

A partial function g is then defined by $MT = \{x^{(j)} \mid j = 1,2,\ldots,2k\}$ and $MF = \{y^{(j)} \mid j = 1,2,\ldots,k^2 + 2k\}$, where $x^{(j)}$ is the jth row of X and $y^{(j)}$ is the jth row of Y. Furthermore, let

$$u = (\underbrace{1,1,\ldots,1}_{2k},\underbrace{0,0,\ldots,0}_{2k}).$$

It is clear that $T(MT) \cap F(MF) = \emptyset$ and $u \notin T(MT) \cup F(MF)$; i.e., u is an unknown vector. Furthermore, $u - e_i \in F(MF)$ for every $i \in ON(u)$ and $u + e_i \in T(MT)$ for every $i \in OFF(u)$, where e_i denotes the unit vector whose ith component is 1. Consider the positive function f satisfying $\min T(f) = MT$. Then it is not difficult to see that $\max F(f) = \{u\} \cup MF$ by using a relation

$$\max F(f) = \{\bar{v} \mid v \in \min T(f^d)\},$$

where f^d is the dual of f. Therefore, $T(MT) \cup F(MF) \cup \{u\} = \{0,1\}^n$; i.e., u is the only unknown vector. Finally, $\| x^{(j)} - u \| = k + 1 > k$ for every j $(1 \le j \le 2k)$, $\| y^{(j)} - u \| = 2k + 2 > k$ for every j $(1 \le j \le k^2)$, and $\| y^{(j)} - u \| = k + 1 > k$ for every j $(k^2 < j \le k^2 + 2k)$. This proves that $N_k(g) \cap U(g) = \emptyset$ and $N_{k+1}(g) \cap U(g) \neq \emptyset$.

(2) $n = 4k + \alpha$ $(k \ge 1,\ 1 \le \alpha \le 3)$: Define a $2k \times (4k + \alpha)$ matrix

$$X = \left(\begin{array}{c|c|c|c} J_k & O & I_{2k} & J_{2k \times \alpha} \\ \hline O & J_k & & \end{array} \right)$$

and a $(k^2 + 2k + \alpha) \times (4k + \alpha)$ matrix

$$Y = \left(\begin{array}{c|c|c} Q & \begin{array}{c|c} J_k & O \\ \hline O & J_k \end{array} & J_{(k^2+2k) \times \alpha} \\ \hline J_{2k} - I_{2k} & & \\ \hline J_{\alpha \times 4k} & & J_\alpha - I_\alpha \end{array} \right).$$

Define a partial function g by $MT = \{x^{(j)} \mid j = 1, 2, \ldots, 2k\}$ and $MF = \{y^{(j)} \mid j = 1, 2, \ldots, k^2 + 2k + \alpha\}$. Furthermore, let

$$u = (\underbrace{1, 1, \ldots, 1}_{2k}, \underbrace{0, 0, \ldots, 0}_{2k}, \underbrace{1, 1, \ldots, 1}_{\alpha}).$$

Similarly to case (1), it can be shown that $U(g) = \{u\}$, and $N_k(g) \cap U(g) = \emptyset$ and $N_{k+1}(g) \cap U(g) \neq \emptyset$ for $k = \lfloor n/4 \rfloor$. □

In Example 2.2, it is easy to see that g is the partial function constructed for $k = 1$ in the proof (1) of the above lemma.

THEOREM 3.2. *Class C_P satisfies*

$$\Lambda_P(n) = \lfloor n/4 \rfloor + 1.$$

Proof. Combine the result in [20] and Lemma 3.1. □

4. Restricted classes of positive functions with constant maximum latencies. In this section, we find some restricted classes of positive functions with constant maximum latency, which are important in practice and theory (e.g.,[2, 13, 25]). As stated in Theorem 2.4, functions in these classes can be identified by incrementally polynomial algorithms.

4.1. 2-monotonic positive functions. An assignment A of binary values 0 or 1 to k variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ is called a *k-assignment* and is denoted by

$$A = (x_{i_1} \leftarrow a_1, x_{i_2} \leftarrow a_2, \ldots, x_{i_k} \leftarrow a_k),$$

where each of $a_1, a_2, \ldots, a_k$ is either 1 or 0. Let the complement of A, denoted by $\bar{A}$, represent the assignment obtained from A by complementing all the 1's and 0's in A. When a function f of n variables and a k-assignment A are given,

$$f_A = f_{(x_{i_1} \leftarrow a_1,\, x_{i_2} \leftarrow a_2, \ldots, x_{i_k} \leftarrow a_k)}$$

denotes the function of $(n-k)$ variables obtained by fixing variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ as specified by A.

Let f be a function of n variables. If either $f_A \leq f_{\bar{A}}$ or $f_A \geq f_{\bar{A}}$ holds for every k-assignment A, then f is said to be *k-comparable*. If f is k-comparable for every k such that $1 \leq k \leq m$, then f is said to be *m-monotonic*. (For more detailed discussion on these topics, see [25] for example.) In particular, f is 1-monotonic if $f_{(x_i \leftarrow 1)} \geq f_{(x_i \leftarrow 0)}$ or $f_{(x_i \leftarrow 1)} \leq f_{(x_i \leftarrow 0)}$ holds for any $i \in \{1, 2, \ldots, n\}$. A 1-monotonic function is also called a *unate* function. A positive function is precisely a 1-monotonic function f for which $f_{(x_i \leftarrow 1)} \geq f_{(x_i \leftarrow 0)}$ holds for all i.

Now consider a 2-assignment $A = (x_i \leftarrow 1, x_j \leftarrow 0)$. If

$$f_A \geq f_{\bar{A}} \ \text{(resp., } f_A > f_{\bar{A}}\text{)}$$

holds, this is denoted by $x_i \succeq_f x_j$ (resp., $x_i \succ_f x_j$). Variables x_i and x_j are said to be *comparable* if either $x_i \succeq_f x_j$ or $x_i \preceq_f x_j$ holds. When $x_i \succeq_f x_j$ and $x_i \preceq_f x_j$ hold simultaneously, it is denoted as $x_i \approx_f x_j$. If f is 2-monotonic, this binary relation $\succeq_f$ over the set of variables is known to be a total preorder (i.e., reflexive, transitive, and comparable) [25]. A 2-monotonic positive function f of n variables is called *regular* if

$$x_1 \succeq_f x_2 \succeq_f \cdots \succeq_f x_n.$$

Any 2-monotonic positive function becomes regular by permuting variables.

The 2-monotonicity was originally introduced in conjunction with threshold functions (e.g., [25]), where a positive function f is *threshold* if there exist $n+1$ nonnegative real numbers $w_1, w_2, \ldots, w_n$ and t such that

$$f(x) = \begin{cases} 1 & \text{if } \sum w_i x_i \geq t, \\ 0 & \text{if } \sum w_i x_i < t. \end{cases}$$

As $w_i \geq w_j$ implies $x_i \succeq_f x_j$ and $w_i = w_j$ implies $x_i \approx_f x_j$, a threshold function is always 2-monotonic, although the converse is not true [25]. Let

C_{2M}: class of 2-monotonic positive functions,
C_{TH}: class of positive threshold functions.

THEOREM 4.1. *Class C_{2M} satisfies*

$$\Lambda_{2M}(n) = 1.$$

Proof. Assume that a 2-monotonic positive function f is regular without loss of generality and that g is a partial function of f defined by MT and MF. Assume that $N_1(g) \cap U(g) = \emptyset$ and $U(g) \neq \emptyset$ (i.e., $\Lambda_{2M}(n) > 1$). Take a $u \in \max U(g)$, where $\max U(g)$ is the set of maximal unknown vectors (i.e., $u + e_j \in T(MT)$ for all $j \in OFF(u)$). Let $j = \max\{i \mid i \in OFF(u)\}$. Since $u + e_j \in T(MT)$, there exists an $a \in MT$ such that $a \leq u + e_j$. Note that $a_j = 1$ holds since otherwise $a \leq u$, contradicting $u \in U(g)$. Then $a - e_j \in F(MF)$ by the assumption $N_1(g) \cap U(g) = \emptyset$. Therefore, there exists a $b \in MF$ such that $b \geq a - e_j$. This b satisfies $b_j = 0$ since otherwise $b \geq a$, a contradiction. For any $l \in OFF(u) \setminus \{j\}$,

$$a - e_j + e_l \in T(f) \subseteq T(MT) \cup U(g)$$

by regularity of f, and hence $b \not\geq a - e_j + e_l$, i.e., $b_l = 0$ for all $l \in OFF(u)$ and hence $b \leq u$. (i) If $b = u$, then $u \in F(MF)$, which is a contradiction. (ii) If $b < u$, then $u \in T(MT)$ by $N_1(g) \cap U(g) = \emptyset$, which is also a contradiction. □

Since the complexity of problem IDENTIFICATION is still open, E. Boros et al. [7, 8] considered the following restricted problem.

PROBLEM IDENTIFICATION-2M.

Input: A membership oracle for a positive function f.

Output: If f is 2-monotonic, then $\min T(f)$ and $\max F(f)$; otherwise, "no."

Two algorithms were proposed therein; one uses $O(n^3m)$ time and $O(n^3m)$ queries, while the other uses $O(nm^2+n^2m)$ time and $O(nm)$ queries, where $m = |\min T(f)| + |\max F(f)|$. Based on the concept of maximum latency, we could recently construct a new algorithm for IDENTIFICATION-2M, which uses $O(n^2m)$ time and $O(n^2m)$ queries [24]. Since $m \gg n$ can be expected in usual cases, this is an improvement over the previous two algorithms.

COROLLARY 4.2. *Class C_{TH} satisfies*

$$\Lambda_{TH}(n) = 1.$$

4.2. Δ-partial positive threshold functions.

DEFINITION 4.3 (see [25]). *A positive function f is called Δ-partial threshold if f is represented by*

$$f(x) = \begin{cases} 1 & \text{if } \sum w_i x_i \geq t + \alpha, \\ 0 & \text{if } \sum w_i x_i < t - \alpha, \\ 0 \text{ or } 1 & \text{otherwise,} \end{cases}$$

where $w_i\,(i = 1, 2, \ldots, n)$, t and Δ are nonnegative real numbers, and

$$\alpha = \Delta w_{\min},$$

where $w_{\min} = \min_i w_i$. Associated with these $w_i\,(i = 1, 2, \ldots, n)$ and t, define the threshold function h *by*

$$h(x) = \begin{cases} 1 & \text{if } \sum w_i x_i \geq t, \\ 0 & \text{if } \sum w_i x_i < t. \end{cases}$$

Then the above f is called a Δ-partial threshold function of h.

In the above definition of f, the value $f(x)$ in the case of "otherwise" can be determined arbitrary, provided that the resulting f is positive.

Example 4.4. Let h be a threshold function represented by

$$h(x) = \begin{cases} 1 & \text{if } 2x_1 + x_2 + x_3 + x_4 \geq 3, \\ 0 & \text{if } 2x_1 + x_2 + x_3 + x_4 < 3, \end{cases}$$

i.e., $h = x_1x_2 \vee x_1x_3 \vee x_1x_4 \vee x_2x_3x_4$, and

$$\min T(h) = \{1100, 1010, 1001, 0111\},$$
$$\max F(h) = \{0011, 0101, 0110, 1000\}.$$

Consider $f = x_1x_2 \vee x_1x_3 \vee x_2x_4$, i.e.,

$$\min T(f) = \{1100, 1010, 0101\},$$
$$\max F(f) = \{0011, 0110, 1001\}.$$

This f is a 1-partial threshold function of h represented by

$$f(x) = \begin{cases} 1 & \text{if } 2x_1 + x_2 + x_3 + x_4 \geq 4, \\ 0 & \text{if } 2x_1 + x_2 + x_3 + x_4 < 2, \\ 0 \text{ or } 1 & \text{otherwise,} \end{cases}$$

where $\alpha = \min_i w_i = 1$.

Let

$C_{\Delta PTH}$: class of Δ-partial positive threshold functions.

THEOREM 4.5. *Class $C_{\Delta PTH}$ satisfies*

$$\Lambda_{\Delta PTH}(n) \leq \lceil \Delta \rceil + 1.$$

Proof. Let f be a Δ-partial threshold function of a threshold function h. We assume $w_1 \geq w_2 \geq \cdots \geq w_n$ without loss of generality. Assuming that g is a partial function of f defined by MT and MF such that $N_{\lceil \Delta \rceil + 1}(g) \cap U(g) = \emptyset$ and $U(g) \neq \emptyset$, we derive a contradiction. Take a $u \in \max U(g)$, and let $j = \max\{i \mid i \in OFF(u)\}$. There exists an $a \in MT$ such that $a \leq u + e_j$. Since $\| u - a \| \geq \lceil \Delta \rceil + 2$ by assumption, $\| u - (a - e_j) \| \geq \lceil \Delta \rceil + 1$ holds, i.e., $|ON(u) \setminus ON(a)| \geq \lceil \Delta \rceil + 1$. Let

$$c^{(1)} = a - e_j + \sum_{l=1}^{\lceil \Delta \rceil} e_{i_l},$$

where i_l, $l = 1, 2, \ldots, \lceil \Delta \rceil$, are arbitrary $\lceil \Delta \rceil$ indices in $ON(u) \setminus ON(a)$. Note that $c^{(1)} \notin U(g)$ (i.e., $c^{(1)} \in T(MT) \cup F(MF)$) because $a \in MT$, $\| a - c^{(1)} \| = \lceil \Delta \rceil + 1$, and, by assumption, $\| x - y \| \geq \lceil \Delta \rceil + 2$ for any pair of $x \in MT$ and $y \in U(g)$. If $c^{(1)} \in T(MT)$, then $u \in T(MT)$ holds since $u \geq c^{(1)}$, which is a contradiction. If $c^{(1)} \in F(MF)$, then there exists a $b^{(1)} \in MF$ such that $b^{(1)} \geq c^{(1)}$. (i) If $b^{(1)} \geq u$, then $u \in F(MF)$, which is a contradiction. (ii) If $b^{(1)} < u$, then $b^{(1)} \in MF$ and $N_{\lceil \Delta \rceil + 1}(g) \cap U(g) = \emptyset$ imply $b^{(1)} + e_i \in T(MT)$ for any $i \in OFF(b^{(1)}) \cap ON(u)$. Since $u \geq b^{(1)} + e_i$, this means $u \in T(MT)$, which is a contradiction. (iii) Otherwise (i.e., $b^{(1)}$ and u are incomparable), $|ON(b^{(1)}) \setminus ON(u)| \neq \emptyset$ holds. We consider the following two cases.

Case 1. $a \in MT \cap T(h)$: Take a $p \in ON(b^{(1)}) \setminus ON(u)$. Then

$$\begin{aligned} \sum w_i c_i^{(1)} + w_p &= \sum w_i a_i - w_j + \sum_{l=1}^{\lceil \Delta \rceil} w_{i_l} + w_p \\ &\geq t + \alpha + (w_p - w_j) \\ &\geq t + \alpha \quad (\text{by } w_p \geq w_j). \end{aligned}$$

Therefore, $c^{(1)} + e_p \in T(f)$, a contradiction to $c^{(1)} + e_p \leq b^{(1)} \in MF$.

Case 2. $a \in MT \cap F(h)$: Let $ON(b^{(1)}) \setminus ON(u) = \{p_1, p_2, \ldots, p_s\}$. If $s \geq \lceil \Delta \rceil + 1$, then

$$\begin{aligned} \sum w_i b_i^{(1)} &\geq \sum w_i c_i^{(1)} + \sum_{l=1}^{s} w_{p_l} \\ &= \sum w_i a_i - w_j + \sum_{l=1}^{\lceil \Delta \rceil} w_{i_l} + \sum_{l=1}^{s} w_{p_l}. \end{aligned} \tag{2}$$

Since we have $\sum w_i a_i \geq t - \alpha$ (by $a \in MT \cap F(h)$), $\sum_{l=1}^{\lceil \Delta \rceil} w_{i_l} \geq \Delta w_{\min} = \alpha$, and

$$\begin{aligned}\sum_{l=1}^{s} w_{p_l} &= w_{p_1} + \sum_{l=2}^{s} w_{p_l} \\ &\geq w_{p_1} + \alpha \quad (\text{by } \ s \geq \lceil \Delta \rceil + 1),\end{aligned}$$

the right-hand side of (2) is at least

$$\begin{aligned}(t-\alpha) - w_j + \alpha + (w_{p_1} + \alpha) &\geq t + \alpha + (w_{p_1} - w_j) \\ &\geq t + \alpha \quad (\text{by } \ w_{p_1} \geq w_j).\end{aligned}$$

This implies $b^{(1)} \in T(f)$, which is a contradiction. Therefore, we conclude $s \leq \lceil \Delta \rceil$. Now, $u \geq b^{(1)} - \sum_{l=1}^{s} e_{p_l}$ implies $u > b^{(1)} - \sum_{l=1}^{s} e_{p_l}$ since $N_{\lceil \Delta \rceil + 1}(g) \cap U(g) = \emptyset$. In other words, there exists a $p' \in ON(u) \setminus ON(b^{(1)})$. Define

$$c^{(2)} = b^{(1)} - \sum_{l=1}^{s} e_{p_l} + e_{p'} \ (\in T(MT) \cup F(MF) \quad \text{by } \ \| \, b^{(1)} - c^{(2)} \, \| \leq \lceil \Delta \rceil + 1).$$

Then $c^{(2)} > c^{(1)}$ since $b^{(1)} - \sum_{l=1}^{s} e_{p_l} \geq c^{(1)}$. If $c^{(2)} \in T(MT)$, then $u \in T(MT)$ holds since $u \geq c^{(2)}$, which is a contradiction. If $c^{(2)} \in F(MF)$, then there exists a $b^{(2)} \in MF$ such that $b^{(2)} \geq c^{(2)}$. (i) If $b^{(2)} \geq u$, then $u \in F(MF)$, a contradiction. (ii) If $b^{(2)} < u$, then similar to the case $b^{(1)} < u$, $u \in T(MT)$, a contradiction. (iii) Otherwise (i.e., $b^{(2)}$ and u are incomparable),

$$ON(b^{(2)}) \setminus ON(u) = \{q_1, q_2, \ldots, q_r\} \text{ and } 0 < r \leq \lceil \Delta \rceil$$

can be shown in a manner similar to $b^{(1)}$. Then define

$$c^{(3)} = b^{(2)} - \sum_{l=1}^{r} e_{q_l} + e_{q'} \in T(MT) \cup F(MF),$$

where $q' \in ON(u) \setminus ON(b^{(2)})$. Then $c^{(3)} > c^{(2)}$ holds since $b^{(2)} - \sum_{l=1}^{r} e_{q_l} \geq c^{(2)}$.

However, this procedure cannot be continued indefinitely, since

$$c^{(1)} < c^{(2)} < c^{(3)} < \cdots < u$$

and the distance between u and $c^{(1)}$ is finite, and it completes the proof. □

As an example, we construct a partial function g of $f \in C_{\Delta PTH}$ of Example 4.4 by

$$\begin{aligned}MT &= \min T(f) - \{1100\} = \{1010, 0101\}, \\ MF &= \max F(f) = \{0011, 0110, 1001\}.\end{aligned} \tag{3}$$

In this case, $U(g) = \{1100\}$ and $\lambda(g) = \lceil 1 \rceil + 1 = 2$.

4.3. *k*-degree positive threshold functions.

DEFINITION 4.6. *A function f is called a* k-degree positive threshold function *if*

$$f(x) = \begin{cases} 1 & \text{if } \sum w_i x_i + \sum_{i_1 < i_2} w_{i_1 i_2} x_{i_1} x_{i_2} + \cdots + \sum_{i_1 < \cdots < i_k} w_{i_1 \ldots i_k} x_{i_1} \ldots x_{i_k} \geq t, \\ 0 & \text{otherwise,} \end{cases}$$

where all weights $w_{i_1 i_2 \cdots i_{k'}}$ $(1 \leq k' \leq k)$ and threshold t are nonnegative.

Example 4.7. Let f be a 2-degree threshold function represented by

$$f(x) = \begin{cases} 1 & \text{if } 5x_1 + 5x_2 + 3x_3 + 3x_4 + x_1x_3 + x_2x_4 \geq 9, \\ 0 & \text{otherwise.} \end{cases}$$

Then $f = x_1x_2 \vee x_1x_3 \vee x_2x_4$, and

$$\begin{aligned} &\min T(f) = \{1100, 1010, 0101\}, \\ &\max F(f) = \{0011, 0110, 1001\}. \end{aligned}$$

Note that a 1-degree threshold function is simply a threshold function. In the following, we show that, if the sum of higher terms

$$\sum_{i_1<i_2} w_{i_1i_2} + \sum_{i_1<i_2<i_3} w_{i_1i_2i_3} + \cdots + \sum_{i_1<\cdots<i_k} w_{i_1\ldots i_k}$$

is relatively small (i.e., higher terms are used only to perturb linear terms), so is the maximum latency.

LEMMA 4.8. *If a k-degree positive threshold function f satisfies*

$$2\Delta w_{\min} \geq \sum_{l=2}^{k} \binom{n}{l} \max_{i_1<i_2<\cdots<i_l} w_{i_1i_2\ldots i_l},$$

where $w_{\min} = \min_i w_i$, then f is a Δ-partial positive threshold function.

Proof. Let f be represented by

$$f(x) = \begin{cases} 1 \text{ if } \sum w_ix_i + \sum_{i_1<i_2} w_{i_1i_2}x_{i_1}x_{i_2} + \cdots + \sum_{i_1<\cdots<i_k} w_{i_1\ldots i_k}x_{i_1}\ldots x_{i_k} \geq t, \\ 0 \text{ otherwise,} \end{cases}$$

and let h be the threshold function represented by

$$h(x) = \begin{cases} 1 & \text{if } \sum w_ix_i \geq t - \alpha, \\ 0 & \text{if } \sum w_ix_i < t - \alpha, \end{cases}$$

where $\alpha = \Delta w_{\min}$. We show that f is represented by

$$f(x) = \begin{cases} 1 & \text{if } \sum w_ix_i \geq t, \\ 0 & \text{if } \sum w_ix_i < t - 2\alpha, \\ 0 \text{ or } 1 & \text{otherwise,} \end{cases}$$

which implies that $f \in C_{\Delta PTH}$.

(i) If $\sum w_ix_i \geq t$, then it is trivial that $f(x) = 1$ since all weights $w_{i_1i_2\cdots i_{k'}}$ $(1 \leq k' \leq k)$ are nonnegative.

(ii) If $\sum w_ix_i < t - 2\alpha$, then

$$\begin{aligned} \sum w_ix_i + \sum_{i_1<i_2} w_{i_1i_2}x_{i_1}x_{i_2} + \cdots + \sum_{i_1<\cdots<i_k} w_{i_1\ldots i_k}x_{i_1}\ldots x_{i_k} \\ &< t - 2\alpha + \sum_{l} \binom{n}{l} \max_{i_1<i_2<\cdots<i_l} w_{i_1i_2\ldots i_l} \\ &< t - 2\alpha + 2\alpha = t, \end{aligned}$$

i.e., $f(x) = 0$. □

In Example 4.7, since

$$2\Delta w_{\min} = 6\Delta \geq \binom{4}{2} \max_{i_1<i_2} w_{i_1 i_2} = 6$$

holds for $\Delta = 1$, f is 1-partial threshold. In fact, with $t = 6$ and $\alpha = \Delta w_{\min} = 3$, f can be represented by

$$f(x) = \begin{cases} 1 & \text{if } 5x_1 + 5x_2 + 3x_3 + 3x_4 \geq 9, \\ 0 & \text{if } 5x_1 + 5x_2 + 3x_3 + 3x_4 < 3, \\ 0 \text{ or } 1 & \text{otherwise.} \end{cases}$$

Combining Lemma 4.8 with Theorem 4.5, we establish the next theorem.

THEOREM 4.9. *For the class of k-degree positive threshold functions, which satisfy*

$$2\Delta w_{\min} \geq \sum_{l=2}^{k} \binom{n}{l} \max_{i_1<i_2<\cdots<i_l} w_{i_1 i_2 \ldots i_l},$$

its maximum latency is not greater than $\lceil \Delta \rceil + 1$.

As an example, let us construct a partial function g of f of Example 4.7 by using MT and MF of (3). In this case, we have $U(g) = \{1100\}$ and $\lambda(g) = \lceil 1 \rceil + 1 = 2$.

4.4. Matroid functions.

DEFINITION 4.10. *A positive function f is called a* matroid *function if for each $v, w \in \min T(f)$ and each $i \in ON(v) \setminus ON(w)$ there exists a $j \in ON(w) \setminus ON(v)$ such that $v - e_i + e_j \in \min T(f)$.*

In other words, for a matroid function f of n variables, $M = (E, \mathcal{F})$ forms a *matroid* [29], where $E = \{1, 2, \ldots, n\}$ and $\mathcal{F} = \{ON(v) \mid v \leq a \text{ for some } a \in \min T(f)\}$. Minimal true vectors of f correspond to the *bases* of matroid M. Let

$$C_{MAT}: \quad \text{class of matroid functions.}$$

From the definition of a matroid function, there exists a nonnegative integer $r\,(\leq n)$ such that $|ON(a)| = r$ for every $a \in \min T(f)$. This r is called the *rank* of f. Matroid functions provide a very rich class of examples [3], including the following example of spanning tree functions [3, 10]. For an undirected graph $G = (V, E)$ where $E = \{1, 2, \ldots n\}$, f is called a *spanning tree function* of G if it is given by

$$f(v) = \begin{cases} 1 \text{ if } G_v = (V, ON(v)) \text{ is connected,} \\ 0 \text{ if } G_v = (V, ON(v)) \text{ is disconnected.} \end{cases}$$

For any $a \in \min T(f)$, $ON(a)$ is a *spanning tree* of G and for any $b \in \max F(f)$, $OFF(b)$ is a *minimal cut* of G. The rank of spanning tree function is $r = |V| - 1$.

First, we derive a lower bound on Λ_{MAT}.

LEMMA 4.11. *Class C_{MAT} of matroid functions satisfies*

$$\Lambda_{MAT}(n) = 1 \quad \textit{if } n = 1, 2, 3,$$
$$\Lambda_{MAT}(n) \geq 2 \quad \textit{if } n \geq 4.$$

Proof. For $n = 1, 2, 3$, it is clear by examining all matroid functions that $\Lambda_{MAT}(n) = 1$. For $n \geq 4$, we provide an example of g with $\lambda(g) = 2$. Consider a spanning tree function f of Figure 2.

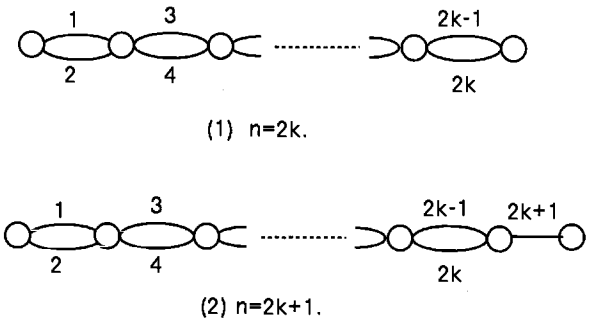


FIG. 2. *The graph g of the proof of Lemma* 4.11.

(1) $n = 2k$ $(k \geq 2)$: Let

$$\min T(f) = \{a \mid |ON(a) \cap \{2j-1, 2j\}| = 1 \quad \text{for all } j = 1, 2, \ldots, k\},$$
$$\max F(f) = \{b \mid OFF(b) = \{2j-1, 2j\} \quad \text{for some } j = 1, 2, \ldots, k\}.$$

Then we construct a partial function g of f by

$$MT = \min T(f) - \{u\},$$
$$MF = \max F(f),$$

where $u = (0101\cdots01)$. For any j, $u + e_{2j-1} \geq u + e_{2j-1} - e_{2j} \in MT$, and hence $u + e_{2j-1} \in T(MT)$. Similarly, $u - e_{2j} \in F(MF)$ for any j. Therefore, $T(MT) \cup F(MF) \cup \{u\} = \{0,1\}^n$. For every $a \in MT$, $\| a \| = \| u \| = k$ and $a \neq u$ imply $\| u - a \| \geq 2$. For every $b \in MF$, we have $u \not\leq b$. Furthermore, $\| b \| = n - 2$ and $\| u \| = k$ imply $u \not\geq b$ and

$$\| u - b \| \geq 2.$$

Therefore, we obtain $\lambda(g) \geq 2$.

(2) $n = 2k + 1$ $(k \geq 2)$: Let

$$\min T(f) = \{a \mid 2k+1 \in ON(a) \text{ and } |ON(a) \cup \{2j-1, 2j\}| = 1 \text{ for all } j = 1, 2, \ldots, k\},$$
$$\max F(f) = \{b \mid OFF(b) = \{2j-1, 2j\} \text{ for some } j = 1, 2, \ldots, k\} \cup \{11\cdots10\},$$

and construct a partial function g of f by

$$MT = \min T(f) - \{0101\cdots011\}$$
$$MF = \max F(f).$$

Then $U(g) = \{0101\cdots011\}$. The rest is similar to case (1), and we have $\lambda(g) \geq 2$. □

Example 4.12. For $n = 4$, the above construction gives $f = x_1x_3 \vee x_1x_4 \vee x_2x_3 \vee x_2x_4$, i.e.,

$$\min T(f) = \{1010, 1001, 0110, 0101\},$$
$$\max F(f) = \{0011, 1100\}$$

and

$$MT = \min T(f) - \{0101\} = \{1010, 1001, 0110\},$$
$$MF = \max F(f) = \{0011, 1100\},$$
$$U(g) = \{0101\}.$$

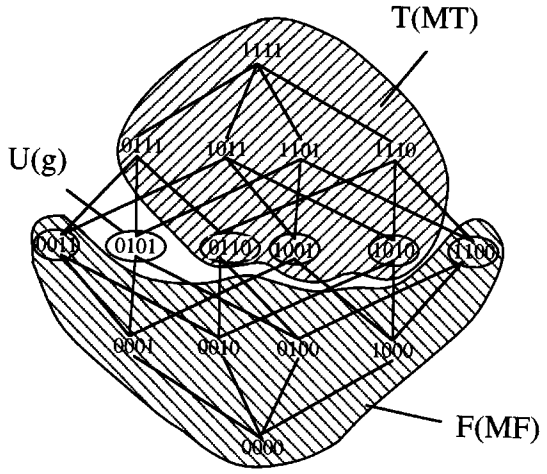


FIG. 3. *The partial function g of Example* 4.12.

The result is illustrated in Figure 3. The latency $\lambda(g)$ is obviously equal to 2.

Next, we consider an upper bound on Λ_{MAT}.

LEMMA 4.13. *Let f be a matroid function. If a partial function g of f defined by MT and MF satisfies $N_2(g) \cap U(g) = \emptyset$, then $MT = \min T(f)$.*

Proof. Note that $MT \cup MF \neq \emptyset$ holds by the definition of a partial function g. First assume $MT = \emptyset$. Then $MF \neq \emptyset$ and we let $b \in MF$. If $OFF(b) = \emptyset$ (i.e., $b = (1, 1, \ldots, 1)$), then $MT = \min T(f) = \emptyset$, which completes the proof. Otherwise, $b + e_j \in T(MT)$ holds for any $j \in OFF(b)$, since $b + e_j \in N_1(g)$ and $N_2(g) \cap U(g) = \emptyset$. This means $MT \neq \emptyset$, a contradiction. Hence in the following, we consider the case of $MT \neq \emptyset$. Assume that there exists a $v \in \min T(f) \setminus MT$. Take an $a^{(1)} \in MT$ and let $k = |ON(a^{(1)}) \setminus ON(v)|$. If $k = 1$, then $\| v - a^{(1)} \| = 2$ since $|ON(v)| = |ON(a^{(1)})| = r$, where r is the rank of f, which is a contradiction. If $k > 1$, there exists $a^{(2)} = a^{(1)} - e_i + e_j \in \min T(f)$ where $i \in ON(a^{(1)}) \setminus ON(v)$ and $j \in ON(v) \setminus ON(a^{(1)})$, since f is a matroid function. Then $\| a^{(2)} - a^{(1)} \| = 2$, and hence $a^{(2)} \in MT$ and $|ON(a^{(2)}) \setminus ON(v)| = k - 1$. By repeating this procedure, we obtain $a^{(1)}, a^{(2)}, \ldots, a^{(k-1)}, a^{(k)} \in MT$ such that $|ON(a^{(k)}) \setminus ON(v)| = 1$, a contradiction. □

After showing the next lemma, we show a sufficient condition to have $MF = \max F(f)$.

LEMMA 4.14. *If f is a matroid function, then for every $a \in \min T(f)$ and every $j \in ON(a)$, there exists exactly one $b \in \max F(f)$ such that $b \geq a - e_j$.*

Proof. See [29], for example. □

LEMMA 4.15. *Let f be a matroid function. If a partial function g of f defined by $MT = \min T(f)$ and some MF satisfies $N_1(g) \cap U(g) = \emptyset$, then $MF = \max F(f)$. That is, $f = g$ (i.e., $\lambda(g) = 0$).*

Proof. Assume that there exists a $v \in \max F(f) \setminus MF$. For any $j \in OFF(v)$, there exists an $a \in MT$ such that $a \leq v + e_j$, and there exists a $b \in MF$ such that $b \geq a - e_j$. Then Lemma 4.14 implies that $v = b$, a contradiction. □

LEMMA 4.16. *If g is a partial function of a matroid function f, then $\lambda(g) \leq 2$.*

Proof. It follows from Lemma 4.13 and Lemma 4.15. □

THEOREM 4.17. *Class C_{MAT} of matroid functions satisfies*

$$\Lambda_{MAT}(n) = \begin{cases} 1, & n = 1, 2, 3, \\ 2, & n \geq 4. \end{cases}$$

Proof. It follows from Lemma 4.11 and Lemma 4.16. □

In fact, it is known [21] that learning $\min T(f)$ of a matroid function f with only a membership oracle can be done in polynomial time in n and $|\min T(f)|$. Since Lemma 4.14 indicates that $|\max F(f)| \leq n|\min T(f)|$, we can show a similar result by our algorithm that uses the maximum latency.

4.5. *k*-tight positive functions.

DEFINITION 4.18. *A positive function f is called* k-tight *for a positive integer k if it satisfies*

$$\max\{\| a-b \| \mid a \in \min T(f), b \in \max F(f) \text{ and } a-e_j \leq b \text{ for some } j \in ON(a)\} \leq k.$$

Let

C_{kTI}: class of k-tight positive functions.

Note that the condition $a - e_j \leq b$ for $a \in \min T(f)$ and $b \in \max F(f)$ is equivalent to the condition $b + e_j \geq a$. Let us imagine $\min T(f)$ and $\max F(f)$ as the bottoms of ravines and the peaks of mountains, respectively. Then k-tightness of f implies that the height difference between a mountain peak and the bottoms of its surrounding ravines is at most k. This means that the ground is locally smooth, even though some mountains may be very high.

For example, a positive threshold function with $w_{\max} \leq k w_{\min}$ is always k-tight, where $w_{\max} = \max_i w_i$ and $w_{\min} = \min_i w_i$, since for any $a \in \min T(f)$, $j \in ON(a)$, and $i_l \in OFF(a)$ $(l = 1, 2, \ldots, k)$,

$$\begin{aligned} \sum_{i=1}^{n} w_i a_i - w_j + \sum_{l=1}^{k} w_{i_l} &\geq \sum w_i a_i - w_{\max} + k w_{\min} \\ &\geq \sum w_i a_i \ \geq \ t, \end{aligned}$$

i.e., $a - e_j + \sum_{l=1}^{k} e_{i_l} \in T(f)$.

To introduce other types of k-tight functions, define the *rank* of a set $S \subseteq \{0,1\}^n$ by $r(S) = \max\{\| x \| \mid x \in S\}$ and the *antirank* by $ar(S) = \min\{\| x \| \mid x \in S\}$, respectively.

THEOREM 4.19. *A positive function f satisfying one of the following conditions is k-tight.*

(i) $|r(\max F(f)) - ar(\min T(f))| \leq k - 2$.
(ii) $ar(\min T(f)) \geq n - k + 1$.
(iii) $r(\max F(f)) \leq k - 1$.

Proof. (i) For any $a \in \min T(f)$ and $b \in \max F(f)$, $| \| b \| - \| a \| | \leq k - 2$. If $a - e_j \leq b$, then $\| b - a \| \leq k$ holds.

Conditions (ii) and (iii) are similar to (i). □

These types of functions are discussed in [13] and other papers.

LEMMA 4.20. *For a positive function f, define an undirected bipartite graph $G_f = (\min T(f), \max F(f), E)$, where $E = \{(a, b) \mid a \in \min T(f), b \in \max F(f)$ and $a - e_j \leq b$ for some $j \in ON(a)\}$. Then this G_f is connected.*

Proof. We proceed by induction on $|\min T(f)|$. For $|\min T(f)| = 0, 1$, it is trivially true. Assume that the lemma holds for all $|\min T(f)| \leq k$ but does not hold for $|\min T(f)| = k+1$, i.e., G_f is not connected. Let $\min T(f) = \{a^{(1)}, a^{(2)}, \ldots, a^{(k)}, a^{(k+1)}\}$, V_1 be the connected component of G_f that includes $a^{(1)}$ and $V_2 = V - V_1$, where V is the set of all nodes in G_f.

First, we show $V_1 \cap \min T(f) \neq \{a^{(1)}\}$. Take a $b \in \max F(f)$ such that $b \geq a^{(1)} - e_j$ for $j \in ON(a^{(1)})$. Then $b_j = 0$, since otherwise $b \geq a^{(1)}$, a contradiction, and hence $\| b \| \leq n-1$. If $\| b \| = n-1$, then $b + e_j = (1, 1, \ldots, 1)$ and $a^{(i)} \leq b + e_j$ for all $a^{(i)} \in \min T(f)$, i.e., $V_1 \cap \min T(f) = \min T(f)$. Otherwise, i.e., $\| b \| < n-1$, there exists an $a' \in \min T(f)$ such that $a' \leq b + e_i$ for some $i \in OFF(b) \setminus \{j\}$. Then $a'_j = 0$ by $a' \leq b + e_i$ and $b_j = 0$. Therefore, $a' \neq a^{(1)}$ and $\{a^{(1)}, a'\} \subseteq V_1 \cap \min T(f)$.

Second, we show $V_2 \cap \min T(f) \neq \emptyset$. If $V_2 \cap \min T(f) = \emptyset$ (i.e., $\min T(f) \subseteq V_1$), then $\max F(f) \subseteq V_1$ holds since for any $b \in \max F(f)$, there exist $j \in OFF(b)$ and $a \in \min T(f)$ such that $a \leq b + e_j$. This contradicts the assumption that G_f is not connected.

Now, define f' by $\min T(f') = \{a^{(2)}, a^{(3)}, \ldots, a^{(k+1)}\}$, and let

$$S = \{b \in \max F(f') \mid b \geq a^{(1)}\}.$$

Then

$$\begin{aligned} \min T(f') \cup \{a^{(1)}\} &= \min T(f), \\ \max F(f') \setminus S &\subseteq \max F(f), \end{aligned}$$

and hence

$$S = (\min T(f') \cup \max F(f')) \setminus (\min T(f) \cup \max F(f)). \tag{4}$$

There is no path from each $a^{(p)} \in V_1 \cap (\min T(f) \setminus \{a^{(1)}\})$ to any $a^{(q)} \in V_2 \cap \min T(f)$ in G_f. However, by the induction hypothesis, these $a^{(p)}$ and $a^{(q)}$ are connected in $G_{f'}$. That is, by property (4), there are $a^{(p)} \in V_1 \cap (\min T(f) \setminus \{a^{(1)}\})$, $a^{(q)} \in V_2 \cap \min T(f)$, and $c \in S$ such that c is adjacent to $a^{(p)}$ and $a^{(q)}$ in $G_{f'}$. We then show that there is a $b \in \max F(f)$ such that b is adjacent to $a^{(1)}$ and $a^{(q)}$ in G_f, which leads to a contradiction. Take a $j \in ON(a^{(1)}) \setminus ON(a^{(q)})$ arbitrarily. Then $c - e_j \in F(f)$ since $\min T(f) \cap \{v \mid v \leq c\} = \{a^{(1)}\}$. Therefore, there exists a $b \in \max F(f)$ such that $b \geq c - e_j$. Since $a^{(1)} - e_j \leq c - e_j \leq b$, this b is adjacent to $a^{(1)}$ in G_f. Moreover, $a^{(q)} - e_i \leq c$ for some $i \in ON(a^{(q)})$, since c is adjacent to $a^{(q)}$ on $G_{f'}$. Hence, $a^{(q)} - e_i \leq c - e_j$ by $a^{(q)}_j = 0$. Therefore, $a^{(q)} - e_i \leq b$, i.e., b is adjacent to $a^{(q)}$. ∎

THEOREM 4.21. *Class C_{kTI} of k-tight positive functions satisfies*

$$\Lambda_{kTI}(n) \leq k.$$

Proof. Assuming that g is a partial function of $f \in C_{kTI}$, defined by MT and MF, such that $N_k(g) \cap U(g) = \emptyset$ and $U(g) \neq \emptyset$, we derive a contradiction. Since $U(g) \neq \emptyset$, there exists a $u \in (\min T(f) \cup \max F(f)) \cap U(g)$. We assume $u \in \min T(f) \cap U(g)$ without loss of generality, and let $T_1 = \{u\}$ and $F_1 = \emptyset$. If there exists some $j \in ON(u)$ such that $u - e_j \in F(MF)$, then there exists a $b \in MF$ such that $b \geq u - e_j$. $\| u - b \| \leq k$ holds since f is k-tight, and hence $u \in N_k(g) \cap U(g)$, which is a contradiction. Therefore, $u - e_j \in U(g)$ holds for all $j \in ON(u)$; i.e., $F_2 = \{b \in \max F(f) \mid b \geq u - e_j$ for some $u \in T_1$ and $j \in ON(u)\}$ satisfies $F_2 \subseteq U(g)$. If $b + e_i \in T(MT)$ for some $b \in F_2$ and $i \in OFF(b)$, then some $a \in MT$ satisfies $a \leq b + e_j$

and $\| a - b \| \leq k$, i.e., $b \in N_k(g) \cap U(g)$, a contradiction. Therefore, $T_2 \subseteq U(g)$, where $T_2 = \{a \in \min T(f) | a \leq u + e_j$ for some $u \in F_2$ and $j \in OFF(u)\}$. We can indefinitely continue this procedure, which generates $T_1 \subseteq T_2 \subseteq \cdots \subseteq U(g)$ and $F_1 \subseteq F_2 \subseteq \cdots \subseteq U(g)$, where F_{i+1} ($\subseteq \max F(f)$) is the set of vertices adjacent to T_i in G_f, and T_{i+1} ($\subseteq \min T(f)$) is the set of vertices adjacent to F_{i+1} in G_f. By Lemma 4.20, we obtain $T_l = \min T(f)$ and $F_l = \max F(f)$ for a sufficient large l. This is a contradiction to $MT \cup MF \neq \emptyset$. □

COROLLARY 4.22. (i) *For the class of positive functions f satisfying $|r(\max F(f)) - ar(\min T(f))| \leq k$, its maximum latency is not greater than $k + 2$.*

(ii) *For the class of positive functions f satisfying $ar(\min T(f)) \geq n - k$, its maximum latency is not greater than $k + 1$.*

(iii) *For the class of positive functions f satisfying $r(\max F(f)) \leq k$, its maximum latency is not greater than $k + 1$.*

Proof. Combine Theorems 4.19 and 4.21. □

5. Positive *k*-DNF functions.

DEFINITION 5.1. *A positive function f which satisfies*

$$r(\min T(f)) \leq k$$

is called a positive k-DNF function*, where DNF stands for disjunctive normal form.*

Let

$$C_{kDNF}: \quad \text{class of positive } k\text{-DNF functions.}$$

The next theorem shows that the maximum latency cannot be constant for this class, though there is an incrementally polynomial identification algorithm [13].

THEOREM 5.2. *Class C_{kDNF} satisfies*

$$\Lambda_{kDNF}(n) \geq \begin{cases} 1, & k = 1, \\ (k-1)\left\lfloor \sqrt{\frac{n}{(k-1)}} \right\rfloor - k + 2, & k \geq 2, \end{cases}$$

for $n \geq 4(k-1)$.

Proof. For $k = 1$, it is clear that $\Lambda_{1DNF}(n) \geq 1$. For $k \geq 2$, we provide an example of g with $\lambda(g) = (k-1)\lfloor \sqrt{\frac{n}{(k-1)}} \rfloor - k + 2$ for $n \geq 4(k-1)$. Let $\alpha = \lfloor \sqrt{\frac{n}{(k-1)}} \rfloor$, where α satisfies $\alpha \geq 2$. Then

$$\sqrt{\frac{n}{(k-1)}} \geq \alpha, \quad \text{i.e., } n \geq \alpha^2(k-1).$$

Therefore, let $n = \alpha^2(k-1) + \beta$, where β is a nonnegative integer. Now define a $\alpha(\alpha-1)(k-1) \times n$ matrix

$$X = \left(\begin{array}{ccc} Q & \left| I_{\alpha(\alpha-1)(k-1)} \right| & O_{\alpha(\alpha-1)(k-1)\times\beta} \end{array} \right),$$

where Q is the $\alpha(\alpha-1)(k-1) \times \alpha(k-1)$ matrix

$$Q = \begin{array}{c} \overbrace{}^{k-1}\overbrace{}^{k-1}\overbrace{}^{k-1}\quad\overbrace{}^{k-1} \\ \left(\begin{array}{cccc} 11\cdots1 & 00\cdots0 & \cdots & 00\cdots0 \\ 11\cdots1 & 00\cdots0 & \cdots & 00\cdots0 \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ 11\cdots1 & 00\cdots0 & \cdots & 00\cdots0 \\ \hline 00\cdots0 & 11\cdots1 & \cdots & 00\cdots0 \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ 00\cdots0 & 11\cdots1 & \cdots & 00\cdots0 \\ \hline \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ \hline 00\cdots0 & 00\cdots0 & \cdots & 11\cdots1 \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ 00\cdots0 & 00\cdots0 & \cdots & 11\cdots1 \end{array}\right) \end{array} \begin{array}{l} \left.\phantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} (\alpha-1)(k-1) \\ \left.\phantom{\begin{array}{c}a\\a\\a\end{array}}\right\} (\alpha-1)(k-1), \\ \vdots \\ \left.\phantom{\begin{array}{c}a\\a\\a\end{array}}\right\} (\alpha-1)(k-1) \end{array}$$

I_j is the $j \times j$ identity matrix, and $O_{i\times j}$ is the $i \times j$ zero matrix. Define f by

$$\min T(f) = (\text{the set of rows of matrix } X)$$

and a partial function g of f by $MT = \min T(f)$ and $MF = \max F(f) \setminus \{u\}$, where

$$u = (\underbrace{1,1,\ldots,1}_{\alpha(k-1)}, \underbrace{0,0,\ldots,0}_{\alpha(\alpha-1)(k-1)}, \underbrace{1,1,\ldots,1}_{\beta}).$$

Then $u + e_j \in T(MT)$ for any $j \in OFF(u)$ since $u \in \max F(f)$ and $MT = \min T(f)$. Moreover, $u - e_j \in F(MF)$ for any $j \in \{1,2,\ldots,\alpha(k-1)\}$ and $u - \sum_{j\in S} e_j \notin F(MF)$, where $S = \{n-\beta+1, n-\beta+2, \ldots, n\}$. In other words,

$$U(g) = \{\ (\underbrace{1,1,\ldots,1}_{\alpha(k-1)}, \underbrace{0,0,\ldots,0}_{\alpha(\alpha-1)(k-1)}, \underbrace{*,*,\ldots,*}_{\beta})\ \},$$

where $*$ stands for 0 or 1. It is not difficult to see that $\| a - w \| \geq (\alpha-1)(k-1)+1$ for every $a \in MT$ and $w \in U(g)$ and $\| b - w \| \geq (\alpha-1)(k-1)+1$ for every $b \in MF$ and $w \in U(g)$. Therefore, its latency is

$$\lambda(g) = (\alpha-1)(k-1)+1 = (k-1)\left\lfloor \sqrt{\frac{n}{(k-1)}} \right\rfloor - k + 2. \qquad \square$$

We note that the f and g of Example 2.2 satisfy the above conditions with $n = 4$ and $k = 2$.

6. Conclusion. The problem of identifying a positive Boolean function f has an incrementally polynomial algorithm if and only if problem EQ can be solved in polynomial time [6]. In this paper, we introduced the maximum latency as a measure for the difficulty to find an unknown vector. If the maximum latency is constant, then EQ can be solved in polynomial time. It turned out that the maximum latency of general positive functions is $\lfloor n/4 \rfloor + 1$, but several subclasses of positive functions have constant maximum latency. Such subclasses include classes of (i) 2-monotonic positive functions, (ii) Δ-partial positive threshold functions, (iii) k-degree positive threshold

functions, (iv) matroid functions, and (v) k-tight positive functions. Finally, it is shown that the class of positive k-DNF functions has maximum latency not less than $O(\sqrt{n})$, even though it is known [13] that problem EQ can be solved in polynomial time for this class of functions.

The last result indicates that the concept of maximum latency is not always sufficient to distinguish polynomially solvable cases from those not solvable in polynomial time. However, it is also evident that the maximum latency is a powerful tool to find polynomially solvable special cases.

Acknowledgments. The authors are grateful to the valuable comments given by H. Nagamochi and the other members of their laboratory, and to the information on [20] provided by T. Hegedus of Comenius University, Slovakia. The authors also appreciate the comments given by two anonymous reviewers, which helped improve the readability of this paper.

REFERENCES

[1] D. Angluin, *Queries and concept learning*, Machine Learning, 2 (1988), pp. 319–342.
[2] M. Anthony and N. Biggs, *Computational Learning Theory*, Cambridge University Press, London, 1992.
[3] M. O. Ball and J. S. Provan, *Disjoint products and efficient computation of reliability*, Oper. Res., 36 (1988), pp. 703–715.
[4] P. Bertolazzi and A. Sassano, *An $O(mn)$ time algorithm for regular set-covering problems*, Theoret. Comput. Sci., 54 (1987), pp. 237–247.
[5] J. C. Bioch and T. Ibaraki, *Decompositions of positive self-dual Boolean functions*, Discrete Math., 140 (1995), pp. 23–46.
[6] J. C. Bioch and T. Ibaraki, *Complexity of identification and dualization of positive Boolean functions*, Inform. and Comput., 123 (1995), pp. 50–63.
[7] E. Boros, P. L. Hammer, T. Ibaraki, and K. Kawakami, *Identifying 2-monotonic positive Boolean functions in polynomial time*, in ISA'91 Algorithms, W. L. Hsu and R. C. T. Lee, eds., Springer Lecture Notes in Computer Science 557, Springer-Verlag, Berlin, 1991, pp. 104–115.
[8] E. Boros, P. L. Hammer, T. Ibaraki, and K. Kawakami, *Polynomial time recognition of 2-monotonic positive boolean functions given by an oracle*, SIAM J. Comput., 26 (1997), pp. 93–109.
[9] N. H. Bshouty, *Exact learning via the monotone theory*, Inform. and Comput., 123 (1995), pp. 146–153.
[10] C. J. Colbourn, *The Combinatorics of Network Reliability*, Oxford University Press, London, 1987.
[11] Y. Crama, *Dualization of regular Boolean functions*, Discrete Appl. Math., 16 (1987), pp. 79–85.
[12] Y. Crama, P. L. Hammer, and T. Ibaraki, *Cause-effect relationships and partially defined Boolean functions*, Ann. Oper. Res., 16 (1988), pp. 299–326.
[13] T. Eiter and G. Gottlob, *Identifying the minimal transversals of a hypergraph and related problems*, SIAM J. Comput., 24 (1995), pp. 1278–1304.
[14] M. Fredman and L. Khachiyan, *On the complexity of dualization of monotone disjunctive normal forms*, J. Algorithms, 21 (1996), pp. 618–628.
[15] D. N. Gainanov, *On one criterion of the optimality of an algorithm for evaluating monotonic Boolean functions*, USSR Comput. Math. Math. Phys., 24 (1984), pp. 176–181.
[16] H. Garcia-Molina and D. Barbara, *How to assign votes in a distributed system*, J. ACM, 32 (1985), pp. 841–860.
[17] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman, New York, 1979.
[18] T. Ibaraki and T. Kameda, *A theory of coteries: Mutual exclusion in distributed systems*, IEEE Trans. Parallel Distrib. Systems, 4 (1993), pp. 779–794.
[19] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *On generating all maximal independent sets*, Inform. Process. Lett., 27 (1988), pp. 119–123.
[20] V. K. Korobkov and T. L. Reznik, *Certain algorithms for the computation of monotonic functions in the algebra of logic*, Soviet Math. Dokl., 3 (1962), pp. 1763–1767.

[21] E. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, *Generating all maximal independent sets: NP-hardness and polynomial-time algorithms*, SIAM J. Comput., 9 (1980), pp. 558–565.

[22] K. Makino and T. Ibaraki, *The maximum latency of partially defined positive Boolean functions*, Trans. IEICE, J76-D-1 (1993), pp. 409–416. (In Japanese.)

[23] K. Makino and T. Ibaraki, *The maximum latency and identification of positive Boolean functions*, in Proceedings 5th International Symposium on Symbolic and Algebraic Computation (ISSAC'94), D. Z. Du and X. S. Zhang, eds., Lecture Notes in Computer Science 834, Springer-Verlag, Berlin, 1994, pp. 324–332.

[24] K. Makino and T. Ibaraki, *A fast and simple algorithm for identifying* 2*-monotonic positive Boolean functions*, in Proceedings 6th International Symposium on Symbolic and Algebraic Computation (ISSAC'95), J. Staples et al., eds., Lecture Notes in Computer Science 1004, Springer-Verlag, Berlin, 1995, pp. 291–300.

[25] S. Muroga, *Threshold Logic and Its Applications*, Wiley-Interscience, New York, 1971.

[26] U. N. Peled and B. Simeone, *Polynomial-time algorithm for regular set-covering and threshold synthesis*, Discrete Appl. Math., 12 (1985), pp. 57–69.

[27] R. Reiter, *A theory of diagnosis from first principles*, Artif. Intell., 32 (1987), pp. 57–95.

[28] L. G. Valiant, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[29] D. J. A. Welsh, *Matroid Theory*, Academic Press, New York, 1976.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1384–1408, October 1997

006

AN EXPANDER-BASED APPROACH TO GEOMETRIC OPTIMIZATION*

MATTHEW J. KATZ† AND MICHA SHARIR‡

Abstract. We present a new approach to problems in geometric optimization that are traditionally solved using the parametric-searching technique of Megiddo [*J. ACM*, 30 (1983), pp. 852–865]. Our new approach is based on expander graphs and range-searching techniques. It is conceptually simpler, has more explicit geometric flavor, and does not require parallelization or randomization. In certain cases, our approach yields algorithms that are asymptotically faster than those currently known (e.g., the second and third problems below) by incorporating into our (basic) technique a subtechnique that is equivalent to (though much more flexible than) Cole's technique for speeding up parametric searching [*J. ACM*, 34 (1987), pp. 200–208]. We exemplify the technique on three main problems—the *slope selection* problem, the planar *distance selection* problem, and the planar *two-line center* problem. For the first problem we develop an $O(n \log^3 n)$ solution, which, although suboptimal, is very simple. The other two problems are more typical examples of our approach. Our solutions have running time $O(n^{4/3} \log^2 n)$ and $O(n^2 \log^4 n)$, respectively, slightly better than the previous respective solutions of [Agarwal et al., *Algorithmica*, 9 (1993), pp. 495–514], [Agarwal and Sharir, *Algorithmica*, 11 (1994), pp. 185–195]. We also briefly mention two other problems that can be solved efficiently by our technique.

In solving these problems, we also obtain some auxiliary results concerning batched range searching, where the ranges are congruent discs or annuli. For example, we show that it is possible to compute deterministically a compact representation of the set of all point-disc incidences among a set of n congruent discs and a set of m points in the plane in time $O((m^{2/3}n^{2/3} + m + n) \log n)$, again slightly better than what was previously known.

Key words. expander graphs, geometric optimization, slope selection, computational geometry, parametric searching, range searching, facility location

AMS subject classifications. 05C99, 68Q20, 68R10, 52C99, 90B80

PII. S0097539794268649

1. Introduction.

More than 10 years ago, Megiddo [34] proposed the ingenious technique of *parametric searching* for efficiently solving a variety of optimization problems. The technique has recently been applied to many problems in geometric optimization, thus demonstrating its power and versatility (see [2, 3, 4, 6, 7, 12, 18, 20, 37] for some of these applications).

The general idea behind parametric searching is that we are given a "decision problem" $P(\lambda)$ that depends on a real parameter λ, and we seek an optimal value λ^* of λ, where a certain extremal condition is satisfied. For example, in the *slope selection* problem, we are given n lines in the plane and an integer k, and we want to find the kth leftmost vertex of the arrangement of the lines. The corresponding decision problem $P(\lambda)$ is to determine whether the number of vertices of the arrangement to the left of or on the vertical line $x = \lambda$ is smaller than, equal to, or larger than k, and the

* Received by the editors May 27, 1994; accepted for publication (in revised form) October 13, 1995. This research was supported by a Fund for Basic Research grant administered by the Israeli Academy of Sciences. The research of the second author was supported by NSF grant CCR-91-22103 and by grants from the U.S.–Israeli Binational Science Foundation and the G. I. F., the German–Israeli Foundation for Scientific Research and Development.

http://www.siam.org/journals/sicomp/26-5/26864.html

† Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel (matya@cs.bgu.ac.il).

‡ School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel (sharir@math.tau.ac.il) and Courant Institute of Mathematical Sciences, New York University.

optimal λ^* is the abscissa of the vertical line that passes through the kth leftmost vertex.

We assume that an "oracle" procedure $A(\lambda)$ that solves the decision problem is available; that is, it can determine whether a given value of λ is smaller than, equal to, or larger than the desired optimal λ^*. Our goal is to run some sort of binary search over λ, using $A(\lambda)$ as a discriminating procedure, until λ^* is found. The problem is that the number of critical values of λ where the output of $A(\lambda)$ might change is often too large, so we cannot afford to generate all of them. (In the slope selection problem, these critical values are the abscissae of the vertices of the arrangement, so there are quadratically many such values. The oracle for this problem simply has to count the number of vertices lying to the left of a vertical line $x = \lambda$, which can easily be done in time $O(n \log n)$ using a standard inversion-counting procedure; see, e.g., [18].)

The parametric-searching technique generates, in an implicit manner, only a small subset of these critical values which is guaranteed to include λ^*. It does so by running a generic parallel version of the oracle $A(\lambda)$; that is, it runs such a parallel procedure without knowing the value of λ, with the intention of simulating its execution at the unknown λ^*. Whenever it needs to resolve, in a single parallel step, a batch of comparisons whose outcome depends on λ, it computes the critical values where any of these comparisons changes its outcome and runs a binary search to locate λ^* among these values, using the (explicit) oracle $A(\lambda)$. This determines the outcome of all comparisons of the current batch, and the next parallel step can be executed. Each parallel step generates further restrictions on the range where λ^* must lie, and at the end of execution this range must essentially shrink to a singleton, thus yielding λ^*. The parallel version of the generic algorithm ensures that the algorithm executes only a small number of (expensive) oracle calls. We refer the reader to [7, 34, 37] for more detailed expositions of the method.

In spite of its power and versatility, parametric searching suffers from several disadvantages. First, it relies crucially on the availability of a parallel version of the oracle procedure, which is not always easy to obtain. Admittedly, this parallel version is simulated sequentially, so we can develop it in the rather loose comparison-based model of Valiant [38], but still it is often a major hurdle in the application of parametric searching. In addition, the parallel algorithm is often much more complicated than its sequential counterpart, and the need to run it generically tends to make its implementation extremely complicated. Second, the behavior of parametric searching is rather mysterious and highly nonintuitive. On one hand, this is where the beauty of the method lies, but it also makes the method difficult to follow, and makes the execution of the algorithm appear to be rather erratic, performing comparisons that are guided by some mysterious rule and that by magic manage to converge on the right value of λ^*. This feeling is more acute in geometric applications of parametric searching, where one would like to have a geometric interpretation that will provide a more explicit explanation and prediction of the behavior of the algorithm, and where such an interpretation is often lacking.

Recently, Chazelle et al. [12] proposed an alternative approach to parametric searching in geometric optimization. Their approach is based on epsilon nets and on recent related partitioning techniques (called *cuttings*). They applied it to the slope selection problem and obtained an $O(n \log^2 n)$ (deterministic) solution. Their solution is conceptually much simpler than the previous optimal $O(n \log n)$ solution of Cole et al. [18]. Moreover, it can be made optimal by replacing the oracle calls by calls to the approximating oracle of [18] (this observation was missed in [12]). It seems that

the approach of [12] can be applied to other geometric problems, although this still remains to be worked out.

Another alternative approach to parametric searching was recently proposed by Dillencourt, Mount, and Netanyahu [19] and by Matoušek [28, 29]. This approach is based on randomization and yields efficient and fairly simple algorithms. For example, in the slope selection problem, it generates a small number of random critical values of λ and locates λ^* among these values. Then it generates a new set of random values of λ from the restricted range where λ^* is now known to lie and again locates λ^* among these new values, and continues in this manner until λ^* is found. In the case of slope selection, the algorithm of [28] has expected running time $O(n \log n)$. Matoušek comments in [28] on possible extensions of his technique to other geometric problems, such as those studied in this paper.

In this paper we propose a different alternative approach to parametric searching in geometric optimization and demonstrate its applicability to the slope selection problem; to the *distance selection* problem, where we are given n points in the plane and an integer k, and we want to find the kth smallest distance between the given points; and to the *two-line center* problem, where we are given n points in the plane and want to find the smallest width r so that all the points can be covered by the union of two strips of width r. Agarwal et al. [2] present a randomized solution to the distance selection problem, based on parametric searching, that runs in expected time $O(n^{4/3} \log^{8/3} n)$. Their solution can be made deterministic using more involved techniques [1]. Another deterministic solution, which also runs in time $O(n^{4/3} \log^{8/3} n)$, was recently given by Goodrich [21]. Here we give an improved $O(n^{4/3} \log^2 n)$ solution. Agarwal and Sharir [6] present a deterministic $O(n^2 \log^5 n)$ algorithm for the two-line center problem, which is based on parametric searching. Here we give an improved $O(n^2 \log^4 n)$ algorithm for this problem.[1] We also briefly mention two more problems for which our approach yields efficient solutions. These problems are as follows.

(i) The *two-center* problem: Given n points in the plane, find the smallest radius r so that all the points can be covered by the union of two discs of radius r.

(ii) The *minimum output rate* problem: Given consecutive time intervals $T_1, \ldots, T_n$ and input rates $I_1, \ldots, I_n$ (where I_j is the input rate during the jth time interval) and a buffer size B, find the minimum output rate R^* required to assure that the buffer does not overflow; see [35] for more details.

We present an $O(n^2 \log^3 n)$ solution to the first problem; the same asymptotic running time was achieved by Agarwal and Sharir [6] using parametric searching. The second problem was solved by Megiddo, Naor, and Anderson [35], who gave an $O(n \log n \log\log n)$ deterministic solution and a randomized solution with expected running time $O(n \log n)$, both using parametric searching. We give a geometric interpretation of the problem and derive a deterministic $O(n \log^3 n)$ solution and a randomized solution (which is based on a different approach) with expected running time $O(n \log n)$, which we believe to be simpler than the solution of [35].

[1] There have been recent developments, after the original submission of this paper, in regard to the solution of the two-line center problem. Glozman, Kedem, and Shpitalnik [*On some geometric selection and optimization problems via sorted matrices*, Proc. Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 995, Springer-Verlag, Berlin, 1995, pp. 26–37] give another algorithm with running time $O(n^2 \log^4 n)$, and Jaromczyk and Kowaluk [*The two-line center problem from a polar view: A new algorithm and data structure*, Proc. Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 955, Springer-Verlag, Berlin, 1995, pp. 13–25] give an improved $O(n^2 \log^2 n)$ solution.

Our approach is based on *expander graphs* that are constructed on certain subsets of the input objects. Since expanders can be constructed in an explicit deterministic manner (see, e.g., [9]), our technique is deterministic. It is conceptually rather simple, does not require parallelization, and has a very clear geometric interpretation. In addition, it avoids performing some of the generic comparisons that are performed in standard parametric-searching applications (such as [2]), which change their outcome at roots of rather high-degree polynomials. Its analysis relies on a simple and known property of expanders, which loosely states that for any pair of sufficiently large sets of vertices, the expander contains sufficiently many edges between them. There are only a few applications of expanders to geometric problems, such as the recent application of expanders to parallel linear programming [8]. We hope that our study will lead to further geometric applications of expanders.

As an exemplification of our technique, the slope selection problem is perhaps not ideal, in the sense that (what we regard as) the elegant and simple algorithm that is given below is suboptimal and runs in time $O(n \log^3 n)$ (which is, by the way, the same running time yielded by the basic parametric-searching method, without the improvements of [17] and of [18]). To improve it, one must develop additional technical tricks that have little to do with the basic method. (In a companion paper [25], we do present an alternative $O(n \log n)$ algorithm, which is also based on expanders.)

The solution to the distance selection problem is a much more typical exemplification of our technique and also demonstrates the general applicability of the technique. Roughly speaking, the algorithm runs in $O(\log n)$ stages. Each stage produces a further restricted range where the kth smallest distance is known to lie, with the additional property that the jth range I_j contains at most $O(n^2\rho^j)$ distances among the given points for some constant $\rho < 1$. The jth stage begins by generating a compact representation of the set of all pairs of points whose distances lie in I_{j-1}. This is done by performing batched range searching with appropriate annuli centered at the given points. This yields a collection of complete bipartite graphs $\{M_t \times P_t\}_t$, where for each t the distance between every point in M_t and every point in P_t lies in I_{j-1}. Next we replace each such bipartite graph by an expander G_t. We associate with each edge of these expanders the distance between the two points that it connects. Since the total number of edges of these graphs is relatively small, we can afford to run a binary search (using the oracle) to locate the desired kth smallest distance in the list of the distances associated with these edges. This search produces the next interval I_j and the whole procedure is repeated.

The analysis of the algorithm, namely, the proof that the numbers of distances in the ranges I_j keep decreasing geometrically, is somewhat intricate and relies on the property of expanders mentioned above and on certain properties of range searching via geometric partitioning.

We enhance our basic technique (as described above) by a subtechnique. The enhanced technique yields, in certain cases, algorithms that are asymptotically faster that those previously known (e.g., our algorithms for the distance selection problem and for the two-line center problem). The subtechnique is in some sense equivalent to Cole's technique for speeding up standard parametric searching [17], although the context in which it is applied is rather different (and considerably more general). In both approaches, the number of (expensive) oracle calls is reduced, so that only a constant (rather than logarithmic) number of calls are made in each stage of the algorithm. For Cole's improvement to apply, the generic parallel algorithm must run on a sorting network (or satisfy similar restrictive conditions), whereas our improvement

is much more flexible and can be applied in fairly general settings.

We also obtain some results concerning *batched range searching*, which we use in the algorithm for the distance selection problem. These results, we believe, are of independent interest. Loosely speaking, each stage of our algorithms begins with a range I where λ^* is known to lie and aims to shrink this range further. To do so, the algorithm performs range searching to obtain, in compact form, all pairs (or triples, etc.) of the data objects, whose "comparisons" generate critical values of λ that fall in I. Once all these interactions have been found, we replace them by certain expander graphs and use only the critical values generated by the edges of those graphs for the oracle-guided binary search. In this setting all the relevant range-searching queries are known in advance, so we can perform all of them using *batched range searching*; this technique, when carefully implemented, can perform better than standard range searching. For example, given n congruent discs and m points in the plane, we compute a compact representation of the set of all point-disc incidences in time $O((m^{2/3}n^{2/3} + m + n)\log n)$, slightly better than what was previously known.

The paper is organized as follows. In section 2 we briefly review expander graphs and their basic properties. In section 3 we obtain some results concerning batched range searching. (The contents of this section are not directly related to the main topic of this paper, so this section may be skipped in a first reading, if so desired.) Sections 4, 5, and 6 present, respectively, our solutions to the distance selection problem, the two-line center problem, and the slope selection problem. In section 7 we briefly mention two more problems for which our approach yields efficient algorithms. We conclude the paper in section 8 with a discussion of the potential of our technique and with some open problems and suggestions for further research.

2. Expanders and their properties.

DEFINITION 2.1. *A graph $G = (V, E)$ is an (n, d, c) expander if it has n vertices, its degree is d, and for every set of vertices $W \subset V$ of cardinality $|W| \leq n/2$, $|N(W)| \geq c|W|$, where $N(W)$ is the set of vertices in $V \setminus W$ that are connected to W by an edge of G.*

The following property is proved in [9, Chap. 9, Corollary 2.2].

LEMMA 2.2. *If G is a d-regular graph with n vertices and λ is the second largest eigenvalue of the adjacency matrix of G, then G is an (n, d, c) expander with $c = (d - \lambda)/2d$.*

Thus, if λ is much smaller than d (which is the largest eigenvalue of the adjacency matrix of G), then G is a good expander.

Lubotzky, Phillips, and Sarnak [26] (and independently Margulis [27]) have given an explicit and very simple description of a d-regular graph G with n vertices, for which $\lambda \leq 2\sqrt{d-1}$, for any $d = p + 1$ and $n = q + 1$, where p and q are primes congruent to 1 modulo 4. These graphs actually have the stronger property that all their eigenvalues (except d) have *absolute* value at most $2\sqrt{d-1}$. We will refer to such graphs as *LPS-expanders*.

From the description in [26] it follows that, whenever d is a constant, an LPS-expander of degree d with n vertices can be constructed deterministically in $O(n)$ time.

The following lemma, which is the main property of LPS-expanders that we will need in this paper, is proved in [9, Chap. 9, Corollary 2.5].

LEMMA 2.3. *Let $G = (V, E)$ be a d-regular graph with n vertices. Assume the absolute value of all its eigenvalues, but the largest is at most λ. Then, for every two sets of vertices A and B of respective cardinalities a and b, we have $|e(A, B) - abd/n| \leq$*

$\lambda\sqrt{ab}$, *where* $e(A, B)$ *is the number of edges of* G *connecting a vertex of* A *with a vertex of* B.

COROLLARY 2.4. *If* A *and* B *are two sets of vertices of respective cardinalities* a *and* b *such that* $e(A, B) = 0$, *then* $ab \leq 4n^2/d$.

Proof. The previous lemma and the fact that $\lambda < 2\sqrt{d}$ imply that if $ab > 4n^2/d$, then $e(A, B) \geq abd/n - 2\sqrt{abd} > 0$, as is easily verified. This contradiction completes the proof. □

COROLLARY 2.5. *If* A *and* B *are two sets of vertices of respective cardinalities* a *and* b *such that* $e(A, B) < 3n$, *then* $ab < 9n^2/d$.

Proof. The previous lemma and the fact that $\lambda < 2\sqrt{d}$ imply that if $ab \geq 9n^2/d$, then $e(A, B) \geq abd/n - 2\sqrt{abd} \geq 3n$, as is easily verified. □

3. Batched range searching. We next describe how to perform efficiently the batched range searching required in step 1 of the distance selection algorithm presented in section 4. For simplicity of presentation, we describe the method for the case where the ranges are congruent discs rather than annuli, but the technique works equally well for congruent annuli. The technique is reminiscent of the approach of Clarkson et al. [15]; it is a result of independent interest, and we hope that it will also find other applications.

Let $C = \{c_1, \ldots, c_n\}$ be a set of n congruent circles, and let $P = \{p_1, \ldots, p_m\}$ be a set of m points in the plane. We wish to compute a compact representation of the set Z of all pairs of the form (c_i, p_j), where $c_i \in C$, $p_j \in P$, and p_j lies inside c_i. The representation we are interested in is a collection of complete pairwise edge-disjoint bipartite subgraphs of $C \times P$, that is, pairs of the form (C_t, P_t), where $C_t \subseteq C$, $P_t \subseteq P$, the sets $C_t \times P_t$ are pairwise disjoint, and $Z = \bigcup_t C_t \times P_t$.

We first consider the following problem. Let C be a set of n circles in the plane. We wish to preprocess C so that, given a query point p, the set of circles of C containing p in their interior can be found quickly. The following theorem is due to Agarwal. Here the circles of C do not have to be congruent.

THEOREM 3.1. *Let* C *be a set of* n *circles in the plane. One can construct, in* $O(n^2 \log n)$ *time, a data structure that uses* $O(n^2 \log n)$ *space, so that the set of circles containing a query point in their interior can be reported in* $O(\log n)$ *time as a collection of* $O(\log n)$ *pairwise disjoint "canonical" (prestored) sets.*

This data structure is constructed as follows. Form the arrangement $\mathcal{A}(C)$ of the circles in C, and let G be the dual graph of $\mathcal{A}(C)$; the nodes of G are the faces of $\mathcal{A}(C)$ and its edges connect pairs of faces adjacent along an edge of $\mathcal{A}(C)$. Compute a spanning tree of G and duplicate each edge of the tree to obtain an Eulerian tour π of G; the tour may visit a face of $\mathcal{A}(C)$ any number of times, but the overall number of its edges is clearly $O(n^2)$ (see Figure 1). Now fix a circle $c \in C$ and mark all the edges of π that cross c (i.e., are dual to an edge of $\mathcal{A}(C)$ contained in c). There are only $O(n)$ such edges in π since there are only $O(n)$ edges in $\mathcal{A}(C)$ that are contained in c and each of them is crossed by π at most twice (once in each direction). These (marked) edges partition π into subsequences, so that the union of cells of $\mathcal{A}(C)$ in a single subsequence is either fully contained inside c or lies fully outside c. The total number of such subsequences, over all circles in C, is only $O(n^2)$. We now build a balanced segment tree T over the nodes of π, in their order along π, and represent each circle c by the collection of sequences of faces of $\mathcal{A}(C)$ that are interior to c, as just constructed, viewing each such subsequence as a "segment" of π. We store each of these "segments," for each circle $c \in C$, at $O(\log n)$ nodes of T in a standard fashion (see Figure 2). The desired data structure consists of the resulting segment tree T

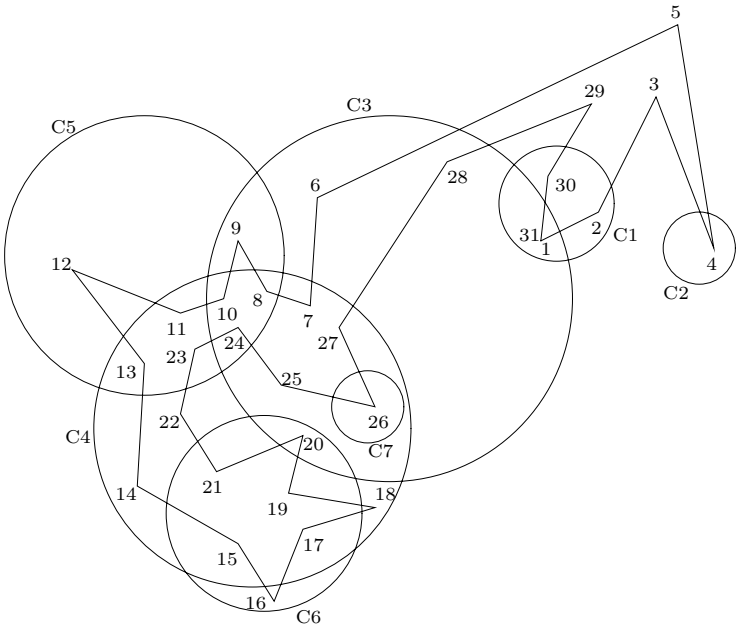


FIG. 1. *An Eulerian path in an arrangement of circles.*

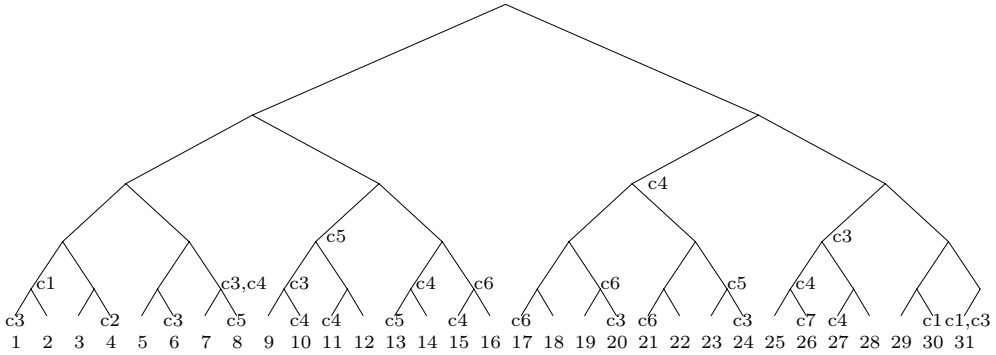


FIG. 2. *The corresponding segment tree.*

plus an efficient point location structure for $\mathcal{A}(C)$, where, in addition, each face of $\mathcal{A}(C)$ points to some leaf of T (that is, to a node of π) that is equal to it. It is easily checked that the data structure requires $O(n^2 \log n)$ space and can be constructed in time $O(n^2 \log n)$.

Next, given a query point p, we locate the face f of $\mathcal{A}(C)$ containing p, access the leaf of T to which f points, construct the path of T from this leaf to the root of T, and report the sets of circles stored at each of the $O(\log n)$ nodes along that path. It is easily seen that each circle containing p appears in exactly one of these sets, which implies the correctness of the query.

We now return to our original problem of computing a compact representation of the set Z of all point-circle containments, for a set C of n congruent circles and a set P of m points in the plane, as defined above. We first describe a "base" solution

to this problem, which is less efficient than our final algorithm, but is required as a subroutine in that algorithm.

We distinguish between two cases. Assume first that $n \leq m^{1/2}$. Construct the data structure of Theorem 3.1 for the collection C of n circles and perform m queries in it, one query for each point $p \in P$; we store p at each node along the path of the segment tree traced by the query. After completing the m queries, we produce the output by traversing all the nodes of the tree and by reporting, for each node t, the complete bipartite graph formed between the set C_t of circles stored at t and the set P_t of points stored at t (if either of these sets is empty, we ignore t). We thus obtain a collection of $O(m)$ complete edge-disjoint bipartite graphs $C_t \times P_t$, with

$$\sum_t |P_t|,\ \sum_t |C_t| = O(m \log n) = O(m \log m).$$

The total cost of the algorithm is

$$O(n^2 \log n + m \log n) = O(m \log n) = O(m \log m).$$

Now consider the case where $n > m^{1/2}$. Partition C into $\lceil \frac{n}{m^{1/2}} \rceil$ subsets of size at most $\lceil m^{1/2} \rceil$ each and construct the data structure of Theorem 3.1 for each of these subsets separately. Now for each $p \in P$ perform $\lceil \frac{n}{m^{1/2}} \rceil$ separate queries, one in each of these data structures. As in the former case, we defer the reporting until all the queries have been processed. The cost of the preprocessing is thus

$$\frac{n}{m^{1/2}} O((m^{1/2})^2 \log m^{1/2}) = O(nm^{1/2} \log m),$$

and the cost of the queries is

$$m \frac{n}{m^{1/2}} O(\log m^{1/2}) = O(nm^{1/2} \log m).$$

After all queries are performed, we traverse all the trees to obtain a collection of

$$\frac{n}{m^{1/2}} O(m) = O(nm^{1/2})$$

complete edge-disjoint bipartite graphs $C_t \times P_t$, as above, with

$$\sum_t |P_t|,\ \sum_t |C_t| = \frac{n}{m^{1/2}} O(m \log m) = O(nm^{1/2} \log m).$$

We next combine both cases and, for technical reasons required by our analysis, also interchange the roles of points and circles (which is possible since the circles are congruent, so a circle c_i contains a point p_j if and only if the circle congruent to c_i and centered at p_j contains the center of c_i) to obtain the following intermediate result.

THEOREM 3.2. *Let C be a set of n congruent circles, and let P be a set of m points in the plane. One can compute the set of pairs of the form (c, p), where $p \in P$, $c \in C$, and p lies inside c, as a collection of complete edge-disjoint bipartite graphs $\{C_t \times P_t\}_t$ in $O((mn^{1/2} + n) \log n)$ space and time. The number of graphs obtained is $O(mn^{1/2} + n)$, and we have $\sum_t |P_t|, \sum_t |C_t| = O((mn^{1/2} + n) \log n)$. Each such point-circle containment pair appears in exactly one of these graphs.*

Next we fix some parameter $1 \leq r \leq n$ (not necessarily a constant) and compute a $(1/r)$-*cutting* for C. That is, we partition the plane into $k = O(r^2)$ cells $\Delta_1, \ldots, \Delta_k$,

each of constant description complexity, so that each cell is intersected by at most n/r circles. This can be done in $O(nr)$ deterministic time, as in [11, 30]. Within this time bound we can also obtain for each cell Δ_i the set C_i of circles crossing Δ_i.

Let $P_i = P \cap \Delta_i$ and $m_i = |P_i|$. Clearly $\sum_{i=1}^{k} m_i = m$. We can compute the sets $P_1, \ldots, P_k$ using $O(r^2 + m)$ space and $O((r^2 + m) \log r)$ time (by preprocessing the above cutting for efficient point location and then by performing m queries with the points of P).

We obtain Z as the disjoint union of two subsets Z_1 and Z_2. The set Z_1 (resp., Z_2) consists of all pairs $(c, p) \in Z$ such that c intersects (resp., fully contains) the cell of the cutting in which p lies. Our final representation will consist of a compact representation of Z_1 and a compact representation of Z_2.

We first compute a compact representation of the set Z_1. The above cutting induces k subproblems, one for each cell of the cutting. In the ith subproblem we must consider the interaction between m_i points and at most n/r congruent circles. According to Theorem 3.2, this can be solved in $O((m_i(n/r)^{1/2} + n/r) \log(n/r))$ time and space. Thus all the k subproblems can be solved in a total of

$$\sum_{i=1}^{k} O((m_i(n/r)^{1/2} + n/r) \log(n/r)) = O((m(n/r)^{1/2} + nr) \log(n/r))$$

time and space. As a result, we obtain a collection of complete edge-disjoint bipartite graphs $C_t \times P_t$, so that the number of graphs is

$$O(m(n/r)^{1/2} + nr)$$

and

$$\sum_t |P_t|, \ \sum_t |C_t| = O((m(n/r)^{1/2} + nr) \log(n/r)).$$

Now we show how to compute a compact representation of the set Z_2. We compute an Eulerian path π of the dual graph of the cutting. The length of π is $O(r^2)$ (and it can be computed in $O(r^2)$ time), and each edge of the cutting is crossed at most twice by the path (once in each direction). Next we associate with each circle $c \in C$ the set of maximal intervals along the path such that for each node in such an interval, its corresponding cell is fully contained in c. We claim that the total number of such intervals, over all circles, is only $O(nr)$. To see this, observe that each such interval I can be charged to a pair (c, w), where c is the relevant circle and w is the node of π that follows I; note that c necessarily crosses the cell Δ of the cutting that w represents. Since each such Δ is crossed by at most n/r circles and π has only $O(r^2)$ nodes, the total number of intervals is at most $O(nr)$.

This argument also implies that we can find all these intervals in time $O(nr \log r)$: for each circle $c \in C$ mark all the nodes of π whose associated cell of the cutting is crossed by c (such a cell may correspond to many nodes of π, and we mark all of them). We next sort all the marked nodes along π and obtain the desired intervals as those nonempty "gaps" between consecutive pairs of marked nodes, which are fully contained in c. We repeat this for each $c \in C$, and the preceding arguments imply that the overall cost of this procedure is $O(nr \log r)$.

Now build a balanced segment tree T over the nodes of π, as above. The construction takes $O(r^2 + nr \log r)$ time and space. We next perform $O(r^2)$ queries, one for each cell of the cutting. (As above, each cell is mapped to only one leaf of T,

even though other leaves may represent the same cell, and the query only traces the path in T from that leaf to the root. The preceding arguments justify this strategy.) The queries take a total of $O(r^2 \log r)$ time. Again, we defer the reporting until all queries are processed and then report, for each node t of T, the complete bipartite graph formed between the circles stored at t and the union $\bigcup P_i$ over all cells Δ_i whose queries have reached t. This yields a collection of $O(r^2)$ complete edge-disjoint bipartite graphs $C_t \times P_t$, with $\sum_t |P_t|$, $\sum_t |C_t| = O(nr \log r)$.

Combining all the preceding steps, we see that the cost of the entire computation is dominated by $O((m(n/r)^{1/2} + nr) \log n)$. If we choose $r = \frac{m^{2/3}}{n^{1/3}}$, we obtain the following summary result.

THEOREM 3.3. *Let C be a set of n congruent circles and P be a set of m points in the plane. One can compute the set of pairs of the form (c, p), where $p \in P$, $c \in C$, and p lies inside c, as a collection $\{C_t \times P_t\}_t$ of complete edge-disjoint bipartite graphs in $O((m^{2/3}n^{2/3} + m + n) \log n)$ time and space. The number of graphs obtained is $O(m^{2/3}n^{2/3} + m + n)$, and we have $\sum_t |P_t|$, $\sum_t |C_t| = O((m^{2/3}n^{2/3} + m + n) \log n)$. Each such point-circle containment pair appears in exactly one of these graphs.*

Remark 1. As a special case, we also obtain the oracle for our distance selection algorithm (see section 4), which has to compute the number of pairs (c, p), where p lies in c. The resulting algorithm thus takes $O((m^{2/3}n^{2/3} + m + n) \log n)$ time and $O(m^{2/3}n^{2/3} + m + n)$ space.

Remark 2. We also obtain an efficient solution to the following problem: given n congruent circles of radius r in the plane, count the number of (resp., report in a compact form all the) intersecting pairs. (Take as the set of points P the set of centers of the given circles, and let the set C consist of circles of radius $2r$ about these centers.) The resulting algorithm thus takes $O(n^{4/3} \log n)$ time and $O(n^{4/3})$ (resp., $O(n^{4/3} \log n)$) space.

Theorem 3.3 and the first remark above can also easily be applied to the case where our ranges are congruent annuli rather than circles. We omit here the easy details.

THEOREM 3.4. *Let M be a set of n congruent annuli and P be a set of m points in the plane. One can compute the set of pairs of the form (A, p), where $p \in P$, $A \in M$, and p lies inside A, as a collection $\{M_t \times P_t\}_t$ of complete edge-disjoint bipartite graphs in $O((m^{2/3}n^{2/3} + m + n) \log n)$ time and space. The number of graphs obtained is $O(m^{2/3}n^{2/3} + m + n)$, and we have $\sum_t |P_t|$, $\sum_t |M_t| = O((m^{2/3}n^{2/3} + m + n) \log n)$. Each such point-annulus containment pair appears in exactly one of these graphs.*

4. Selecting distances in the plane. In this section we present our technique as applied to the distance selection problem. Let $P = \{p_1, \ldots, p_n\}$ be a given set of n points in the plane in general position, and let k be a given integer in the range $[1, \binom{n}{2}]$. The problem is to find the kth smallest distance among the points of P. The algorithm proceeds in stages; the jth stage produces an interval $I_j = (\alpha_j, \beta_j)$, so that the kth smallest distance among the points of P is known to lie in that interval, and the total number of distances among the points of P that lie in I_j is at most $O(n^2 \rho^j)$ for some constant parameter $\rho < 1$. Thus, after logarithmically many stages, we are left with a sufficiently small number of distances, from which it will be easy to retrieve the desired distance. Initially, we have $I_0 = (0, \infty)$.

For clarity of exposition, we first describe an initial version of the algorithm, which does not yet employ our Cole-like improvement. In this version, the number of calls to the oracle in each stage is $O(\log n)$. Since the number of stages is $O(\log n)$ and each oracle call takes $O(n^{4/3} \log n)$, the complexity of the entire algorithm is $O(n^{4/3} \log^3 n)$.

Afterwards we will obtain an improved version by applying our Cole-like technique to step 4 of each stage (see below). This will reduce the number of calls to the oracle to only $O(1)$ per stage and thus improve the complexity of the entire algorithm by a logarithmic factor.

An initial version: The jth stage proceeds as follows.

1. For $i = 1, \ldots, n$, let A_i denote the annulus centered at p_i and having radii α_{j-1} and β_{j-1}. Let M denote the set of these annuli. We perform batched range searching with the sets M and P, using the technique described in section 3, whose results are summarized in Theorem 3.4. This batched range searching thus takes time and space $O(n^{4/3} \log n)$, and its output consists of a collection of $O(n^{4/3})$ complete bipartite graphs $\{M_t \times P_t\}_t$, where $\sum_t |P_t|, \sum_t |M_t| = O(n^{4/3} \log n)$. Note also that $\sum_t |P_t| \cdot |M_t|$ is twice the number of pairs of points of P whose distance lies between α_{j-1} and β_{j-1}, and is thus $O(n^2 \rho^{j-1})$ by assumption. If this number is sufficiently small, we simply examine all edges of these graphs and select the desired distance, terminating the algorithm. (For this we need to know the number k_0 of pairs of points at distance $\leq \alpha_{j-1}$; our desired distance is then the $(k-k_0)$th smallest distance represented in the above bipartite graph collection. We obtain k_0 as in Remark 1 following Theorem 3.3.)
2. Replace each complete bipartite graph $M_t \times P_t$ by the following smaller graph G_t. Put $m_t = |M_t|$ and $n_t = |P_t|$. Partition M_t into $k = \lfloor \frac{m_t}{n_t} \rfloor$ subsets, $M_{t1}, \ldots, M_{tk}$, so that each subset, with the possible exception of the last one M_{tk}, has exactly n_t elements, and M_{tk} has between n_t and $2n_t - 1$ elements. For each $i \leq k$, construct a d-regular LPS-expander graph G_{ti} on the vertex set $M_{ti} \cup P_t$. Let G_t be the union of all these graphs. Assuming d to be a constant, the number of edges of G_t is clearly $O(|M_t| + |P_t|)$. Thus the total number of edges of all the graphs G_t is proportional to $\sum_t O(|M_t| + |P_t|) = O(n^{4/3} \log n)$.
3. Each edge (A, p) of G_t that connects a node representing an annulus with a node representing a point is associated with the distance between p and the center of A. (All other edges of G_t are ignored.) Let L be the list of all these edge distances, over all graphs G_t, sorted in increasing order. The length of L is $O(n^{4/3} \log n)$.
4. Now run a binary search over L, using as an oracle, for each $\lambda \in L$, a counting variant of the batched range-searching procedure of section 3, involving the set P of points and the set of circles of radius λ centered at the points of P. As shown in section 3, the cost of an oracle call is also $O(n^{4/3} \log n)$. This yields a new interval $I_j = (\alpha_j, \beta_j)$ where the kth smallest distance is known to lie. Note that the interval I_j has the property that it contains no distance induced by any edge of any of the graphs G_t.
5. We now repeat the whole procedure with the new interval I_j.

Once the algorithm terminates, its correctness is easy to prove. The main step in the analysis is to establish the invariant concerning the number of distances that lie in each interval I_j, thereby showing that the algorithm does indeed terminate and that it takes only $O(\log n)$ stages to do so.

Fix one of the pairs (M_t, P_t) obtained by the partitioning procedure of step 1. Let M_{ti} be one of the subsets of M_t in the partition of step 2. Replace each annulus A in M_{ti} by another annulus A' with the same center as A but with radii α_j and β_j (see Figure 3); let M'_{ti} denote the resulting set of annuli. Compute (only for the purpose of

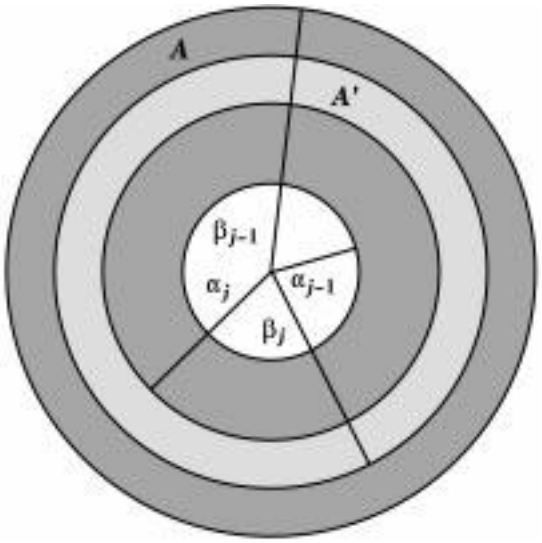


FIG. 3. *The concentric annuli A and A'.*

analysis) a $1/r$-cutting for the set M'_{ti} of size $O(r^2)$. That is, fix some parameter r and partition the plane into $k = O(r^2)$ cells, $\Delta_1, \ldots, \Delta_k$, so that each cell is intersected by at most $|M'_{ti}|/r$ of the circles bounding the annuli of M'_{ti}. (Actually, for our analysis, it is sufficient to assume that for some constant $c > 0$, a $1/r$-cutting of size $O(r^c)$ can be computed.)

For each $u = 1, \ldots, k$, let $P_t^{(u)}$ denote the set of points in P_t that lie inside Δ_u, let $K_{ti}^{'(u)}$ denote the set of annuli in M'_{ti} that fully contain Δ_u, and let $L_{ti}^{'(u)}$ denote the set of annuli in M'_{ti} that have a bounding circle that crosses Δ_u. It is clear that the number D_{ti} of distances between centers of annuli in M'_{ti} and points in P_t that lie in I_j satisfies

$$D_{ti} \leq \sum_u |L_{ti}^{'(u)}| \cdot |P_t^{(u)}| + \sum_u |K_{ti}^{'(u)}| \cdot |P_t^{(u)}|.$$

Since, for each u, we have $|L_{ti}^{'(u)}| \leq |M'_{ti}|/r = |M_{ti}|/r$, and since $\sum_u |P_t^{(u)}| = |P_t|$, it follows that

$$\sum_u |L_{ti}^{'(u)}| \cdot |P_t^{(u)}| = O\left(\frac{|M_{ti}| \cdot |P_t|}{r}\right).$$

Next consider the second sum $\sum_u |K_{ti}^{'(u)}| \cdot |P_t^{(u)}|$. Since, for each $A' \in K_{ti}^{'(u)}$ and each $p \in P_t^{(u)}$, the distance between p and the center of A' is in I_j, and since the corresponding expander G_{ti} has no edge whose distance lies in I_j, it follows that G_{ti} contains no edge connecting between $K_{ti}^{'(u)}$ (or, more precisely, the original annuli in M_{ti} corresponding to the annuli of $K_{ti}^{'(u)}$) and $P_t^{(u)}$. Hence, by Corollary 2.4, it follows that

$$|K_{ti}^{'(u)}| \cdot |P_t^{(u)}| \leq \frac{4(|M_{ti}| + |P_t|)^2}{d}.$$

Since there are only $O(r^2)$ cells, it follows that the corresponding sum satisfies

$$\sum_u |K_{ti}^{'(u)}| \cdot |P_t^{(u)}| = O\left(\frac{r^2(|M_{ti}| + |P_t|)^2}{d}\right).$$

Hence,

$$D_{ti} = O\left(\frac{r^2(|M_{ti}| + |P_t|)^2}{d}\right) + O\left(\frac{|M_{ti}| \cdot |P_t|}{r}\right).$$

Recall that $|P_t| \leq |M_{ti}| < 2|P_t|$, which is easily seen to imply that $(|M_{ti}| + |P_t|)^2 \leq 5|M_{ti}| \cdot |P_t|$. Hence,

$$D_{ti} = O\left(|M_{ti}| \cdot |P_t| \cdot \left[\frac{r^2}{d} + \frac{1}{r}\right]\right).$$

If we now choose $r = d^{1/3}$ and ρ to be appropriately proportional to $1/d^{\frac{1}{3}}$, then we have, for each of these pairs,

$$D_{ti} \leq \rho |M_{ti}| \cdot |P_t|.$$

Hence, summing up all these inequalities, we conclude that the total number of distances between the points of P that fall in I_j is at most ρ times the number of these distances within I_{j-1}. This establishes the desired invariant, thus completing the analysis of the algorithm, and yielding the following intermediate result.

LEMMA 4.1. *The above algorithm computes the kth smallest distance among n points in the plane in (deterministic) time* $O(n^{4/3} \log^3 n)$.

An improved version: We now apply our Cole-like improvement to step 4 of each stage of the algorithm. This will reduce the number of oracle calls in a single stage of the algorithm to only a constant and will improve the complexity of the algorithm by a logarithmic factor. To this end, we replace step 4 above by the following.

4. Let W be the number of pairs of points of P whose distance lies in I_{j-1}; that is, $W = 1/2 \sum_t |P_t| \cdot |M_t|$. We assume that $W \geq 8$; otherwise, we can simply check the constant number of distances in I_{j-1} and select out of them the desired distance. We assign a weight to each distance in L, so that the weight of each of the at most $(|P_t| + |M_{ti}|)d/2$ distances induced by the edges of an expander G_{ti} is $|P_t||M_{ti}|/(|P_t| + |M_{ti}|)$ (recall that some of these expander edges do not induce a distance; see step 3 of the algorithm). The total weight of the distances in L is

$$\leq \sum_{t,i} (|P_t| + |M_{ti}|) \frac{d}{2} \cdot \frac{|P_t||M_{ti}|}{|P_t| + |M_{ti}|} = \frac{d}{2} \sum_{t,i} |P_t| \cdot |M_{ti}| = Wd,$$

and, using the fact that $|P_t| \leq |M_{ti}|$, the weight of any single distance in L is

$$\frac{|P_t||M_{ti}|}{|P_t| + |M_{ti}|} < |P_t| \leq \sqrt{|P_t| \cdot |M_{ti}|} \leq \sqrt{2W} \leq \frac{W}{2},$$

since we have assumed that $W \geq 8$. Partition the list L into at most $2d$ intervals, each of weight at most W and at least $W/2$, and perform at most $\log(2d) = O(1)$ oracle calls to obtain the interval I_j that contains the desired distance. (Note that there is no need to sort L for the binary search; instead,

one may use repeated weighted-median findings on appropriate portions of L, which take a total time linear in the size of L.) Below we show that although now I_j is not necessarily defined by two consecutive values in L, the number of distances between the points of P that lie in I_j is still only ρ times the number of these distances within I_{j-1} for some constant $\rho < 1$.

It remains to show that, after the above change, it is still true that the number of distances that lie in the interval I_j is only $O(n^2\rho^j)$ for some constant $\rho < 1$. The analysis below is similar to the analysis for the basic version. The main difference is that we use Corollary 2.5 instead of Corollary 2.4.

As above, fix one of the pairs (M_t, P_t) obtained by the partitioning procedure of step 1 and let M_{ti} be one of the subsets of M_t in the partition of step 2. Suppose first that the corresponding expander graph G_{ti} contains at least $3(|P_t| + |M_{ti}|)$ edges whose distances lie in I_j. Then, by definition, G_{ti} contributes at least $3|P_t| \cdot |M_{ti}|$ to the total weight of edges whose distances lie in I_j. Since, by construction, this total weight is at most W, we conclude that $\sum_{t,i} |P_t| \cdot |M_{ti}|$, over all t, i for which G_{ti} has the above property, is at most $W/3$.

Thus we only need to consider graphs G_{ti} that have fewer than $3(|P_t| + |M_{ti}|)$ edges whose distances lie in I_j. Let M'_{ti} be the corresponding set of new annuli, as above, and compute (only for the purpose of analysis) a $1/r$-cutting of M'_{ti}, as above, to obtain the sets $P_t^{(u)}$, $K_{ti}^{'(u)}$, and $L_{ti}^{'(u)}$, $u = 1, \ldots, r$. As above, the number D_{ti} of distances between centers of annuli in M'_{ti} and points in P_t that lie in I_j satisfies

$$D_{ti} \leq \sum_u |L_{ti}^{'(u)}| \cdot |P_t^{(u)}| + \sum_u |K_{ti}^{'(u)}| \cdot |P_t^{(u)}| = O\left(\frac{|M_{ti}| \cdot |P_t|}{r}\right) + \sum_u |K_{ti}^{'(u)}| \cdot |P_t^{(u)}|.$$

Next consider the sum $\sum_u |K_{ti}^{'(u)}| \cdot |P_t^{(u)}|$. Since, for each $A' \in K_{ti}^{'(u)}$ and each $p \in P_t^{(u)}$, the distance between p and the center of A' is in I_j, and since the corresponding expander G_{ti} has fewer than $3(|P_t| + |M_{ti}|)$ edges whose distances lie in I_j, it follows that G_{ti} contains fewer than $3(|P_t| + |M_{ti}|)$ edges connecting between $K_{ti}^{'(u)}$ (or, more precisely, the original annuli in M_{ti} corresponding to the annuli of $K_{ti}^{'(u)}$) and $P_t^{(u)}$. Hence, by Corollary 2.5, it follows that

$$|K_{ti}^{'(u)}| \cdot |P_t^{(u)}| \leq \frac{9(|M_{ti}| + |P_t|)^2}{d}.$$

Thus we can repeat the analysis given above, with the only difference that the constants of proportionality are now larger, to conclude that, for an appropriate constant $\rho = O(1/d^{\frac{1}{3}})$, we have $D_{ti} \leq \rho |M_{ti}| \cdot |P_t|$. Hence, summing up all these inequalities and adding the contribution of expanders G_{ti} with many edges whose distances fall in I_j, we conclude that the total number of distances between the points of P that fall in I_j is at most $(\rho + \frac{1}{6})$ times the number of these distances within I_{j-1}. The factor $(\rho + \frac{1}{6})$ is smaller than 1 if we choose d sufficiently large. This establishes the desired invariant and thus completes the analysis of the algorithm. Since both the range-searching procedure and the oracle procedure cost $O(n^{4/3} \log n)$, the total running time of the above algorithm is $O(n^{4/3} \log^2 n)$. To summarize, we have shown the following theorem.

THEOREM 4.2. *The improved algorithm computes the kth smallest distance among n points in the plane in (deterministic) time* $O(n^{4/3} \log^2 n)$.

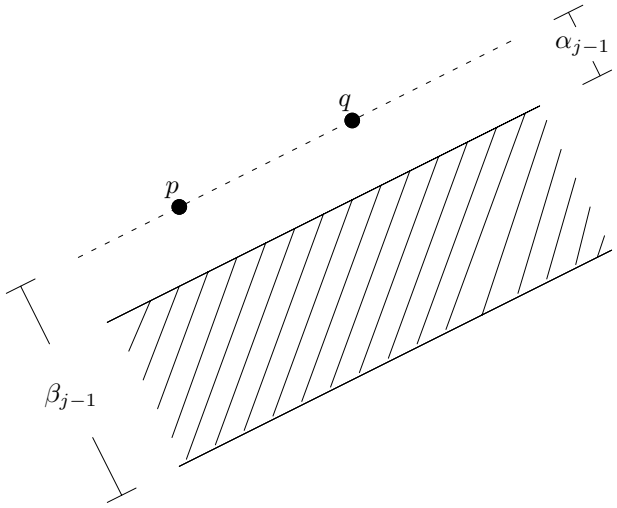


FIG. 4. *The strip $A^r_{p,q}$ defined by p and q.*

5. The two-line center problem. In this section we apply our new technique to obtain an improved algorithm for the two-line center problem. Let P be a given set of n points in the plane in general position. The problem is to find two strips whose union contains P, so that the larger width w^* of the two strips is as small as possible. Note that w^* is determined by three points of P—two points defining one of the borders of the wider strip and the third point defining the opposite border. We use the algorithm of Agarwal and Sharir [6], whose running time is $O(n^2 \log^3 n)$, for the appropriate decision problem: given a value $w > 0$, determine whether $w^* < w$, $w^* > w$, or $w^* = w$.

The main algorithm proceeds in stages; the jth stage produces an interval $I_j = (\alpha_j, \beta_j)$, so that w^* is known to lie in that interval, and the total number of strips defined by triples of points in P whose width lies in I_j is at most $O(n^3\rho^j)$ for some constant parameter $\rho < 1$. Thus, after logarithmically many stages, we are left with a sufficiently small number of widths, from which it will be easy to retrieve the desired width. Initially, we have $I_0 = (0, \infty)$. The jth stage proceeds as follows.

1. For each pair of points $p, q \in P$, draw the open strip $A^r_{p,q}$ of width $\beta_{j-1} - \alpha_{j-1}$, whose borders are parallel to the line passing through p and q and lie to the right of this line at distances α_{j-1} and β_{j-1}, respectively (see Figure 4). Let $A^l_{p,q}$ denote the strip symmetric to $A^r_{p,q}$ with respect to the line through p and q. (We note that, for $p_1, p_2, p_3 \in P$, we have $p_3 \in A^r_{p_1,p_2}$ (resp., $p_3 \in A^l_{p_1,p_2}$) if and only if the width of the strip defined by $(\{p_1, p_2\}, p_3)$ (i.e., defined by the line through p_1 and p_2 and the line parallel to it through p_3) lies in $(\alpha_{j-1}, \beta_{j-1})$ and the center line of this strip lies to the right (resp., to the left) of p_1p_2.) Let M denote the set of these $n(n-1)$ strips. Perform $n(n-1)$ batched range-searching queries on the set P with these strips. Using standard techniques, such as those of [31], [32], or [13], we can perform these queries so that the output consists of a collection of pairs (M_t, P_t), where, for each t, P_t is a subset of P, M_t is a subset of M, for each $p \in P_t$ and each $A \in M_t$ we have $p \in A$, and each such containment is obtained in exactly one

pair $M_t \times P_t$. Moreover, using an approach similar to that of [32], one can show that the number of pairs is at most $O(n^2)$, $\sum_t |P_t| = O(n^2)$, $\sum_t |M_t| = O(n^2 \log^2 n)$ and the batched range searching takes time $O(n^2 \log^2 n)$. Note also that $\sum_t |P_t| \cdot |M_t|$ is equal to the number of strips with width between α_{j-1} and β_{j-1} that are defined by triples of points of P, and is thus $O(n^3 \rho^{j-1})$ by assumption. If this number is sufficiently small, we find the desired width by examining all the relevant strips, in increasing width order, and stop.

2. For each pair (M_t, P_t), construct a graph G_t exactly as in step 2 of the preceding algorithm. Here the total number of edges of all the graphs G_t is proportional to $\sum_t O(|M_t| + |P_t|) = O(n^2 \log^2 n)$.
3. Each edge (A, p) of G_t is associated with the width of the strip spanned by the two points defining A and the point p. Let L be the list of all these edge values, over all graphs G_t, sorted in increasing order. The length of L is $O(n^2 \log^2 n)$. (As above, only some of the edges of G_t are used to form L.)
4. Now run a binary search over L, using the oracle of [6], to obtain a new interval $I_j = (\alpha_j, \beta_j)$ where w^* is known to lie. As in section 4, we make only a constant number of oracle calls, employing a weighing scheme similar to that used above, with obvious and straightforward modifications. (As in the preceding algorithm, there is no need to actually sort L for the binary search.)
5. We now repeat the whole procedure with the new interval I_j.

Once the algorithm terminates, its correctness is easy to establish. Using an analysis similar to that of the preceding section (which we therefore omit), one easily shows that the algorithm does indeed terminate and that it takes only $O(\log n)$ stages to do so. The running time of the algorithm is thus $O(\log n)$ times the cost of a single stage, and that cost is dominated by the cost of an oracle call, which is $O(n^2 \log^3 n)$ [6]. Thus we have the following theorem.

THEOREM 5.1. *The above algorithm solves the two-line center problem in (deterministic) time* $O(n^2 \log^4 n)$.

Remark. Note that if the oracle of [6] could be improved, say by a logarithmic factor, the complexity of the above algorithm would improve to $O(n^2 \log^3 n)$. We leave this as an open problem for further research.

6. Slope selection. In this section we consider the slope selection problem and develop an algorithm that is very similar to those obtained in the preceding sections. Let $\mathcal{L} = \{\ell_1, \ldots, \ell_n\}$ be a given collection of n lines in the plane in general position, and let $1 \le k \le \binom{n}{2}$ be an integer. We want to find the kth leftmost vertex v_k of the arrangement $\mathcal{A}(\mathcal{L})$ of $\mathcal{L}$. As in the preceding section, the algorithm proceeds in $O(\log n)$ stages; the jth stage produces a vertical slab $\sigma_j = (\alpha_j, \beta_j)$ (this is a shorthand notation for $\alpha_j < x < \beta_j$), so that v_k is known to lie in σ_j, and the total number of vertices of $\mathcal{A}(\mathcal{L})$ within σ_j is at most $O(n^2 \rho^j)$ for some constant parameter $\rho < 1$. Initially, we have $\sigma_0 = (-\infty, \infty)$ (that is, the entire plane).

The jth stage proceeds as follows.

1. For a line ℓ, let $\xi(\ell)$, $\eta(\ell)$ denote the y-coordinates of the intercepts of ℓ with the left and right borders of σ_{j-1}, respectively. Note that two lines ℓ and ℓ' intersect within σ_{j-1} if and only if either
   (i) $\xi(\ell) < \xi(\ell')$ and $\eta(\ell) > \eta(\ell')$ or
   (ii) $\xi(\ell) > \xi(\ell')$ and $\eta(\ell) < \eta(\ell')$.
   In other words, we can map each line ℓ_i to the point $\ell_i^* = (\xi(\ell_i), \eta(\ell_i))$ in a dual plane, so that, given any query line ℓ, the set of lines of $\mathcal{L}$ that intersect ℓ

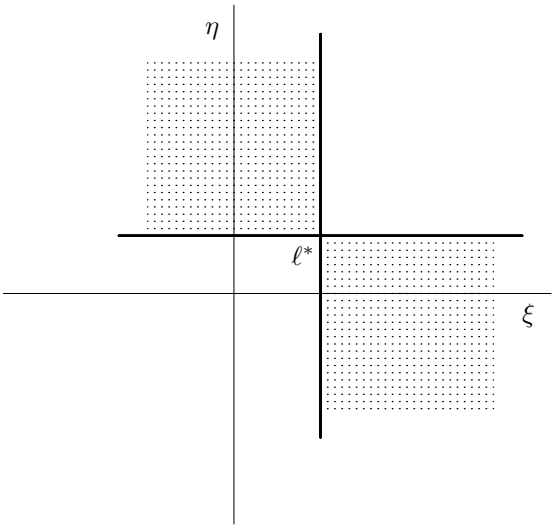


FIG. 5. *The two quadrant ranges defined by ℓ.*

within σ_{j-1} is the (set dual to the) output of the following pair of *orthogonal range-searching queries* (see Figure 5):

$$\{\ell_i^* \quad | \quad \xi(\ell_i) < \xi(\ell),\ \eta(\ell_i) > \eta(\ell)\},$$

$$\{\ell_i^* \quad | \quad \xi(\ell_i) > \xi(\ell),\ \eta(\ell_i) < \eta(\ell)\}.$$

To perform such range-searching queries, we use the standard two-dimensional *range tree* data structure. As is well known, we can perform our $2n$ quadrant range queries in this structure, so that each query takes $O(\log^2 n)$ time and returns its output as the disjoint union of $O(\log^2 n)$ canonical subsets. Thus we obtain a collection of pairs $(\mathcal{K}_t, \mathcal{L}_t)$ of subsets of $\mathcal{L}$ (where each $\mathcal{L}_t$ is some canonical set and $\mathcal{K}_t$ is the set of queries associated with it) that satisfy the following properties.

(i) For each t, every line $\ell \in \mathcal{K}_t$ intersects every line $\ell' \in \mathcal{L}_t$ within σ_{j-1}.
(ii) For each pair of lines ℓ, ℓ' of $\mathcal{L}$ that intersect within σ_{j-1}, there exists a unique t such that $\ell \in \mathcal{K}_t$, $\ell' \in \mathcal{L}_t$.
(iii) $\sum_t |\mathcal{L}_t|, \sum_t |\mathcal{K}_t| = O(n \log^2 n)$.

Properties (i) and (ii) imply that the total number of vertices of $\mathcal{A}(\mathcal{L})$ within σ_{j-1} is at most $\sum_t |\mathcal{L}_t| \cdot |\mathcal{K}_t|$.

2. For each pair $(\mathcal{K}_t, \mathcal{L}_t)$, construct a graph G_t in the same manner as in the preceding sections. The total number of edges of all the graphs G_t is proportional to $\sum_t O(|\mathcal{K}_t| + |\mathcal{L}_t|) = O(n \log^2 n)$.
3. Each edge (ℓ, ℓ') of G_t, with $\ell \in \mathcal{K}_t, \ell' \in \mathcal{L}_t$, is associated with the vertex of $\mathcal{A}(\mathcal{L})$ formed by the intersection of ℓ and ℓ'. Let L be the list of all these vertices, over all graphs G_t, sorted in the order of increasing x-coordinates. The length of L is $O(n \log^2 n)$.
4. Now run a binary search over L, using the inversion-counting oracle mentioned in the introduction, to obtain a new slab $\sigma_j = (\alpha_j, \beta_j)$ where the kth leftmost

vertex v_k is known to lie. Note that the slab σ_j has the property that it contains no vertex associated with any edge of any of the graphs G_t. (As in the preceding sections, there is no need to sort L, and the repeated median findings that are used instead take only linear time in $|L|$.)

5. We now repeat the whole procedure with the new slab σ_j until we obtain a slab that contains only a single vertex, and then we terminate the algorithm and report that vertex.

Once the algorithm terminates, its correctness is easy to establish (under the assumption of general position). The analysis, showing that the algorithm does indeed terminate and that it takes only $O(\log n)$ stages to do so, is similar to that given in the preceding sections. Specifically, fix one of the pairs $(\mathcal{K}_t, \mathcal{L}_t)$ obtained by the range-searching procedure of step 1. Let $\mathcal{K}_{ti}$ be one of the subsets of $\mathcal{K}_t$ in the partition of step 2. Consider a modified dual plane, obtained by replacing the ξ and η coordinates by the respective intercepts ξ', η' of lines along the left and right borders of the new slab σ_j. We fix some parameter r, to be determined shortly, and apply (only for the purpose of analysis) the following partitioning of the new dual plane. The plane is partitioned into r vertical slabs, so that each slab contains $|\mathcal{L}_t|/r$ points dual to lines in $\mathcal{L}_t$, and then each slab is partitioned, by horizontal cuts, into r rectangles, each containing $|\mathcal{L}_t|/r^2$ dual points. For each line $\ell \in \mathcal{K}_{ti}$, consider the pair of quadrants

$$Q_1(\ell) = \{\xi < \xi'(\ell),\ \eta > \eta'(\ell)\},$$

$$Q_2(\ell) = \{\xi > \xi'(\ell),\ \eta < \eta'(\ell)\}.$$

For each of the resulting quadrants Q, there are at most $2r$ rectangles of the partitioning that are crossed by the boundary of Q; each other rectangle is either fully contained within Q or lies fully outside Q.

For each $u = 1, \ldots, r$, let R_u denote the uth rectangle of the partitioning, let $\mathcal{L}_t^{(u)}$ denote the set of lines of $\mathcal{L}_t$ whose dual points fall in R_u, let $\mathcal{Q}_{ti}^{'(u)}$ denote the set of lines of $\mathcal{K}_{ti}$ whose associated quadrants fully contain R_u, and let $\mathcal{Q}_{ti}^{''(u)}$ denote the set of lines of $\mathcal{K}_{ti}$ whose associated quadrants are such that their boundary crosses R_u. It is clear that the number D_{ti} of intersection points between lines in $\mathcal{L}_t$ and lines in $\mathcal{K}_{ti}$ that lie in σ_j satisfies

$$D_{ti} \le \sum_u |\mathcal{Q}_{ti}^{'(u)}| \cdot |\mathcal{L}_t^{(u)}| + \sum_u |\mathcal{Q}_{ti}^{''(u)}| \cdot |\mathcal{L}_t^{(u)}|.$$

By the property of "small crossing number" that our partition has, it follows that $\sum_u |\mathcal{Q}_{ti}^{''(u)}| = O(|\mathcal{K}_{ti}|r)$, so

$$\sum_u |\mathcal{Q}_{ti}^{''(u)}| \cdot |\mathcal{L}_t^{(u)}| = O\left(\frac{|\mathcal{K}_{ti}| \cdot |\mathcal{L}_t|}{r}\right).$$

Next consider the other sum $\sum_u |\mathcal{Q}_{ti}^{'(u)}| \cdot |\mathcal{L}_t^{(u)}|$. Since, for each line $\ell' \in \mathcal{Q}_{ti}^{'(u)}$ and each line $\ell \in \mathcal{L}_t^{(u)}$, the intersection point between ℓ and ℓ' lies in σ_j, and since the corresponding expander G_{ti} has no edge whose corresponding intersection point lies in σ_j, it follows that G_{ti} contains no edge connecting between $\mathcal{Q}_{ti}^{'(u)}$ and $\mathcal{L}_t^{(u)}$. Hence, by Corollary 2.4, it follows that

$$|\mathcal{Q}_{ti}^{'(u)}| \cdot |\mathcal{L}_t^{(u)}| \le \frac{4(|\mathcal{K}_{ti}| + |\mathcal{L}_t|)^2}{d}.$$

Since there are only r^2 indices u, it follows that the corresponding sum satisfies

$$\sum_u |\mathcal{Q}_{ti}^{'(u)}| \cdot |\mathcal{L}_t^{(u)}| \leq \frac{4r^2(|\mathcal{K}_{ti}| + |\mathcal{L}_t|)^2}{d}.$$

Hence, as in the preceding sections,

$$\begin{aligned} D_{ti} &= \frac{4r^2(|\mathcal{K}_{ti}| + |\mathcal{L}_t|)^2}{d} + O\left(\frac{|\mathcal{K}_{ti}| \cdot |\mathcal{L}_t|}{r}\right) \\ &= O\left(|\mathcal{K}_{ti}| \cdot |\mathcal{L}_t| \cdot \left[\frac{r^2}{d} + \frac{1}{r}\right]\right). \end{aligned}$$

If we now choose $r = d^{1/3}$ and ρ to be appropriately proportional to $1/d^{\frac{1}{3}}$, then we have, for each of these pairs, $D_{ti} \leq \rho|\mathcal{K}_{ti}| \cdot |\mathcal{L}_t|$, and, summing up all these inequalities, we conclude that the total number of vertices of $\mathcal{A}(\mathcal{L})$ that fall in σ_j is at most ρ times the number of these vertices within σ_{j-1}. This establishes the desired invariant, thus completing the analysis of the algorithm, and yielding the following summary result.

THEOREM 6.1. *The above algorithm computes the kth leftmost vertex in an arrangement of n lines in the plane in (deterministic) time* $O(n \log^3 n)$.

Remark. We have not used here the Cole-like improvement, since it does not improve the overall running time of the algorithm; see a discussion concerning this issue in the concluding section.

7. Other applications. In this section we briefly mention two other problems for which our approach yields efficient algorithms that are comparable with the corresponding known algorithms, which are based on parametric searching.

7.1. The two-center problem. In this subsection we present an algorithm for the planar two-center problem, defined as follows. Let P be a given set of n points in the plane in general position. We wish to find two closed discs whose union contains P, so that the larger radius r^* of the two discs is as small as possible. Note that r^* is determined either by a pair of points of P, in which case r^* is half the distance between these points, or by a triple of points of P, in which case r^* is the radius of the circumcircle of the triangle spanned by these points. We use the recent algorithm of Hershberger [23] (see also [6, 24]), whose running time is $O(n^2)$, for the corresponding decision problem: given a value $r > 0$, determine whether $r^* < r$, $r^* > r$, or $r^* = r$. We begin our algorithm with a preliminary stage in which we perform a binary search for r^* among the $\binom{n}{2}$ half-distances determined by pairs of points in P. The complexity of this stage is $O(n^2 \log n)$, and at its end we have either found r^* (if r^* is determined by a pair of points in P) or obtained an open interval (α_0, β_0) such that r^* is known to lie in this interval and none of the half-distances determined by pairs of points in P is in (α_0, β_0).

The main algorithm is very similar to the algorithm of section 5 for the two-line center problem. We therefore describe only the range-searching part of the jth stage.

For each pair of points $p, q \in P$, at distance less than $2\beta_{j-1}$, draw the two open discs $C_{p,q}^r, D_{p,q}^r$ of respective radii α_{j-1} and β_{j-1}, whose centers lie to the right of the segment pq and whose bounding circles pass through p and q. Let $L_{p,q}^r$ denote the symmetric difference of these two discs. Let $L_{p,q}^l$ denote the symmetric difference of the two discs with the same radii whose centers lie to the left of pq and whose bounding circles pass through p and q. We will refer to ranges like $L_{p,q}^r$ and $L_{p,q}^l$ as *double lunes* (see Figure 6). Note that, for $p_1, p_2, p_3 \in P$, we have $p_3 \in L_{p_1,p_2}^r$

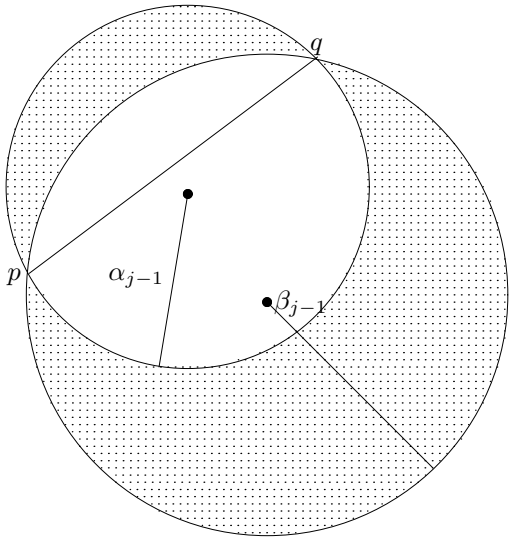


FIG. 6. *The double lune $L^r_{p,q}$ defined by p and q.*

(resp., $p_3 \in L^l_{p_1,p_2}$) if and only if the radius of the circumcircle of p_1, p_2, p_3 lies in $(\alpha_{j-1}, \beta_{j-1})$, and the center of this circle lies to the right (resp., to the left) of p_1p_2. We thus must compute the collection of all pairs of the form $(L^x_{p_1,p_2}, p_3)$, where x stands for either r or l and where p_3 lies in the double lune $L^x_{p_1,p_2}$, and represent it as a collection of complete bipartite graphs as in the previous examples. We distinguish among four types of such pairs, according to whether x is r or l and whether p_3 lies in $C^x_{p_1,p_2} - D^x_{p_1,p_2}$ or in $D^x_{p_1,p_2} - C^x_{p_1,p_2}$. Each type is handled separately. We will show how to compute the collection of all pairs of the form $(L^r_{p_1,p_2}, p_3)$, where p_3 lies in $C^r_{p_1,p_2} - D^r_{p_1,p_2}$; the other three types are computed in a similar and symmetric fashion.

Let $\mathcal{C}^r$ (resp., $\mathcal{D}^r$) denote the set of discs $C^r_{p,q}$ (resp., $D^r_{p,q}$), and let A denote the set of discs of radius α_{j-1} centered at the points of P. We construct a $(1/n)$-cutting for the set A, using the hierarchical decomposition technique of Chazelle [11], appropriately adapted to the case of circles. We retain the tree $\mathcal{T}$ that is obtained in the construction, where the nodes of the kth level of the tree represent the cells of the cutting at the kth level in the hierarchy (this is a $(1/r_0^k)$-cutting of A for some constant r_0). We denote the set of centers of the discs of $\mathcal{C}^r$ by $\overline{\mathcal{C}^r}$. We perform $\binom{n}{2}$ point-location queries in the final cutting, with the points of $\overline{\mathcal{C}^r}$, in batched mode; each query is performed by searching with the query point through an appropriate path of $\mathcal{T}$, so it takes $O(\log n)$ time, for a total of $O(n^2 \log n)$ time for all queries. At each node v of $\mathcal{T}$, excluding the root, we obtain a pair $(A_v, \overline{\mathcal{C}^r_v})$ of sets, where $A_v \subseteq A$ is the set of discs whose bounding circles cross the cell associated with the parent of v and which fully contain the cell associated with v, and where $\overline{\mathcal{C}^r_v} \subseteq \overline{\mathcal{C}^r}$ is the set of query points passing through v, i.e., lying within the cell associated with v. Note that $\sum_v |A_v|$, over all nodes v of $\mathcal{T}$, is $O(n^2)$ and that $\sum_v |\overline{\mathcal{C}^r_v}|$, over all nodes v at a fixed level of $\mathcal{T}$, is $O(n^2)$. For each node v we obtain the following subproblem: denote by $\mathcal{D}^r_v$ the subset of $\mathcal{D}^r$ that corresponds to the subset $\overline{\mathcal{C}^r_v}$, that is, the set of discs $D^r_{p_1,p_2}$, where the centers of the corresponding discs $C^r_{p_1,p_2}$ belong to $\overline{\mathcal{C}^r_v}$. Denote by $P_v \subseteq P$

the set of centers of the discs of A_v. We want to report, in compact form, all pairs $(D^r_{p_1,p_2}, p_3)$, where $D^r_{p_1,p_2} \in \mathcal{D}^r_v$ and $p_3 \in P_v$ such that p_3 does not lie in the interior of $D^r_{p_1,p_2}$. Put $|\overline{\mathcal{C}^r_v}| = m_v$ and $|A_v| = n_v$.

We now apply the algorithm of Theorem 3.1 to each of these subproblems. This requires a few (simple and straightforward) modifications because here we are interested in points lying in the *exterior* of the given discs; we omit the easy details. We can thus report the desired pairs, for the subproblem at a node v, in time and storage $O(n_v^2 \log n_v + m_v \log n_v)$ (for this, we need to interchange the roles of discs and points). The overall cost of the computation at all the nodes of the kth level of the tree is

$$O(s_k(n/r_0^{k-1})^2 \log(n/r_0^{k-1})) + \sum_{v \text{ at level } k} O(m_v \log(n/r_0^{k-1}))$$
$$= O(s_k(n/r_0^{k-1})^2 \log n + n^2 \log n),$$

where s_k is the size of the kth level cutting. Since the construction of [11] ensures that $s_k \le r_0^{2(k+1)}$, assuming that r_0 is sufficiently large, it follows that the cost of the computations at the nodes of the kth level of the tree is $O(r_0^4 n^2 \log n) = O(n^2 \log n)$. Thus, summing over all nodes of the tree, we get that the total cost of a single application of the range-searching step, as well as the overall size of the bipartite graphs that it produces, is $O(n^2 \log^2 n)$. This completes the description of the range-searching stage, and the rest of the algorithm remains essentially as in the previous examples. The total running time of the algorithm is thus $O(n^2 \log^3 n)$, matching the running time of the best previous algorithm of [6]. (Note that here we do not need to apply our Cole-like improvement because we can afford to perform $O(\log n)$ oracle calls per stage without increasing the overall running time.)

7.2. The minimum output rate problem. In this subsection we present algorithms for the *minimum output rate* problem: given consecutive time intervals $T_1, \ldots, T_n$, input rates $I_1, \ldots, I_n$ (where I_j is the input rate during the jth time interval), and a buffer size B, find the minimum output rate R^* required to assure that the buffer does not overflow. This problem was defined and solved by Megiddo, Naor, and Anderson [35], who give an $O(n \log n \log\log n)$ deterministic algorithm and an $O(n \log n)$ randomized algorithm, both based on parametric searching. Let $a_i \equiv T_i I_i$ denote the total amount of data received during the ith interval, for $i = 1, \ldots, n$, and define a set H of $\binom{n}{2} + n$ lines of negative slope in the plane:

$$l_{i,j} = (a_i + \cdots + a_j) - (T_i + \cdots + T_j)x, \quad 1 \le i \le j \le n.$$

Megiddo, Naor, and Anderson prove that the desired rate R^* is equal to the x-coordinate of the intersection point of the upper envelope of H and the horizontal line at height B, which we denote by l. That is, it is equal to the x-coordinate of the rightmost intersection point of the lines of H with l (see Figure 7).

Given some value x_0, we would like to compute a compact representation of the lines intersecting l to the right of x_0. Define (as in [35]) $W_i = W_i(x_0) = \sum_{k=1}^{i}(a_k - T_k x_0)$, $i = 1, \ldots, n$, and $W_0 = 0$. (The values $W_1, \ldots, W_n$ can be computed in $O(n)$ time by computing the auxiliary values $U_i = \sum_{k=1}^{i} a_k$ and $V_i = \sum_{k=1}^{i} T_k x_0$, $i = 1, \ldots, n$.) Note that the line $l_{i,j}$ intersects l to the right of x_0 if and only if it intersects the vertical line through x_0 above l, that is, if and only if $W_j - W_{i-1} > B$. We compute a compact representation of the pairs (i, j), $0 \le i < j \le n$, such that $W_j - W_i > B$, as follows. Map the value W_i to the point $(i, W_i - B)$, $i = 0, \ldots, n$,

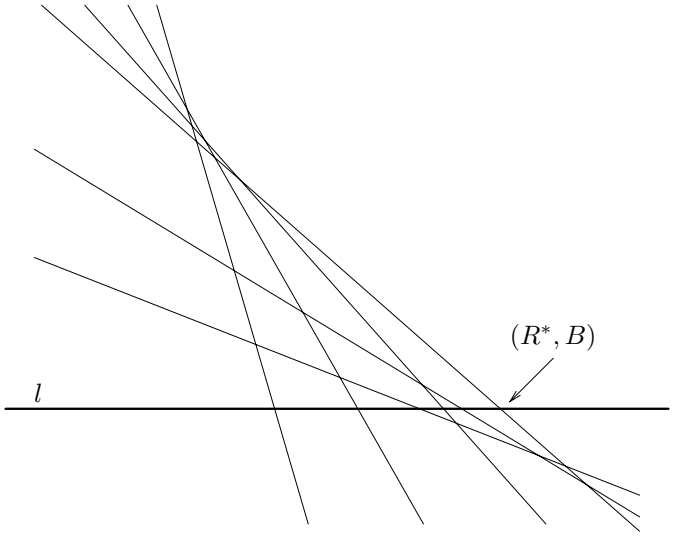


FIG. 7. *The minimum output rate problem in geometric setting.*

and let P denote the resulting set of points. Note that for any fixed index i, the set of pairs (i, j) with $W_j - W_i > B$ is determined by the set of points of P lying in the (open) northeast quadrant whose apex is the point (i, W_i). Thus we preprocess the points in P for efficient quadrant range searching. We can actually perform n batched queries to obtain in total time $O(n \log^2 n)$ the desired set of pairs as a collection of $O(n \log^2 n)$ complete bipartite graphs with vertex sets of total size $O(n \log^2 n)$.

The above simple method to compute a compact representation for the set of lines of H intersecting l to the right of some given x-value gives rise to a simple, expander-based $O(n \log^3 n)$ deterministic algorithm for the minimum output rate problem. We omit here the details, which are very similar to those in earlier sections.

We also note that the above method gives rise to an $O(n \log n)$ randomized algorithm, which we believe to be simpler than the one presented in [35]. We describe it here, even though it is not within the scope of this paper, since it is very simple and short (it also resembles the algorithms considered in [36]). Pick a random sample of size $O(n \log n)$ from the set H. (A random line can be chosen by picking randomly two indices i and j.) We regard the sample as a random sample of the intersection points of the lines of H with the line l. It is easy to show that, with high probability, the number of intersection points between any two consecutive sample points (alternatively, the number of intersection points to the left of the leftmost sample point or to the right of the rightmost sample point) is less than cn for some constant c. Let x_0 be the rightmost point in the sample. Below we describe a simple procedure for reporting the k intersection points to the right of x_0 in time $O(n \log n + k)$. By the above remark, the expected running time of this procedure is only $O(n \log n)$. After applying this procedure, we select the rightmost intersection point from the linear number of points that were output. The x-coordinate of this point is the desired R^*.

The reporting of the required intersections is equivalent to the reporting of all pairs (i, j) for which $i < j$ and $W_j - W_i > B$. We do this in stages. In the jth stage, for $j = 0, \ldots, n$, we report all pairs (i, j) such that $i < j$ and $W_j - W_i > B$. At the beginning of the jth stage the values $W_0, \ldots, W_{j-1}$ are already stored in some

balanced binary search tree (which is empty initially). We search in this tree with the value $W_j - B$ and report all pairs (i, j) such that W_i is smaller than $W_j - B$. Next we insert the value W_j into the tree and proceed to the next stage. The cost of the jth stage is clearly $O(\log n + k_j)$, where k_j is the number of pairs reported in this stage, implying a total reporting cost of $O(n \log n + k)$, as claimed.

8. Conclusion. In this paper we have proposed a new approach to geometric optimization problems which is based on expander graphs combined with range-searching techniques and have demonstrated its applicability to several specific problems, including the slope selection problem, the distance selection problem, and the two-line center problem. Our approach can be used instead of the parametric-searching technique or instead of randomized approaches of the sort suggested in [19, 28, 29]. It yields deterministic and rather simple algorithms, avoids the parallelization that is required in parametric searching, and also avoids some of the more complex generic comparisons that standard parametric-searching algorithms tend to generate.

How general is our technique? Roughly speaking, it works well when the critical values of the problem, through which we want to run a binary search, are obtained by interaction between pairs (or triples, or any fixed-size tuples) of the input objects. Rather than constructing all such critical values, we construct expanders on certain subsets of the data items and use their edges to retrieve only a small subset of critical values, thus speeding up the algorithm considerably. The cost of the algorithm is generally dominated by two terms: (i) $O(\log n)$ times the cost of the binary search oracle and (ii) $O(\log n)$ times the cost of the batched range searching that is performed at each stage of the algorithm. Often these two costs are similar, up to a polylogarithmic factor, and then our solution tends to be (more or less) as efficient as the solutions obtained by the previous methods. In fact, part (i) of the cost of our technique is at least as fast as the cost of the other (parametric-searching or randomized) approaches, and can be faster in cases where Cole's improvement cannot be applied in the parametric-searching solution. Inspecting the three main examples given above, we see that (a) in the slope selection problem the range-searching part was the bottleneck (so that we didn't even have to apply our Cole-like improvement); (b) in the two-line center problem the oracle cost was the bottleneck (so we didn't have to use a particularly fast batched range-searching technique); (c) only in the distance selection problem do the two parts of the cost balance each other.

To see what kinds of limitations and/or challenges are imposed by our technique, consider the problem of computing the diameter of a set P of n points in 3-space. The oracle for this problem is rather efficient [10] and can be implemented in deterministic $O(n \log n)$ time (see also [12], [33], and [16]). However, the range searching that seems to be required here is much more expensive: we are given the n points of P and n ranges, each being the exterior of a ball of some fixed radius centered at a point of P, and we want to perform these range-searching queries in batch mode. This seems to be much more expensive than an oracle call (the only techniques we are aware of require time close to $O(n^{4/3})$), so the naive application of our technique seems to be much less efficient than previous solutions. We pose it as an open problem to find an alternative and more efficient range-searching setting for this problem which will make our technique yield a solution with close-to-linear running time. Intuitively, the oracle is so efficient here because it exploits the delicate combinatorial property that the intersection of n congruent balls in 3-space has only linear complexity. So far we seem to lose this property when applying the range searching mentioned above. The

irony is that our approach does not need the oracle at all: assuming all the critical values are induced by expander edges, these values are distances between pairs of points of P, so the diameter must clearly be at least the largest of these values, so no binary search and thus no oracle calls are required.

We finally note that our approach could also be applied to other, nongeometric optimization problems, provided that the search for the optimal value of the parameter can be guided by an appropriate range-searching mechanism, for which the underlying range space has finite VC dimension (see [14, 22] for details). However, we will not elaborate on this possibility; rather, we leave it as another open problem to explore.

Acknowledgments. We wish to thank Noga Alon for several helpful discussions that provided valuable information concerning expanders and their applications, and Pankaj Agarwal for helpful discussions and suggestions concerning the range-searching technique of section 3.

REFERENCES

[1] P. K. Agarwal, personal communication.
[2] P. K. Agarwal, B. Aronov, M. Sharir, and S. Suri, *Selecting distances in the plane*, Algorithmica, 9 (1993), pp. 495–514.
[3] P. K. Agarwal, A. Efrat, M. Sharir, and S. Toledo, *Computing a segment center for a planar point set*, J. Algorithms, 15 (1993), pp. 314–323.
[4] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, SIAM J. Comput., 22 (1993), pp. 794–806.
[5] P. K. Agarwal and J. Matoušek, *On range searching with semi-algebraic sets*, Discrete Comput. Geom., 11 (1994), pp. 393–418.
[6] P. K. Agarwal and M. Sharir, *Planar geometric location problems*, Algorithmica, 11 (1994), pp. 185–195.
[7] P. K. Agarwal, M. Sharir, and S. Toledo, *Applications of parametric searching in geometric optimization*, J. Algorithms, 17 (1994), pp. 292–318.
[8] M. Ajtai and N. Megiddo, *A deterministic* Poly$(\log\log n)$*-time* n*-processor algorithm for linear programming in fixed dimension*, in Proc. 24th ACM Symp. on Theory of Computing, Victoria, British Columbia, 1992, pp. 327–338.
[9] N. Alon and J. Spencer, *The Probabilistic Method*, Wiley-Interscience, New York, 1992.
[10] H. Brönnimann, B. Chazelle, and J. Matoušek, *Product range spaces, sensitive sampling, and derandomization*, in Proc. 34th IEEE Symp. on Foundations of Computer Science, Palo Alto, CA, 1993, pp. 400–409.
[11] B. Chazelle, *Cutting hyperplanes for divide-and-conquer*, Discrete Comput. Geom., 9 (1993), pp. 145–158.
[12] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *Diameter, width, closest line-pair, and parametric searching*, Discrete Comput. Geom., 10 (1993), pp. 183–196.
[13] B. Chazelle, M. Sharir, and E. Welzl, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, Algorithmica, 8 (1992), pp. 407–429.
[14] B. Chazelle and E. Welzl, *Quasi-optimal range searching in spaces of finite VC-dimension*, Discrete Comput. Geom., 4 (1989), pp. 467–489.
[15] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl, *Combinatorial complexity bounds for arrangements of curves and spheres*, Discrete Comput. Geom., 5 (1990), pp. 99–160.
[16] K. Clarkson and P. Shor, *Applications of random sampling in computational geometry* II, Discrete Comput. Geom., 4 (1989), pp. 387–421.
[17] R. Cole, *Slowing down sorting networks to obtain faster sorting algorithms*, J. ACM, 34 (1987), pp. 200–208.
[18] R. Cole, J. Salowe, W. Steiger, and E. Szemerédi, *Optimal slope selection*, SIAM J. Comput., 18 (1989), pp. 792–810.
[19] M. Dillencourt, D. Mount, and N. Netanyahu, *A randomized algorithm for slope selection*, Internat. J. Comput. Geom. Appl., 2 (1992), pp. 1–27.
[20] A. Efrat, M. Sharir, and A. Ziv, *Computing the smallest* k*-enclosing circle and related problems*, Comput. Geom. Theory Appl., 4 (1994), pp. 119–136.

[21] M. Goodrich, *Geometric partitioning made easier, even in parallel*, in Proc. 9th ACM Symp. on Computational Geometry, San Diego, CA, 1993, pp. 73–82.
[22] D. Haussler and E. Welzl, *Epsilon nets and simplex range queries*, Discrete Comput. Geom., 2 (1987), pp. 127–151.
[23] J. Hershberger, *A faster algorithm for the two-center decision problem*, Inform. Process. Lett., 47 (1993), pp. 23–29.
[24] J. Hershberger and S. Suri, *Finding tailored partitions*, J. Algorithms, 12 (1991), pp. 431–463.
[25] M. J. Katz and M. Sharir, *Optimal slope selection via expanders*, Inform. Process. Lett., 47 (1993), pp. 115–122.
[26] A. Lubotzky, R. Phillips, and P. Sarnak, *Explicit expanders and the Ramanujan conjectures*, in Proc. 18th ACM Symp. on Theory of Computing, Berkeley, CA, 1986, pp. 240–246; see also *Ramanujan graphs*, Combinatorica, 8 (1988), pp. 261–277.
[27] G. A. Margulis, *Explicit group-theoretical constructions of combinatorial schemes and their applications to the design of expanders and superconcentrators*, Problemy Peredachi Informatsii, 24 (1988), pp. 51–60 (in Russian). English translation in Problems Inform. Transmission, 24 (1988), pp. 39–46.
[28] J. Matoušek, *Randomized optimal algorithm for slope selection*, Inform. Process. Lett., 39 (1991), pp. 183–187.
[29] J. Matoušek, *On enclosing k points by a circle*, Inform. Process. Lett., 53 (1995), pp. 217–221.
[30] J. Matoušek, *Approximations and optimal geometric divide-and-conquer*, J. Comput. System Sci., 50 (1995), pp. 203–208.
[31] J. Matoušek, *Efficient partition trees*, Discrete Comput. Geom., 8 (1992), pp. 315–334.
[32] J. Matoušek, *Range searching with efficient hierarchical cuttings*, Discrete Comput. Geom., 10 (1993), pp. 157–182.
[33] J. Matoušek and O. Schwarzkopf, *A deterministic algorithm for the three-dimensional diameter problem*, Comput. Geom. Theory Appl., 6 (1996), pp. 253–262.
[34] N. Megiddo, *Applying parallel computation algorithms in the design of serial algorithms*, J. ACM, 30 (1983), pp. 852–865.
[35] N. Megiddo, M. Naor, and D.P. Anderson, *The minimum reservation rate problem in digital audio/video systems*, in Proc. 2nd Israel Symp. on Theory of Computing and Systems, 1993, pp. 43–48.
[36] L. Shafer and W. Steiger, *Randomizing optimal geometric algorithms*, in Proc. 5th Canadian Conf. on Computational Geometry, Waterloo, Ontario, 1993, pp. 133–138.
[37] M. Sharir and S. Toledo, *Extremal polygon containment problems*, Comput. Geom. Theory Appl., 4 (1994), pp. 99–118.
[38] L. Valiant, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

Introduction to Special Section on Quantum Computation

The rapid evolution of computers in the half century since their invention has resulted in dramatically smaller and faster computers. However, from a computational point of view, all these computers look alike; for example, they are built out of simple logic gates. A fundamental thesis of computer science---the modern form of the Church--Turing thesis---asserts that this is inevitable in a deep sense. Any computer can be simulated with at most a polynomial factor slowdown by a probabilistic Turing machine. Quantum computation poses the first credible challenge to this thesis. It goes back to a suggestion by Feynman [4], who pointed out that there appears to be no efficient way of simulating a quantum mechanical system on a computer, and suggested that, perhaps, a computer based on quantum physical principles might be able to carry out the simulation efficiently. Two formal models for quantum computers---the quantum Turing machine [2] and quantum computational networks [3] ---were defined by Deutsch.

The first three papers in this issue describe efficient quantum algorithms for computational tasks that we do not know how to solve classically. In "Quantum Complexity Theory," Bernstein and Vazirani give the first formal evidence that quantum computers violate the modern form of the Church--Turing thesis. They show that a certain problem---the recursive Fourier sampling problem---can be solved in polynomial time on a quantum Turing machine, but relative to an oracle, requires superpolynomial time on a classical probabilistic Turing machine. Simon, in the paper "On the Power of Quantum Computation" introduces a fundamental projection technique and uses it to design an efficient quantum algorithm to determine whether a certain type of function is 2-1 or 1-1. He further shows that, relative to an oracle, this problem requires exponential time on a classical probabilistic Turing machine. In the paper "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," Shor gives remarkable polynomial time quantum algorithms for two of the most famous problems in computer science: factoring and discrete log. Since the computational hardness of these problems is the basis of several famous cryptosystems, Shor's paper very dramatically underlines the power of quantum computers.

To understand the computational power of quantum computers, it is helpful to consider a quantum mechanical system of n particles, each of which can be in one of two states, labeled $|0\rangle$ and $|1\rangle$. If this were a classical system, then its instantaneous state could be described by n bits. However, in quantum physics, the system is allowed to be in a linear superposition of configurations, and indeed the instantaneous state of the system is described by a unit vector in the 2^n dimensional vector space, whose basis vectors correspond to all the 2^n classical configurations. Therefore, to describe the instantaneous state of the system, we must specify 2^n complex numbers. Nature must update 2^n complex numbers at each instant to evolve the system in time. This is an extraordinary amount of effort, since even for $n = 200$, 2^n is larger than estimates of the number of elementary particles in the visible universe.

Nonetheless, there are limits to the power of quantum computers. In "Strengths and Weaknesses of Quantum Computing," Bennett, Bernstein, Brassard, and Vazirani show that, relative to a random oracle, with probability 1, the class *NP* cannot be solved on a quantum Turing machine in time $o(2^{n/2})$. This bound is tight, since recent work of Grover [5] has shown how to accept any language in *NP* in time $O(2^{n/2})$ on a quantum Turing machine.

Quantum computers are necessarily time reversible. Indeed, Bennett's work [1] on reversible computation inspired early work on quantum computation that preceded Feynman's paper [4]. The reversibility requirement makes it quite complex to implement even basic computational primitives such as looping or composition. In "Quantum Complexity Theory," Bernstein and Vazirani show how to implement quantum programming primitives and give a construction for an efficient universal quantum Turing machine. The structure of the universal quantum Turing machine is quite simple: it consists of a deterministic Turing machine with a single "quantum coin flip." In "Quantum Computability," Adleman, DeMarrais, and Huang greatly simplify this further by showing that a very simple type of coin flip is sufficient---a rotation by an angle θ such that $\sin\theta = 3/5$.

Making quantum computers robust against noise and decoherence is an important and challenging problem. In "Stabilization of Quantum Computations by Symmetrization," Barenco, Berthiaume, Deutsch, Ekert, Jozsa, and Macchiavello show how to use the quantum watchdog effect to stabilize a quantum computation against noise. Their method is based on running several copies of the quantum computer in parallel and projecting its state into the symmetric subspace at frequent intervals. They show that the quantum watchdog effect results in the suppression of errors that lie outside the symmetric subspace.

Quantum computation touches upon the foundations of both computer science and quantum physics. It is not unlikely that the issues raised by quantum computation will stimulate further research into the foundations of quantum physics.

I wish to express my gratitude to several people who made this special section possible. Oded Goldreich acted as editor for two of the papers in the issue and dealt with them with his characteristic efficiency and judgment. The editorial staff at SIAM, most notably Lisa Dougherty, Beth Gallagher, Deidre Wunderlich, and Sam Young, were extremely helpful, patient, and resourceful. Finally, I would like to thank a number of referees whose careful and timely reviews were critical to putting together this issue.

Umesh Vazirani

References

[1] C.H. Bennett, *Logical reversibility of computation*, IBM J. Res. Develop., 17 (1973), pp. 525--532.

[2] D. Deutsch, *Quantum theory, the Church-Turing principle and the universal quantum computer*, Proc. Roy. Soc. London Ser. A., 400 (1985), pp. 97--117.

[3] D. Deutsch, *Quantum computational networks*, Proc. Roy. Soc. London Ser. A, 425 (1989), pp. 73--90.

[4] R. Feyman, *Simulating physics with computers*, Internat. J. Theoret. Phys., 21 (1982), pp. 467--488.

[5] L. Grover, *Searching for a Needle in a Haystack---A Fast Mechanical Algorithm*, manuscript, 1995.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1411–1473, October 1997

007

QUANTUM COMPLEXITY THEORY*

ETHAN BERNSTEIN† AND UMESH VAZIRANI‡

Abstract. In this paper we study quantum computation from a complexity theoretic viewpoint. Our first result is the existence of an efficient universal quantum Turing machine in Deutsch's model of a quantum Turing machine (QTM) [Proc. Roy. Soc. London Ser. A, 400 (1985), pp. 97–117]. This construction is substantially more complicated than the corresponding construction for classical Turing machines (TMs); in fact, even simple primitives such as looping, branching, and composition are not straightforward in the context of quantum Turing machines. We establish how these familiar primitives can be implemented and introduce some new, purely quantum mechanical primitives, such as changing the computational basis and carrying out an arbitrary unitary transformation of polynomially bounded dimension.

We also consider the precision to which the transition amplitudes of a quantum Turing machine need to be specified. We prove that $O(\log T)$ bits of precision suffice to support a T step computation. This justifies the claim that the quantum Turing machine model should be regarded as a discrete model of computation and not an analog one.

We give the first formal evidence that quantum Turing machines violate the modern (complexity theoretic) formulation of the Church–Turing thesis. We show the existence of a problem, relative to an oracle, that can be solved in polynomial time on a quantum Turing machine, but requires superpolynomial time on a bounded-error probabilistic Turing machine, and thus not in the class **BPP**. The class **BQP** of languages that are efficiently decidable (with small error-probability) on a quantum Turing machine satisfies $\mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{P}^{\sharp\mathbf{P}}$. Therefore, there is no possibility of giving a mathematical proof that quantum Turing machines are more powerful than classical probabilistic Turing machines (in the unrelativized setting) unless there is a major breakthrough in complexity theory.

Key words. quantum computation, quantum Turing machines, reversibility, quantum polynomial time, Fourier sampling, universal quantum Turing machine

AMS subject classifications. 68Q05, 68Q15, 03D10, 03D15

PII. S0097539796300921

1. Introduction.

Just as the theory of computability has its foundations in the Church–Turing thesis, computational complexity theory rests upon a modern strengthening of this thesis, which asserts that any "reasonable" model of computation can be *efficiently* simulated on a probabilistic Turing machine (an efficient simulation is one whose running time is bounded by some polynomial in the running time of the simulated machine). Here, we take reasonable to mean in principle physically realizable. Some models of computation, though interesting for other reasons, do not meet this criterion. For example, it is clear that computers that operate on arbitrary length words in unit time or that exactly compute with infinite precision real numbers are not realizable. It has been argued that the TM (actually, the polynomial time equivalent cellular automaton model) is the inevitable choice once we assume that we can implement only finite precision computational primitives. Given the widespread belief that $\mathbf{NP} \not\subseteq \mathbf{BPP}$, this would seem to put a wide range of im-

*Received by the editors March 21, 1996; accepted for publication (in revised form) December 2, 1996. A preliminary version of this paper appeared in *Proc. 25th Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, NY, 1993, pp. 11–20.
http://www.siam.org/journals/sicomp/26-5/30092.html

†Microsoft Corporation, One Microsoft Way, Redmond, WA 98052 (ethanb@microsoft.com). The work of this author was supported by NSF grant CCR-9310214.

‡Computer Science Division, University of California, Berkeley, CA 94720 (vazirani@cs.berkeley.edu). The work of this author was supported by NSF grant CCR-9310214.

portant computational problems (the **NP**-hard problems) well beyond the capability of computers.

However, the TM fails to capture all physically realizable computing devices for a fundamental reason: the TM is based on a classical physics model of the universe, whereas current physical theory asserts that the universe is quantum physical. Can we get inherently new kinds of (discrete) computing devices based on quantum physics? Early work on the computational possibilities of quantum physics [6] asked the opposite question: does quantum mechanics' insistence on unitary evolution restrict the class of efficiently computable problems? They concluded that as far as deterministic computation is concerned, the only additional constraint imposed by quantum mechanics is that the computation must be reversible, and therefore by Bennett's [7] work it follows that quantum computers are at least as powerful as classical computers. The issue of the extra computational power of quantum mechanics over probabilistic computers was first raised by Feynman [25] in 1982. In that paper, Feynman pointed out a very curious problem: the natural simulation of a quantum physical system on a probabilistic TM requires an exponential slowdown. Moreover, it is unclear how to carry out the simulation more efficiently. In view of Feynman's observation, we must re-examine the foundations of computational complexity theory, and the complexity-theoretic form of the Church–Turing thesis, and study the computational power of computing devices based on quantum physics.

A precise model of a quantum physical computer—hereafter referred to as the QTM—was formulated by Deutsch [20]. There are two ways of thinking about quantum computers. One way that may appeal to computer scientists is to think of a quantum TM as a quantum physical analogue of a probabilistic TM—it has an infinite tape and a transition function, and the actions of the machine are local and completely specified by this transition function. Unlike probabilistic TMs, QTMs allow branching with complex "probability amplitudes" but impose the further requirement that the machine's evolution be time reversible. This view is elaborated in section 3.2. Another way is to view a quantum computer as effecting a transformation in a space of complex superpositions of configurations. Quantum physics requires that this transformation be unitary. A quantum algorithm may then be regarded as the decomposition of a unitary transformation into a product of unitary transformations, each of which makes only simple local changes. This view is elaborated on in section 3.3. Both formulations play an important role in the study of quantum computation.

One important concern is whether QTMs are really analog devices, since they involve complex transition amplitudes. It is instructive to examine the analogous question for probabilistic TMs. There, one might worry that probabilistic machines are not discrete and therefore not "reasonable," since they allow transition probabilities to be real numbers. However, there is extensive work showing that probabilistic computation can be carried out in a such a way that it is so insensitive to the transition probabilities that they can be allowed to vary arbitrarily in a large range [34, 44, 47]. In this paper, we show in a similar sense that QTMs are discrete devices: the transition amplitudes need only be accurate to $O(\log T)$ bits of precision to support T steps of computation. As Lipton [30] pointed out, it is crucial that the number of bits is $O(\log T)$ and not $O(T)$ (as it was in an early version of this paper), since k bits of precision require pinning down the transition amplitude to one part in 2^k. Since the transition amplitude is some physical quantity such as the angle of a polarizer or the

length of a π pulse, we must not assume that we can specify it to better than one part in some polynomial in T, and therefore the precision must be $O(\log T)$.

Another basic question one may ask is whether it is possible to define the notion of a general purpose quantum computer. In the classical case, this question is answered affirmatively by showing that there is an efficient universal TM. In this paper, we prove that there is an efficient QTM. When given as input the specification of an arbitrary QTM M, an input x to M, a time bound T, and an accuracy ϵ, the universal machine produces a superposition whose Euclidean distance from the time T superposition of M on x is at most ϵ. Moreover, the simulation time is bounded by a polynomial in T, $|x|$, and $\frac{1}{\epsilon}$. Deutsch [20] gave a different construction of a universal QTM. The simulation overhead in Deutsch's construction is exponential in T (the issue Deutsch was interested in was computability, not computational complexity). The structure of the efficient universal QTM constructed in this paper is very simple. It is just a deterministic TM with a single type of quantum operation—a quantum coin flip (an operation that performs a rotation on a single bit). The existence of this simple universal QTM has a bearing on the physical realizability of QTMs in general, since it establishes that it is sufficient to physically realize a simple quantum operation on a single bit (in addition to maintaining coherence and carrying out deterministic operations, of course). Adleman, DeMarrais, and Huang [1] and Solovay and Yao [40] have further clarified this point by showing that quantum coin flips with amplitudes $\frac{3}{5}$ and $\frac{4}{5}$ are sufficient for universal quantum computation.

Quantum computation is necessarily time reversible, since quantum physics requires unitary evolution. This makes it quite complicated to correctly implement even simple primitives such as looping, branching, and composition. These are described in section 4.2. In addition, we also require programming primitives, such as changing the computational basis, which are purely quantum mechanical. These are described in section 5.1. Another important primitive is the ability to carry out any specified unitary transformation of polynomial dimension to a specified degree of accuracy. In section 6 we show how to build a QTM that implements this primitive. Finally, all these pieces are put together in section 7 to construct the universal QTM.

We can still ask whether the QTM is the most general model for a computing device based on quantum physics. One approach to arguing affirmatively is to consider various other reasonable models and to show that the QTM can efficiently simulate each of them. An earlier version of this work [11] left open the question of whether standard variants of a QTM, such as machines with multiple tapes or with modified tape access, are more powerful than the basic model. Yao [46] showed that these models are polynomially equivalent to the basic model, as are quantum circuits (which were introduced in [21]). The efficiency of Yao's simulation has been improved in [10] to show that the simulation overhead is a polynomial with degree independent of the number of tapes. Arguably, the full computational power of quantum physics for discrete systems is captured by the quantum analogue of a cellular automaton. It is still an open question whether a quantum cellular automaton might be more powerful than a QTM (there is also an issue about the correct definition of a quantum cellular automaton). The difficulty has to do with decomposing a unitary transformation that represents many overlapping sites of activity into a product of simple, local unitary transformations. This problem has been solved in the special case of linearly bounded quantum cellular automata [24, 45].

Finally, several researchers have explored the computational power of QTMs. Early work by Deutsch and Jozsa [22] showed how to exploit some inherently quan-

tum mechanical features of QTMs. Their results, in conjunction with subsequent results by Berthiaume and Brassard [12, 13], established the existence of oracles under which there are computational problems that QTMs can solve in polynomial time with certainty, whereas if we require a classical probabilistic TM to produce the correct answer with certainty, then it must take exponential time on some inputs. On the other hand, these computational problems are in **BPP**—the class of problems that can be solved in polynomial time by probabilistic TMs that are allowed to give the wrong answer with small probability. Since **BPP** is widely considered the class of efficiently computable problems, these results left open the question of whether quantum computers are more powerful than classical computers.

In this paper, we give the first formal evidence that quantum Turing machines violate the modern form of the Church–Turing thesis by showing that, relative to an oracle, there is a problem that can be solved in polynomial time on a quantum Turing machine, but cannot be solved in $n^{o(\log n)}$ time on a probabilistic Turing machine with any fixed error probability $< 1/2$. A detailed discussion about the implications of these oracle results is in the introduction of section 8.4. Simon [39] subsequently strengthened our result in the time parameter by proving the existence of an oracle relative to which a certain problem can be solved in polynomial time on a quantum Turing machine, but cannot be solved in less than $2^{n/2}$ steps on a probabilistic Turing machine (Simon's problem is in $\mathbf{NP} \cap \mathbf{co}$–$\mathbf{NP}$ and therefore does not address the nondeterminism issue). More importantly, Simon's paper also introduced an important new technique which was one of the ingredients in a remarkable result proved subsequently by Shor [37]. Shor gave polynomial time quantum algorithms for the factoring and discrete log problems. These two problems have been well studied, and their presumed intractability forms the basis of much of modern cryptography. These results have injected a greater sense of urgency into the actual implementation of a quantum computer. The class **BQP** of languages that are efficiently decidable (with small error-probability) on a quantum Turing machine satisfies $\mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{P}^{\#\mathbf{P}}$. This rules out the possibility of giving a mathematical proof that quantum Turing machines are more powerful than classical probabilistic Turing machines (in the unrelativized setting) unless there is a major breakthrough in complexity theory.

It is natural to ask whether QTMs can solve every problem in NP in polynomial time. Bennett, Bernstein, Brassard, and Vazirani [9] give evidence showing the limitations of QTMs. They show that relative to an oracle chosen uniformly at random, with probability 1, the class **NP** cannot be solved on a QTM in time $o(2^{n/2})$. They also show that relative to a permutation oracle chosen uniformly at random, with probability 1, the class $\mathbf{NP} \cap \mathbf{co}$–$\mathbf{NP}$ cannot be solved on a QTM in time $o(2^{n/3})$. The former bound is tight since recent work of Grover [28] shows how to accept the class **NP** relative to any oracle on a quantum computer in time $O(2^{n/2})$.

Several designs have been proposed for realizing quantum computers [17, 23, 31]. A number of authors have argued that there are fundamental problems in building quantum computers, most notably the effects of the decoherence of quantum superpositions or the entanglement of the system with the environment [14, 18, 29, 35, 42]. Very recently, there has been a sequence of important results showing how to implement quantum error-correcting codes and also how to use these codes to make quantum algorithms (quite) robust against the effects of decoherence [16, 38].

Quantum computation touches upon the foundations of both computer science and quantum physics. The nature of quantum physics was clarified by the Einstein–Podolsky–Rosen paradox and Bell's inequalities (discussed in [25]), which demonstrate

the difference between its statistical properties and those of any "classical" model. The computational differences between quantum and classical physics are if anything more striking and can be expected to offer new insights into the nature of quantum physics. For example, one might naively argue that it is impossible to experimentally verify the exponentially large size of the Hilbert space associated with a discrete quantum system, since any observation leads to a collapse of its superposition. However, an experiment demonstrating the exponential speedup offered by quantum computation over classical computation would establish that something like the exponentially large Hilbert space must exist. Finally, it is important, as Feynman pointed out [25], to clarify the computational overhead required to simulate a quantum mechanical system. The simple form of the universal QTM constructed here—the fact that it has only a single nontrivial quantum operation defined on a single bit—suggests that even very simple quantum mechanical systems are capable of universal quantum computation and are therefore hard to simulate on classical computers.

This paper is organized as follows. Section 2 introduces some of the mathematical machinery and notation we will use. In section 3 we introduce the QTM as a natural extension of classical probabilistic TMs. We also show that QTMs need not be specified with an unreasonable amount of precision. In sections 4 and 5 we demonstrate the basic constructions which will allow us to build up large, complicated QTMs in subsequent sections. Many actions which are quite easy for classical machines, such as completing a partially specified machine, running one machine after another, or repeating the operation of a machine a given number of times, will require nontrivial constructions for QTMs. In section 6, we show how to build a single QTM which can carry out any unitary transformation which is provided as input. Then, in section 7, we use this simulation of unitary transformations to build a universal quantum computer. Finally, in section 8 we give our results, both positive and negative, on the power of QTMs.

2. Preliminaries. Let $\mathbf{C}$ denote the field of complex numbers. For $\alpha \in \mathbf{C}$, we denote by α^* its complex conjugate.

Let V be a vector space over $\mathbf{C}$. An inner product over V is a complex function $(\cdot,\cdot)$ defined on $V \times V$ which satisfies the following.

1. $\forall x \in V, (x,x) \geq 0$. Moreover $(x,x) = 0$ iff $x = 0$.
2. $\forall x, y, z \in V, (\alpha x + \beta y, z) = \alpha(x,z) + \beta(y,z)$.
3. $\forall x, y \in V, (x,y) = (y,x)^*$.

The inner product yields a norm given by $\|x\| = (x,x)^{1/2}$. In addition to the triangle inequality $\|x+y\| \leq \|x\| + \|y\|$, the norm also satisfies the Schwarz inequality $\|(x,y)\| \leq \|x\|\|y\|$.

An *inner-product space* is a vector space V together with an inner product $(\cdot,\cdot)$.

An inner-product space $\mathcal{H}$ over $\mathbf{C}$ is a *Hilbert space* if it is complete under the induced norm, where $\mathcal{H}$ is complete if every Cauchy sequence converges. For example, if $\{x_n\}$ is a sequence with $x_n \in \mathcal{H}$ such that $\lim_{n,m\to\infty} \|x_n - x_m\| = 0$, then there is an x in $\mathcal{H}$ with $\lim_{n\to\infty} \|x_n - x\| = 0$.

Given any inner-product space V, each vector $x \in V$ defines a linear functional $x^* : V \to \mathbf{C}$, where $x^*(y) = (x,y)$. The set of such linear functionals is also an inner-product space and will be referred to as the *vector dual* of V and denoted V^*. In the case that V is a Hilbert space, V^* is called the *dual* of V and is the set of all continuous linear functionals on V, and the dual space V^* is also a Hilbert space.

In Dirac's notation, a vector from an inner-product space V is identified using the "ket" notation $|\ \rangle$, with some symbol(s) placed inside to distinguish that vector

from all others. We denote elements of the dual space using the "bra" notation $\langle\ |$. Thus the dual of $|\phi\rangle$ is $\langle\phi|$, and the inner product of vectors $|\psi\rangle$ and $|\phi\rangle$, which is the same as the result of applying functional $\langle\psi|$ to the vector $|\phi\rangle$, is denoted by $\langle\psi|\phi\rangle$.

Let U be a linear operator on V. In Dirac's notation, we denote the result of applying U to $|\phi\rangle$ as $U|\phi\rangle$. U also acts as a linear operator on the dual space V^* mapping each linear functional $\langle\phi|$ of the dual space to the linear functional which applies U followed by $\langle\phi|$. We denote the result of applying U to $\langle\phi|$ by $\langle\phi|U$.

For any inner-product space V, we can consider the usual vector space basis or Hamel basis $\{|\theta_i\rangle\}_{i\in I}$. Every vector $|\phi\rangle \in V$ can be expressed as a finite linear combination of basis vectors. In the case that $\||\theta_i\rangle\| = 1$ and $(\langle\theta_i|, |\theta_j\rangle) = 0$ for $i \neq j$, we refer to the basis as an orthonormal basis. With respect to an orthonormal basis, we can write each vector $|\phi\rangle \in V$ as $|\phi\rangle = \sum_{i\in I} \alpha_i|\theta_i\rangle$, where $\alpha_i = \langle\theta_i|\phi\rangle$. Similarly each dual vector $\langle\phi| \in V^*$ can be written as $\langle\phi| = \sum_{i\in I} \beta_i\langle\theta_i|$, where $\alpha_i = \langle\phi|\theta_i\rangle$. Thus each element $|\phi\rangle \in V$ can be thought of as a column vector of the α_i's, and each element $\langle\phi| \in V$ can be thought of as a row vector of the β_i's. Similarly each linear operator U may be represented by the set of *matrix elements* $\{\langle\theta_i|U|\theta_j\rangle\}_{i,j\in I}$, arranged in a "square" matrix with rows and columns both indexed by I. Then, the "column" of U with index i is the vector $U|\theta_i\rangle$, and the "row" of U with index i is the dual vector $\langle\theta_i|U$.

For a Hilbert space $\mathcal{H}$, $\{|\theta_i\rangle\}_{i\in I}$ is a *Hilbert space basis* for $\mathcal{H}$ if it is a maximal set of orthonormal vectors in $\mathcal{H}$. Every vector $|\phi\rangle \in \mathcal{H}$ can be expressed as the limit of a sequence of vectors, each of which can be expressed as a finite linear combination of basis vectors.

Given a linear operator U in an inner product space, if there is a linear operator U^* which satisfies $\langle U^*\phi|\psi\rangle = \langle\phi|U\psi\rangle$ for all ϕ, ψ, then U^* is called the *adjoint* or *Hermitian conjugate* of U. If a linear operator in an inner product space has an adjoint, it is unique. The adjoint of a linear operator in a Hilbert space or in a finite-dimensional inner product space always exists. It is easy to see that if the adjoints of U_1 and U_2 exist, then $(U_1 + U_2)^* = U_1^* + U_2^*$ and $(U_1U_2)^* = U_2^*U_1^*$. An operator U is called *Hermitian* or *self-adjoint* if it is its own adjoint ($U^* = U$). The linear operator U is called *unitary* if its adjoint exists and satisfies $U^*U = UU^* = I$.

If we represent linear operators as "square" matrices indexed over an orthonormal basis, then U^* is represented by the conjugate transpose of U. So, in Dirac's notation we have the convenient identity $\langle\phi|U^*|\psi\rangle = (\langle\psi|U|\phi\rangle)^*$.

Recall that if the inner-product space V is the tensor product of two inner-product spaces V_1, V_2, then for each pair of vectors $|\phi_1\rangle \in V_1, |\phi_2\rangle \in V_2$ there is an associated tensor product $|\phi_1\rangle \otimes |\phi_2\rangle$ in V. In Dirac's notation, we denote $|\phi_1\rangle \otimes |\phi_2\rangle$ as $|\phi_1\rangle|\phi_2\rangle$.

The *norm* of U is defined as $\|U\| = \sup_{\||x\rangle\|=1} \|U|x\rangle\|$. A linear operator is called *bounded* if $\|U\|$ is finite. We will freely use the following standard facts about bounded linear operators:

$$\text{if } U^* \text{ exists then } \|U^*\| = \|U\|, \tag{2.1}$$

$$\|U_1U_2\| \leq \|U_1\|\|U_2\|, \tag{2.2}$$

$$\|U_1\| - \|U_2\| \leq \|U_1 + U_2\| \leq \|U_1\| + \|U_2\|. \tag{2.3}$$

Notice that a unitary operator U must satisfy $\|U\| = 1$. We will often use the following fact which tells us that if we approximate a series of unitary transformations with other unitary transformations, the error increases only additively.

FACT 2.1. *If U_1, U_1', U_2, U_2' are unitary transformations on an inner-product space, then*

$$\|U_1'U_2' - U_1U_2\| \;\leq\; \|U_1' - U_1\| + \|U_2' - U_2\|.$$

This fact follows from statements (2.3) and (2.2) above, since

$$\begin{aligned} \|U_1'U_2' - U_1U_2\| &\leq \|U_1'U_2' - U_1U_2'\| \;+\; \|U_1U_2' - U_1U_2\| \\ &\leq \|U_1' - U_1\|\|U_2'\| \;+\; \|U_1\|\|U_2' - U_2\|. \end{aligned}$$

2.1. Miscellaneous notation. If d is a direction $\in \{L, R\}$, then $\bar{d}$ is the opposite of d.

Given two probability distributions $\mathcal{P}_1$ and $\mathcal{P}_2$ over the same domain I, the *total variation distance* between $\mathcal{P}_1$ and $\mathcal{P}_2$ is equal to $\frac{1}{2}\sum_{i\in I}|\mathcal{P}_1(i) - \mathcal{P}_2(i)|$.

We will refer to the cardinality of a set S as $\mathrm{card}(S)$ and the length of a string x as $|x|$.

3. QTMs.

3.1. A physics-like view of randomized computation. Before we formally define a QTM, we introduce the necessary terminology in the familiar setting of probabilistic computation. As a bonus, we will be able to precisely locate the point of departure in the definition of a QTM.

Quantum mechanics makes a distinction between a system's evolution and its measurement. In the absence of measurement, the time evolution of a probabilistic TM can be described by a sequence of probability distributions. The distribution at each step gives the likelihood of each possible configuration of the machine. We can also think of the probabilistic TM as specifying an infinite-dimensional stochastic matrix[1] M whose rows and columns are indexed by configurations. Each column of this matrix gives the distribution resulting from the corresponding configuration after a single step of the machine. If we represent the probability distribution at one time step by a vector $|v\rangle$, then the distribution at the next step is given by the product $M|v\rangle$. In quantum physics terminology, we call the distribution at each step a "linear superposition" of configurations, and we call the coefficient of each configuration (its probability) its "amplitude." The stochastic matrix is referred to as the "time evolution operator."

Three comments are in order. First, not every stochastic matrix has an associated probabilistic TM. Stochastic matrices obtained from probabilistic TM are finitely specified and map each configuration by making only local changes to it. Second, the support of the superposition can be exponential in the running time of the machine. Third, we need to constrain the entries allowed in the transition function of our probabilistic TM. Otherwise, it is possible to smuggle hard-to-compute quantities into the transition amplitudes, for instance by letting the ith bit indicate whether the ith deterministic TM halts on a blank tape. A common restriction is to allow amplitudes only from the set $\{0, \frac{1}{2}, 1\}$. More generally, we might allow any real number in the interval $[0, 1]$ which can be computed by some deterministic algorithm to within any desired 2^{-n} in time polynomial in n. It is easily shown that the first possibility is computationally no more restrictive than the second.

[1] Recall that a matrix is stochastic if it has nonnegative real entries that sum to 1 in each column.

Returning to the evolution of the probabilistic TM, when we observe the machine after some number of steps, we do not see the linear superposition (probability distribution) but just a sample from it. If we "observe" the entire machine, then we see a configuration sampled at random according to the superposition. The same holds if we observe just a part of the machine. In this case, the superposition "collapses" to one that corresponds to the probability distribution conditioned on the value observed. By the linearity of the law of alternatives,[2] the mere act of making observations at times earlier than t does not change the probability for each outcome in an observation at time t. So, even though the unobserved superposition may have support that grows exponentially with running time, we need only keep track of a constant amount of information when simulating a probabilistic TM which is observed at each step. The computational possibilities of quantum physics arise out of the fact that observing a quantum system changes its later behavior.

3.2. Defining a QTM. Our model of randomized computation is already surprisingly close to Deutsch's model of a QTM. The major change that is required is that in quantum physics, the amplitudes in a system's linear superposition and the matrix elements in a system's time evolution operator are allowed to be complex numbers rather than just positive reals. When an observation is made, the probability associated with each configuration is not the configuration's amplitude in the superposition, but rather the squared magnitude of its amplitude. So instead of always having a linear superposition whose entries sum to 1, we will now always have a linear superposition whose Euclidean length is 1. This means that QTMs must be defined so that their time evolution preserves the Euclidean length of superpositions.

Making these changes to our model, we arrive at the following definitions.

For completeness, let us recall the definition of a deterministic TM. There are many standard variations to the definition of a deterministic TM, none of which affect their computational power. In this paper we will make our choices consistent with those in Deutsch's paper [20]: we consider TMs with a two-way infinite tape and a single tape head which must move left or right one square on each step. We also give standard definitions for interpreting the input, output, and running time of a deterministic TM. Note that although we usually restrict our discussion to TMs with tape head movements $\{L, R\}$, we will sometimes consider *generalized* TMs with tape head movements $\{L, N, R\}$ (where N means no head movement).

DEFINITION 3.1. *A* deterministic TM *is defined by a triplet* (Σ, Q, δ)*, where* Σ *is a finite alphabet with an identified blank symbol* $\#$*,* Q *is a finite set of states with an identified initial state* q_0 *and final state* $q_f \neq q_0$*, and* δ*, the* deterministic transition function*, is a function*

$$\delta \ : \ Q \times \Sigma \rightarrow \Sigma \times Q \times \{L, R\}.$$

The TM has a two-way infinite tape of cells indexed by $\mathbf{Z}$ *and a single read/write tape head that moves along the tape.*

A configuration *or* instantaneous description *of the TM is a complete description of the contents of the tape, the location of the tape head, and the state* $q \in Q$ *of the finite control. At any time only a finite number of tape cells may contain nonblank symbols.*

[2] The law of alternatives says exactly that the probability of an event A doesn't change if we first check to see whether event B has happened, $P(A) = P(A|B)P(B) + P(A|\overline{B})P(\overline{B})$.

For any configuration c of TM M, the successor configuration c' is defined by applying the transition function to the current state q and currently scanned symbol σ in the obvious way. We write $c \to_M c'$ to denote that c' follows from c in one step.

By convention, we require that the initial configuration *of M to satisfies the following conditions: the tape head is in cell 0, called the* start cell*, and the machine is in state q_0. An initial configuration has* input $x \in (\Sigma - \#)^*$ *if x is written on the tape in positions $0, 1, 2, \ldots$, and all other tape cells are blank. The TM* halts *on input x if it eventually enters the final state q_f. The number of steps a TM takes to halt on input x is its* running time *on input x. If a TM halts then its* output *is the string in Σ^* consisting of those tape contents from the leftmost nonblank symbol to the rightmost nonblank symbol, or the empty string if the entire tape is blank. A TM which halts on all inputs therefore computes a function from $(\Sigma - \#)^*$ to Σ^*.*

We now give a slightly modified version of the definition of a QTM provided by Deutsch [20]. As in the case of probabilistic TM, we must limit the transition amplitudes to efficiently computable numbers. Adleman, DeMarrais, and Huang [1] and Solovay and Yao [40] have separately shown that further restricting QTMs to rational amplitudes does not reduce their computational power. In fact, they have shown that the set of amplitudes $\{0, \pm\frac{3}{5}, \pm\frac{4}{5}, 1\}$ are sufficient to construct a universal QTM. We give a definition of the computation of a QTM with a particular string as input, but we defer discussing what it means for a QTM to halt or give output until section 3.5. Again, we will usually restrict our discussion to QTMs with tape head movements $\{L, R\}$ but will sometimes consider "generalized" QTMs with tape head movements $\{L, N, R\}$. As we pointed out in the introduction, unlike in the case of deterministic TMs, these choices do make a greater difference in the case of QTMs. This point is also discussed later in the paper.

DEFINITION 3.2. *Call $\tilde{\mathbf{C}}$ the set consisting of $\alpha \in \mathbf{C}$ such that there is a deterministic algorithm that computes the real and imaginary parts of α to within 2^{-n} in time polynomial in n.*

A QTM M is defined by a triplet (Σ, Q, δ), where Σ is a finite alphabet with an identified blank symbol $\#$, Q is a finite set of states with an identified initial state q_0 and final state $q_f \neq q_0$, and δ, the quantum transition function*, is a function*

$$\delta \quad : \quad Q \times \Sigma \to \tilde{\mathbf{C}}^{\Sigma \times Q \times \{L,R\}}.$$

The QTM has a two-way infinite tape of cells indexed by $\mathbf{Z}$ and a single read/write tape head that moves along the tape. We define configurations, initial configurations, and final configurations exactly as for deterministic TMs.

Let $\mathcal{S}$ be the inner-product space of finite complex linear combinations of configurations of M with the Euclidean norm. We call each element $\phi \in \mathcal{S}$ a superposition *of M. The QTM M defines a linear operator $U_M : \mathcal{S} \to \mathcal{S}$, called the* time evolution operator *of M, as follows: if M starts in configuration c with current state p and scanned symbol σ, then after one step M will be in superposition of configurations $\psi = \sum_i \alpha_i c_i$, where each nonzero α_i corresponds to a transition $\delta(p, \sigma, \tau, q, d)$, and c_i is the new configuration that results from applying this transition to c. Extending this map to the entire space $\mathcal{S}$ through linearity gives the linear time evolution operator U_M.*

Note that we defined $\mathcal{S}$ by giving an orthonormal basis for it: namely, the configurations of M. In terms of this standard basis, each superposition $\psi \in \mathcal{S}$ may be represented as a vector of complex numbers indexed by configurations. The time evolution operator U_M may be represented by the (countable dimensional) "square"

matrix with columns and rows indexed by configurations where the matrix element from column c and row c' gives the amplitude with which configuration c leads to configuration c' in a single step of M.

For convenience, we will overload notation and use the expression $\delta(p, \sigma, \tau, q, d)$ to denote the amplitude in $\delta(p, \sigma)$ of $|\tau\rangle|q\rangle|d\rangle$.

The next definition provides an extremely important condition that QTMs must satisfy to be consistent with quantum physics. We have introduced this condition in the form stated below for expository purposes. As we shall see later (in section 3.3), there are other equivalent formulations of this condition that are more familiar to quantum physics.

DEFINITION 3.3. *We will say that M is* well formed *if its time evolution operator U_M preserves Euclidean length.*

Wellformedness is a necessary condition for a QTM to be consistent with quantum physics. As we shall see in the next subsection, wellformedness is equivalent to unitary time evolution, which is a fundamental requirement of quantum physics.

Next, we define the rules for observing the QTM M. For those familiar with quantum mechanics, we should state that the definition below restricts the measurements to be in the computational basis of $\mathcal{S}$. This is because the actual basis in which the measurement is performed must be efficiently computable, and therefore we may, without loss of generality, perform the rotation to that basis during the computation itself.

DEFINITION 3.4. *When QTM M in superposition $\psi = \sum_i \alpha_i c_i$ is observed or measured, configuration c_i is seen with probability $|\alpha|^2$. Moreover, the superposition of M is updated to $\psi' = c_i$.*

We may also perform a partial measurement*, say only on the first cell of the tape. In this case, suppose that the first cell may contain the values* 0 *or* 1*, and suppose the superposition was $\psi = \sum_i \alpha 0_i c0_i + \sum_i \alpha 1_i c1_i$, where the $c0_i$ are those configurations that have a* 0 *in the first cell, and $c1_i$ are those configurations that have a* 1 *in the first cell. Measuring the first cell results in $\Pr[0] = \sum_i |\alpha 0_i|^2$. Moreover, if a* 0 *is observed, the new superposition is given by $\frac{1}{\sqrt{\Pr[0]}} \sum_i \alpha 0_i c0_i$, i.e., the part of the superposition consistent with the answer, with amplitudes scaled to give a unit vector.*

Note that the wellformedness condition on a QTM simply says that the time evolution operator of a QTM must satisfy the condition that in each successive superposition, the sum of the probabilities of all possible configurations must be 1.

Notice that a QTM differs from a classical TM in that the "user" has decisions beyond just choosing an input. A priori it is not clear whether multiple observations might increase the power of QTMs This point is discussed in more detail in [10], and there it is shown that one may assume without loss of generality that the QTM is only observed once. Therefore in this paper we shall make simplifying assumptions about the measurement of the final result of the QTM. The fact that these assumptions do not result in any loss of generality follows from the results in [10].

In general, the "output" of a QTM is a sample from a probability distribution. We can regard two QTMs as functionally equivalent, for practical purposes, if their output distributions are sufficiently close to each other. A formal definition of what it means for one QTM to simulate another is also given in [10]. As in the case of classical TMs, the formal definition is quite unwieldy. In the actual constructions, it will be easy to see in what sense they are simulations. Therefore we will not replicate the formal definitions from [10] here. We give a more informal definition below.

DEFINITION 3.5. *We say that QTM M' simulates M with slowdown f with accuracy ϵ if the following holds: let $\mathcal{D}$ be a distribution such that observing M on input x after T steps produces a sample from $\mathcal{D}$. Let $\mathcal{D}'$ be a distribution such that observing M' on input x after $f(T)$ steps produces a sample from $\mathcal{D}'$. Then we say that M' simulates M with accuracy ϵ if $|\mathcal{D} - \mathcal{D}'| \leq \epsilon$.*

We will sometimes find it convenient to measure the accuracy of a simulation by calculating the Euclidean distance between the target superposition and the superposition achieved by the simulation. The following shows that the variation distance between the resulting distributions is at most four times this Euclidean distance.

LEMMA 3.6. *Let $\phi, \psi \in \mathcal{S}$ such that $\|\phi\| = \|\psi\| = 1$, and $\|\phi - \psi\| \leq \epsilon$. Then the total variation distance between the probability distributions resulting from measurements of ϕ and ψ is at most 4ϵ.*

Proof. Let $\phi = \sum_i \alpha_i |i\rangle$ and $\psi = \sum_i \beta_i |i\rangle$. Observing ϕ gives each $|i\rangle$ with probability $|\alpha_i|^2$, while observing ψ gives each $|i\rangle$ with probability $|\beta_i|^2$. Let $\pi = \phi - \psi = \sum_i (\alpha_i - \beta_i)|i\rangle$. Then the latter probability $|\beta_i|^2$ can be expressed as

$$\beta_i \beta_i^* = (\alpha_i + \gamma_i)(\alpha_i + \gamma_i)^* = |\alpha_i|^2 + |\gamma_i|^2 + \alpha_i \gamma_i^* + \gamma_i \alpha_i^*.$$

Therefore, the total variation distance between these two distributions is at most

$$\sum_i \|\gamma_i\|^2 + |\alpha_i \gamma_i^*| + |\gamma_i \alpha_i^*| \leq \sum_i |\gamma_i|^2 + \langle \gamma || \alpha \rangle + \langle \alpha || \gamma \rangle \leq \epsilon^2 + 2\|\alpha\|\|\gamma\| \leq \epsilon^2 + 2\epsilon.$$

Finally, note that since we have unit superpositions, we must have $\epsilon \leq 2$. ∎

3.3. Quantum computing as a unitary transformation. In the preceding sections, we introduced QTMs as extensions of the notion of probabilistic TMs. We stated there that a QTM is well formed if it preserves the norm of the superpositions. In this section, we explore a different, and extremely useful, alternative view of QTMs: in terms of properties of the time evolution operator. We prove that a QTM is well formed iff its time evolution is unitary. Indeed unitary time evolution is a fundamental constraint imposed by quantum mechanics, and we chose to state the wellformedness condition in the last section mainly for expository purposes.

Understanding unitary evolution from an intuitive point of view is quite important to comprehending the computational possibilities of quantum mechanics. Let us explore this in the setting of a quantum mechanical system that consists of n parts, each of which can be in one of two states labeled $|0\rangle$ and $|1\rangle$ (these could be n particles, each with a spin state). If this were a classical system, then at any given instant it would be in a single configuration which could be described by n bits. However, in quantum physics, the system is allowed to be in a linear superposition of configurations, and indeed the instantaneous state of the system is described by a unit vector in the 2^n-dimensional vector space, whose basis vectors correspond to all the 2^n configurations. Therefore, to describe the instantaneous state of the system, we must specify 2^n complex numbers. The implications of this are quite extraordinary: even for a small system consisting of 200 particles, nature must keep track of 2^{200} complex numbers just to "remember" its instantaneous state. Moreover, it must update these numbers at each instant to evolve the system in time. This is an extravagant amount of effort, since 2^{200} is larger than the standard estimates on the number of particles in the visible universe. So if nature puts in such extravagant amounts of effort to evolve even a tiny system at the level of quantum mechanics, it would make sense that we should design our computers to take advantage of this.

However, unitary evolution and the rules for measurement in quantum mechanics place significant constraints on how these features can be exploited for computational purposes. One of the basic primitives that allows these features to be exploited while respecting the unitarity constraints is the discrete fourier transform—this is described in more detail in section 8.4. Here we consider some very simple cases: one interesting phenomenon supported by unitary evolution is the interference of computational paths. In a probabilistic computation the probability of moving from one configuration to another is the sum of the probabilities of each possible path from the former to the latter. The same is true of the probability amplitudes in quantum computation but not necessarily of the probabilities of observations. Consider, for example, applying the transformation

$$U = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

twice in sequence to the same tape cell which at first contains the symbol 0. If we observe the cell after the first application of U, we see either symbol with probability $\frac{1}{2}$. If we then observe after applying U a second time, the symbols are again equally likely. However, since U^2 is the identity, if we only observe at the end, we see a 0 with probability 1. So, even though there are two computational paths that lead from the initial 0 to a final symbol 1, they interfere destructively cancelling each other out. The two paths leading to a 0 interfere constructively so that even though both have probability $\frac{1}{4}$, when we observe twice we have probability 1 of reaching 0 if we observe only once. Such a boosting of probabilities, by an exponential factor, lies at the heart of the QTM's advantage over a probabilistic TM in solving the Fourier sampling problem.

Another constraint inherent in computation using unitary transformations is reversibility. We show in section 4.2 that for any QTM M there is a corresponding QTM M^R, whose time evolution operator is the conjugate transpose of the time evolution operator of M, and therefore undoes the actions of M. Sections 3.5 and 4.1 are devoted to defining the machinery to deal with this feature of QTMs.

We prove below in Appendix A that a QTM is well formed if its time evolution operator is unitary. This establishes that our definition (which did not mention unitary evolution for expository purposes) does satisfy the fundamental constraint imposed by quantum mechanics—unitary evolution. Actually one could still ask why this is consistent with quantum mechanics, since the space $\mathcal{S}$ of *finite* linear combinations of configurations is not a Hilbert space since it is not complete. To see that this doesn't present a problem, notice that $\mathcal{S}$ is a dense subset of the Hilbert space $\mathcal{H}$ of all (not just finite) linear combinations of configurations. Moreover, any unitary operator U on $\mathcal{S}$ has a unique extension $\hat{U}$ to $\mathcal{H}$; and $\hat{U}$ is unitary and its inverse is $\hat{U}^*$. The proof is quite simple. Let $\hat{U}$ and $\hat{U}^*$ be the continuous extensions of U and U^* to $\mathcal{H}$. Let $x, y \in \mathcal{H}$. Then there are sequences $\{x_n\}, \{y_n\} \in \mathcal{S}$ such that $x_n \to x$ and $y_n \to y$. Moreover, for all n, $(Ux_n, y_n) = (x_n, U^* y_n)$. Taking limits, we get that $(\hat{U}x, y) = (x, \hat{U}^* y)$, as desired.

As an aside, we should briefly mention that another resolution of this issue is achieved by following Feynman [26], who suggested that if we use a quantum mechanical system with Hamiltonian $U + U^*$, then the resulting system has a local, time invariant Hamiltonian. It is easy to probabilistically recover the computation of the original system from the computation of the new one.

It is interesting to note that the following theorem would not be true if we defined QTMs using a one-way infinite tape. In that case, the trivial QTM which always moves its tape head right would be well formed, but its time evolution would not be unitary since its start configuration could not be reached from any other configuration.

THEOREM A.5. *A QTM is well formed iff its time evolution operator is unitary.*

3.4. Precision required in a QTM. One important concern is whether QTMs are really analog devices, since they involve complex transition amplitudes. The issue here is how accurately these transition amplitudes must be specified to ensure that the correctness of the computation is not compromised. In an earlier version of this paper, we showed that $T^{O(1)}$ bits of precision are sufficient to correctly carry out T steps of computation to within accuracy ϵ for any constant ϵ. Lipton [30] pointed out that for the device to be regarded as a discrete device, we must require that its transition amplitudes be specified to at most one part in $T^{O(1)}$ (as opposed to accurate to within $T^{O(1)}$ bits of precision). This is because the transition amplitude represents some physical quantity such as the angle of a polarizer or the length of a π pulse, and we must not assume that we can specify it to better than one part in some polynomial in T, and therefore the number of bits of precision must be $O(\log T)$. This is exactly what we proved shortly after Lipton's observation, and we present that proof in this section.

The next theorem shows that because of the unitary time evolution errors in the superposition made over a sequence of steps will, in the worst case, only add.

THEOREM 3.7. *Let U be the time evolution operator of a QTM M and $T > 0$. If $|\phi_0\rangle, |\tilde{\phi}_0\rangle, \ldots, |\phi_T\rangle, |\tilde{\phi}_T\rangle$ are superpositions of U such that*

$$\| |\phi_i\rangle - |\tilde{\phi}_i\rangle \| \ \leq \ \epsilon,$$

$$|\phi_i\rangle \ = \ U|\tilde{\phi}_{i-1}\rangle,$$

then $\| |\tilde{\phi}_T\rangle - U^T|\phi_0\rangle \| \leq T\epsilon$.

Proof. Let $|\psi_i\rangle = |\tilde{\phi}_i\rangle - |\phi_i\rangle$. Then we have

$$|\tilde{\phi}_T\rangle \ = \ U^T|\phi_0\rangle + U^T|\psi_0\rangle + U^{T-1}|\psi_1\rangle + \cdots + |\psi_T\rangle$$

The theorem follows by the triangle inequality since U is unitary and $\| |\psi_i\rangle \| \leq \epsilon$. □

DEFINITION 3.8. *We say that QTMs M and M' are* ϵ-close *if they have the same state set and alphabet and if the difference between each pair of corresponding transition amplitudes has magnitude at most ϵ. Note that M and M' can be ϵ-close even if one or both are not well formed.*

The following theorem shows that two QTMs which are close in the above sense give rise to time evolution operators which are close to each other, even if the QTMs are not well formed. As a simple consequence, the time evolution operator of a QTM is always bounded, even if the QTM is not well formed.

THEOREM 3.9. *If QTMs M and M' with alphabet Σ and state set Q are ϵ-close, then the difference in their time evolutions has norm at most* 2 $\text{card}(\Sigma)\ \text{card}(Q)\epsilon$. *Moreover, this statement holds even if one or both of the machines are not well formed.*

Proof. Let QTMs M and M' with alphabet Σ and state set Q be given which are ϵ-close. Let U be the time evolution of M, and let U' be the time evolution of M'.

Now, consider any unit length superposition of configurations $|\phi\rangle = \sum_j \alpha_j |c_j\rangle$. Then we can express the difference in the machines' operations on $|\phi\rangle$ as follows:

$$U|\phi\rangle - U'|\phi\rangle = \sum_j \left(\sum_{i \in P(j)} (\lambda_{i,j} - \lambda'_{i,j})\alpha_i \right) |c_j\rangle,$$

where $P(j)$ is the set of i such that configuration c_i can lead to c_j in a single step of M or M' and where $\lambda_{i,j}$ and $\lambda'_{i,j}$ are the amplitudes with which c_i leads to c_j in M and M'.

Applying the triangle inequality and the fact that the square of the sum of n reals is at most n times the sum of their squares, we have

$$\begin{aligned} \|U|\phi\rangle - U'|\phi\rangle\|^2 &= \sum_j \left| \sum_{i \in P(j)} (\lambda_{i,j} - \lambda'_{i,j})\alpha_i \right|^2 \\ &\leq \sum_j 2 \text{ card}(\Sigma) \text{ card}(Q) \sum_{i \in P(j)} \left| (\lambda_{i,j} - \lambda'_{i,j})\alpha_i \right|^2. \end{aligned}$$

Then since M and M' are ϵ-close, we have

$$\begin{aligned} \sum_j 2 \text{ card}(\Sigma) \text{ card}(Q) \sum_{i \in P(j)} & \left| (\lambda_{i,j} - \lambda'_{i,j})\alpha_i \right|^2 \\ &= 2 \text{ card}(\Sigma) \text{ card}(Q) \sum_j \sum_{i \in P(j)} \left| \lambda_{i,j} - \lambda'_{i,j} \right|^2 |\alpha_i|^2 \\ &\leq 2 \text{ card}(\Sigma) \text{ card}(Q) \epsilon^2 \sum_j \sum_{i \in P(j)} |\alpha_i|^2. \end{aligned}$$

Finally since for any configuration c_j, there are at most $2 \text{ card}(\Sigma) \text{ card}(Q)$ configurations that can lead to c_j in a single step, we have

$$\begin{aligned} 2 \text{ card}(\Sigma) \text{ card}(Q) \epsilon^2 \sum_j \sum_{i \in P(j)} |\alpha_i|^2 &\leq 4 \text{ card}(\Sigma)^2 \text{ card}(Q)^2 \epsilon^2 \sum_i |\alpha_i|^2 \\ &= 4 \text{ card}(\Sigma)^2 \text{ card}(Q)^2 \epsilon^2. \end{aligned}$$

Therefore, for any unit length superposition $|\phi\rangle$

$$\|(U - U')|\phi\rangle\| \leq 2 \text{ card}(\Sigma) \text{ card}(Q) \epsilon. \qquad \square$$

The following corollary shows that $O(\log T)$ bits of precision are sufficient in the transition amplitudes to simulate T steps of a QTM to within accuracy ϵ for any constant ϵ.

COROLLARY 3.10. *Let $M = (\Sigma, Q, \delta)$ be a well-formed QTM, and let M' be a QTM which is $\frac{\epsilon}{24 \text{ card}(\Sigma) \text{ card}(Q) T}$-close to M, where $\epsilon > 0$. Then M' simulates M for time T with accuracy ϵ. Moreover, this statement holds even if M' is not well formed.*

Proof. Let $b = \frac{1}{24 \text{ card}(\Sigma) \text{ card}(Q) T}$. Without loss of generality, we further assume $\epsilon < \frac{1}{2}$.

Consider running M and M' with the same initial superposition. Since M is well formed, by Theorem A.5, its time evolution operator U is unitary. By Theorem 3.9 the time evolution operator of M', U' is within $\delta = \frac{\epsilon}{12T}$ of U.

Applying U' can always be expressed as applying U and then adding a perturbation of length at most δ times the length of the current superposition. So, the length of the superposition of U' at time t is at most $(1 + \delta)^t$. Since $\delta \leq \frac{1}{T}$, this length is at most e. Therefore, appealing to Theorem 3.7 above, the difference between the superpositions of M and M' at time T is a superposition of norm at most $3\delta T \leq \frac{\epsilon}{4}$. Finally, Lemma 3.6 tells us that observing M' at time T gives a sample from a distribution which is within total variation distance ϵ of the distributions sampled from by observing M at time T. $\square$

3.5. Input/output conventions for QTMs. Timing is crucial to the operation of a QTM, because computational paths can only interfere if they take the same number of time steps. Equally important are the position of the tape head and alignment of the tape contents. In this subsection, we introduce several input/output conventions on QTMs and deterministic TMs which will help us maintain these relationships while manipulating and combining machines.

We would like to think of our QTMs as finishing their computation when they reach the final state q_f. However, it is unclear how we should regard a machine that reaches a superposition in which some configurations are in state q_f but others are not. We try to avoid such difficulties by saying that a QTM halts on a particular input if it reaches a superposition consisting entirely of configurations in state q_f.

DEFINITION 3.11. *A* final configuration *of a QTM is any configuration in state q_f. If when QTM M is run with input x, at time T the superposition contains only final configurations, and at any time less than T the superposition contains no final configuration, then M* halts *with* running time T *on input x. The superposition of M at time T is called the* final superposition *of M run on input x. A polynomial-time QTM is a well-formed QTM, which on every input x halts in time polynomial in the length of x.*

We would like to define the output of a QTM which halts as the superposition of the tape contents of the configurations in the machine's final superposition. However, we must be careful to note the position of the tape head and the alignment relative to the start cell in each configuration since these details determine whether later paths interfere. Recall that the output string of a final configuration of a TM is its tape contents from the leftmost nonblank symbol to the rightmost nonblank symbol. This means that giving an output string leaves unspecified the alignment of this string on the tape and the location of the tape head to be identified. When describing the input/output behavior of a QTM we will sometimes describe this additional information. When we do not, the additional information will be clear from context. For example, we will often build machines in which all final configurations have the output string beginning in the start cell with the tape head scanning its first symbol.

DEFINITION 3.12. *A QTM is called* well behaved *if it halts on all input strings in a final superposition where each configuration has the tape head in the same cell. If this cell is always the start cell, we call the machine* stationary. *Similarly, a deterministic TM is called stationary if it halts on all inputs with its tape head back in the start cell.*

To simplify our constructions, we will often build QTMs and then combine them in simple ways, like running one after the other or iterating one a varying number of times. To do so we must be able to add new transitions into the initial state q_0 of a machine. However, since there may already be transitions into q_0, the resulting machine may not be reversible. But, we can certainly redirect the transitions out of the final state q_f of a reversible TM or a well-behaved QTM without affecting its behavior. Note that for a well-formed QTM, if q_f always leads back to q_0, then there can be no more transitions into q_0. In that case, redirecting the transitions out of q_f will allow us to add new ones into q_0 without violating reversibility. We will say that a machine with this property is in *normal form.* Note that a well-behaved QTM in normal form always halts before using any transition out of q_f and therefore also before using any transition into q_0. This means that altering these transitions will not alter the relevant part of the machine's computation. For simplicity, we arbitrarily

define normal form QTMs to step right and leave the tape unchanged as they go from state q_f to q_0.

DEFINITION 3.13. *A QTM or deterministic TM $M = (\Sigma, Q, \delta)$ is in* normal form *if*

$$\forall \sigma \in \Sigma \quad \delta(q_f, \sigma) = |\sigma\rangle|q_0\rangle|R\rangle.$$

We will need to consider QTMs with the special property that any particular state can be entered while the machine's tape head steps in only one direction. Though not all QTMs are "unidirectional," we will show that any QTM can be efficiently simulated by one that is. Unidirectionality will be a critical concept in reversing a QTM, in completing a partially described QTM, and in building our universal QTM. We further describe the advantages of unidirectional machines after Theorem 5.3 in section 5.2.

DEFINITION 3.14. *A QTM $M = (\Sigma, Q, \delta)$ is called* unidirectional *if each state can be entered from only one direction: in other words, if $\delta(p_1, \sigma_1, \tau_1, q, d_1)$ and $\delta(p_2, \sigma_2, \tau_2, q, d_2)$ are both nonzero, then $d_1 = d_2$.*

Finally, we will find it convenient to use the common tool of thinking of the tape of a QTM or deterministic TM as consisting of several tracks.

DEFINITION 3.15. *A* multitrack *TM with k tracks is a TM whose alphabet Σ is of the form $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_k$ with a special blank symbol $\#$ in each Σ_i so that the blank in Σ is $(\#, \ldots, \#)$. We specify the input by specifying the string on each "track" and optionally by specifying the alignment of the contents of the tracks. So, a TM run on input $x_1; x_2; \ldots; x_k \in \Pi_{i=1}^{k} (\Sigma_i - \#)^*$ is started in the (superposition consisting only of the) initial configuration with the nonblank portion of the ith coordinate of the tape containing the string x_i starting in the start cell. More generally, on input $x_1|y_1; x_2|y_1; \ldots; x_k|y_k$ with $x_i, y_i \in \Sigma_i^*$ the nonblank portion of the ith track is $x_i y_i$ aligned so that the first symbol of each y_i is in the start cell. Also, input $x_1; x_2; \ldots; x_k$ with $x_{l+1}, \ldots, x_k = \epsilon$ is abbreviated as $x_1; x_2; \ldots; x_l$.*

4. Programming a QTM. In this section we explore the fundamentals of building up a QTM from several simpler QTMs. Implementing basic programming primitives such as looping, branching, and reversing, a computation is straightforward for deterministic TMs. However, these constructions are more difficult for QTMs because one must be very careful to maintain reversibility. In fact, the same difficulties arise when building reversible deterministic TMs. However, building up reversible TMs out of simpler reversible TMs has never been necessary. This is because Bennett [7] showed how to efficiently simulate any deterministic TM with one which is reversible. So, one can build a reversible TM by first building the desired computation with a nonreversible machine and then by using Bennett's construction. None of the constructions in this section make any special use of the quantum nature of QTMs, and in fact all techniques used are the same as those required to make the analogous construction for reversible TMs.

We will show in this section that reversible TMs are a special case of QTMs. So, as Deutsch [20] noted, Bennett's result allows any desired deterministic computation to be carried out on a QTM. However, Bennett's result is not sufficient to allow us to use deterministic computations when building up QTMs, because the different computation paths of a QTM will only interfere properly provided that they take exactly the same number of steps. We will therefore carry out a modified version of Bennett's construction to show that any deterministic computation can be carried out

by a reversible TM whose running time depends only on the length of its input. Then, different computation paths of a QTM will take the same number of steps provided that they carry out the same deterministic computation on inputs of the same length.

4.1. Reversible TMs.

DEFINITION 4.1. *A* reversible TM *is a deterministic TM for which each configuration has at most one predecessor.*

Note that we have altered the definition of a reversible TM from the one used by Bennett [7, 8], so that our reversible TMs are a special case of our QTMs. First, we have restricted our reversible TM to move its head only left and right, instead of also allowing it to stay still. Second, we insist that the transition function δ be a complete function rather than a partial function. Finally, we consider only reversible TMs with a single tape, though Bennett worked with multitape machines.

THEOREM 4.2. *Any reversible TM is also a well-formed QTM.*

Proof. The transition function δ of a deterministic TM maps the current state and symbol to an update triple. If we think of it as instead giving the unit superposition with amplitude 1 for that triple and 0 elsewhere, then δ is also a quantum transition function and we have a QTM. The time evolution matrix corresponding to this QTM contains only the entries 1 and 0. Since Corollary B.2 proven below in Appendix B tells us that each configuration of a reversible TM has exactly one predecessor, this matrix must be a permutation matrix. If the TM is reversible then there must be at most one 1 in each row. Therefore, any superposition of configurations $\sum_i \alpha_i |c_i\rangle$ is mapped by this time evolution matrix to some other superposition of configurations $\sum_i \alpha_i |c'_i\rangle$. So, the time evolution preserves length, and the QTM is well formed. □

Previous work of Bennett shows that reversible machines can efficiently simulate deterministic TMs. Of course, if a deterministic TM computes a function which is not one-to-one, then no reversible machine can simulate it exactly. Bennett [7] showed that a generalized reversible TM can do the next best thing, which is to take any input x and compute $x; M(x)$, where $M(x)$ is the output of M on input x. He also showed that if a deterministic TM computes a function which is one-to-one, then there is a generalized reversible TM that computes the same function. For both constructions, he used a multitape TM and also suggested how the simulation could be carried out using only a single-tape machine. Morita, Shirasaki, and Gono [33] use Bennett's ideas and some further techniques to show that any deterministic TM can be simulated by a generalized reversible TM with a two symbol alphabet.

We will give a slightly different simulation of a deterministic TM with a reversible machine that preserves an important timing property.

First, we describe why timing is critical. In later sections, we will build QTMs with interesting and complex interference patterns. However, two computational paths can only interfere if they reach the same configuration at the same time. We will often want paths to interfere which run much of the same computation but with different inputs. We can only be sure they interfere if we know that these computations can be carried out in exactly the same running time. We therefore want to show that any function computable in deterministic polynomial time can be computed by a polynomial time reversible TM in such a way that the running time of the latter is determined entirely by the length of its input. Then, provided that all computation paths carry out the same deterministic algorithms on the inputs of the same length, they will all take exactly the same number of steps.

We prove the following theorem in Appendix B on page 1465 using ideas from the constructions of Bennett and Morita, Shirasaki, and Gono.

THEOREM 4.3 (synchronization theorem). *If f is a function mapping strings to strings which can be computed in deterministic polynomial time and such that the length of $f(x)$ depends only on the length of x, then there is a polynomial time, stationary, normal form reversible TM which given input x, produces output $x; f(x)$, and whose running time depends only on the length of x.*

If f is a function from strings to strings such that both f and f^{-1} can be computed in deterministic polynomial time and such that the length of $f(x)$ depends only on the length of x, then there is a polynomial time, stationary, normal form reversible TM which given input x, produces output $f(x)$, and whose running time depends only on the length of x.

4.2. Programming primitives. We now show how to carry out several programming primitives reversibly. The branching, reversal, and looping lemmas will be used frequently in subsequent sections.

The proofs of the following two lemmas are straightforward and are omitted. However, they will be quite useful as we build complicated machines, since they allow us to build a series of simpler machines while ignoring the contents of tracks not currently being used.

LEMMA 4.4. *Given any QTM (reversible TM) $M = (\Sigma, Q, \delta)$ and any set Σ', there is a QTM (reversible TM) $M' = (\Sigma \times \Sigma', Q, \delta')$ such that M' behaves exactly as M while leaving its second track unchanged.*

LEMMA 4.5. *Given any QTM (reversible TM) $M = (\Sigma_1 \times \cdots \times \Sigma_k, Q, \delta)$ and permutation $\pi : [1, k] \to [1, k]$, there is a QTM (reversible TM) $M' = (\Sigma_{\pi(1)} \times \cdots \times \Sigma_{\pi(k)}, Q, \delta')$ such that the M' behaves exactly as M except that its tracks are permuted according to π.*

The following two lemmas are also straightforward, but stating them separately makes Lemma 4.8 below easy to prove. The first deals with swapping transitions of states in a QTM. We can swap the outgoing transitions of states p_1 and p_2 for transition function δ by defining $\delta'(p_1, \sigma) = \delta(p_2, \sigma)$, $\delta'(p_2, \sigma) = \delta(p_1, \sigma)$, and $\delta'(p, \sigma) = \delta(p, \sigma)$ for $p \neq p_1, p_2$. Similarly, we can swap the incoming transitions of states q_1 and q_2 by defining $\delta'(p, \sigma, \tau, q_1, d) = \delta(p, \sigma, \tau, q_2, d)$, $\delta'(p, \sigma, \tau, q_2, d) = \delta(p, \sigma, \tau, q_1, d)$, and $\delta'(p, \sigma, \tau, q, d) = \delta(p, \sigma, \tau, q, d)$ for $q \neq q_1, q_2$.

LEMMA 4.6. *If M is a well-formed QTM (reversible TM), then swapping the incoming or outgoing transitions between a pair of states in M gives another well-formed QTM (reversible TM).*

LEMMA 4.7. *Let $M_1 = (\Sigma, Q_1, \delta_1)$ and $M_2 = (\Sigma, Q_2, \delta_2)$ be two well-formed QTMs (reversible TMs) with the same alphabet and disjoint state sets. Then the union of the two machines, $M = (\Sigma, Q_1 \cup Q_2, \delta_1 \cup \delta_2)$, with arbitrarily chosen start state $q_0 \in Q_1 \cup Q_2$, is also a well-formed QTM (reversible TM).*

Using the two preceding lemmas, we can insert one machine in place of a state in another. When we perform such an insertion, the computations of the two machines might disturb each other. However, sometimes this can easily be seen not to be the case. For example, in the insertion used by the dovetailing lemma below, we will insert one machine for the final state of a well-behaved QTM, so that the computation of the original machine has been completed before the inserted machine begins. In the rest of the insertions we carry out, the two machines will operate on different tracks, so the only possible disturbance involves the position of the tape head.

LEMMA 4.8. *If M_1 and M_2 are normal form QTMs (reversible TMs) with the same alphabet and q is a state of M_1, then there is a normal form QTM M which acts as M_1 except that each time it would enter state q, it instead runs machine M_2.*

Proof. Let M_1 and M_2 be as stated with initial and final states $q_{1,0}, q_{2,0}, q_{1,f}, q_{2,f}$, and with q a state of M_1.

Then we can construct the desired machine M as follows. First, take the union of M_1 and M_2 according to Lemma 4.7 on page 1428 and make the start state $q_{1,0}$ if $q \neq q_{1,0}$ and $q_{2,0}$ otherwise, and make the final state $q_{1,f}$ if $q \neq q_{1,f}$ and $q_{2,f}$ otherwise. Then, according to Lemma 4.6, swap the incoming transitions of q and $q_{2,0}$ and the outgoing transitions of q and $q_{2,f}$ to get the well-formed machine M.

Since M_1 is in normal form, the final state of M leads back to its initial state no matter whether q is the initial state of M_1, the final state of M_1, or neither. □

Next, we show how to take two machines and form a third by "dovetailing" one onto the end of the other. Notice that when we dovetail QTMs, the second QTM will be started with the final superposition of the first machine as its input superposition. If the second machine has differing running times for various strings in this superposition, then the dovetailed machine might not halt even though the two original machines were well behaved. Therefore, a QTM built by dovetailing two well-behaved QTMs may not itself be well behaved.

LEMMA 4.9 (dovetailing lemma). *If M_1 and M_2 are well-behaved, normal form QTMs (reversible TMs) with the same alphabet, then there is a normal form QTM (reversible TM) M which carries out the computation of M_1 followed by the computation of M_2.*

Proof. Let M_1 and M_2 be well-behaved, normal form QTMs (reversible TMs) with the same alphabet and with start states and final states $q_{1,0}, q_{2,0}, q_{1,f}, q_{2,f}$.

To construct M, we simply insert M_2 for the final state of M_1 using Lemma 4.8 on page 1428.

To complete the proof we need to show that M carries out the computation of M_1 followed by that of M_2.

To see this, first recall that since M_1 and M_2 are in normal form, the only transitions into $q_{1,0}$ and $q_{2,0}$ are from $q_{1,f}$ and $q_{2,f}$, respectively. This means that no transitions in M_1 have been changed except for those into or out of state $q_{1,f}$. Therefore, since M_1 is well behaved and does not prematurely enter state $q_{1,f}$, the machine M, when started in state $q_{1,0}$, will compute exactly as M_1 until M_1 would halt. At that point M will instead reach a superposition with all configurations in state $q_{2,0}$. Then, since no transitions in M_2 have been changed except for those into or out of $q_{2,f}$, M will proceed exactly as if M_2 had been started in the superposition of outputs computed by M_1. □

Now we show how to build a conditional branch around two existing QTMs or reversible TMs. The branching machine will run one of the two machines on its first track input, depending on its second track input. Since a TM can have only one final state, we must rejoin the two branches at the end. We can join reversibly if we write back out the bit that determined which machine was used. The construction will simply build a reversible TM that accomplishes the desired branching and rejoining and then insert the two machines for states in this branching machine.

LEMMA 4.10 (branching lemma). *If M_1 and M_2 are well-behaved, normal form QTMs (reversible TMs) with the same alphabet, then there is a well-behaved, normal form QTM (reversible TM) M such that if the second track is empty, M runs M_1 on its first track and leaves its second track empty, and if the second track has a* 1 *in the start cell (and all other cells blank), M runs M_2 on its first track and leaves the* 1 *where its tape head ends up. In either case, M takes exactly four more time steps than the appropriate M_i.*

Proof. Let M_1 and M_2 be well-behaved, normal form QTMs (reversible TMs) with the same alphabet.

Then we can construct the desired QTM as follows. We will show how to build a stationary, normal form reversible TM BR which always takes four time steps and leaves its input unchanged, always has a superposition consisting of a single configuration, and has two states q_1 and q_2 with the following properties. If BR is run with a 1 in the start cell and blanks elsewhere, then BR visits q_1 once with a blank tape and with its tape head in the start cell and doesn't visit q_2 at all. Similarly, if BR is run with a blank tape, then BR visits q_2 once with a blank tape and with its tape head in the start cell and doesn't visit q_1 at all. Then if we extend M_1 and M_2 to have a second track with the alphabet of BR, extend BR to have a first track with the common alphabet of M_1 and M_2, and insert M_1 for state q_1 and M_2 for state q_2 in BR, we will have the desired QTM M.

We complete the construction by exhibiting the reversible TM BR. The machine enters state q_1' or q_2' depending on whether the start cell contains a 1 and steps left and enters the corresponding q_1 or q_2 while stepping back right. Then it enters state q_3 while stepping left and state q_f while stepping back right.

So, we let BR have alphabet $\{\#, 1\}$, state set $\{q_0, q_1, q_1', q_2, q_2', q_3, q_f\}$, and transition function defined by the following table

	$\#$	1
q_0	$\#, q_2', L$	$\#, q_1', L$
q_1'	$\#, q_1, R$	
q_2'	$\#, q_2, R$	
q_1	$1, q_3, L$	
q_2	$\#, q_3, L$	
q_3	$\#, q_f, R$	
q_f	$\#, q_0, R$	$1, q_0, R$

It can be verified that the transition function of BR is one-to-one and that it can enter each state while moving in only one direction. Therefore, appealing to Corollary B.3 on page 1466 it can be completed to give a reversible TM. □

Finally, we show how to take a unidirectional QTM or reversible TM and build its reverse. Two computational paths of a QTM will interfere only if they reach configurations that do not differ in any way. This means that when building a QTM we must be careful to erase any information that might differ along paths which we want to interfere. We will therefore sometimes use this lemma when constructing a QTM to allow us to completely erase an intermediate computation.

Since the time evolution of a well-formed QTM is unitary, we could reverse a QTM by applying the unitary inverse of its time evolution. However, this transformation is not the time evolution of a QTM since it acts on a configuration by changing tape symbols to the left and right of the tape head. However, since each state in a unidirectional QTM can be entered while moving in only one direction, we can reverse the head movements one step ahead of the reversal of the tape and state. Then, the reversal will have its tape head in the proper cell each time a symbol must be changed.

DEFINITION 4.11. *If QTMs M_1 and M_2 have the same alphabet, then we say that* M_2 reverses *the computation of M_1 if identifying the final state of M_1 with the initial state of M_2 and the initial state of M_1 with the final state of M_2 gives the following. For any input x on which M_1 halts, if c_x and ϕ_x are the initial configuration and final superposition of M_1 on input x, then M_2 halts on initial superposition ϕ_x with final superposition consisting entirely of configuration c_x.*

LEMMA 4.12 (reversal lemma). *If M is a normal form, reversible TM or unidirectional QTM, then there is a normal form, reversible TM or QTM M' that reverses the computation of M while taking two extra time steps.*

Proof. We will prove the lemma for normal form, unidirectional QTMs, but the same argument proves the lemma for normal form reversible TMs.

Let $M = (\Sigma, Q, \delta)$ be a normal form, unidirectional QTM with initial and final states q_0 and q_f, and for each $q \in Q$ let d_q be the direction which M must move while entering state q. Then, we define a bijection π on the set of configurations of M as follows. For configuration c in state q, let $\pi(c)$ be the configuration derived from c by moving the tape head one square in direction $\bar{d}_q$ (the opposite of direction d_q). Since the set of superpositions of the form $|c\rangle$ give an orthonormal basis for the space of superpositions of M, we can extend π to act as a linear operator on this space of superpositions by defining $\pi|c\rangle = |\pi c\rangle$. It is easy to see that π is a unitary transformation on the space of superpositions on M.

Our new machine M' will have the same alphabet as M and state set given by Q together with new initial and final states q'_0 and q'_f. The following three statements suffice to prove that M' reverses the computation of M while taking two extra time steps.

1. If c is a final configuration of M and c' is the configuration c with state q_f replaced by state q'_0, then a single step of M' takes superposition $|c'\rangle$ to superposition $\pi(|c\rangle)$.

2. If a single step of M takes superposition $|\phi_1\rangle$ to superposition $|\phi_2\rangle$, where $|\phi_2\rangle$ has no support on a configuration in state q_0, then a single step of M' takes superposition $\pi(|\phi_2\rangle)$ to superposition $\pi(|\phi_1\rangle)$.

3. If c is an initial configuration of M and c' is the configuration c with state q_0 replaced by state q'_f, then a single step of M' takes superposition $\pi(|c\rangle)$ to superposition $|c'\rangle$.

To see this, let x be an input on which M halts, and let $|\phi_1\rangle, \ldots, |\phi_n\rangle$ be the sequence of superpositions of M on input x, so that $|\phi_1\rangle = |c_x\rangle$ where c_x is the initial superposition of M on x and $|\phi_n\rangle$ has support only on final configurations of M. Then, since the time evolution operator of M is linear, statement 1 tells us that if we form the initial configuration $|\phi'_n\rangle$ of M' by replacing each state q_f in the $|\phi_n\rangle$ with state q'_0, then M' takes $|\phi'_n\rangle$ to $\pi(|\phi_n\rangle)$ in a single step. Since M is in normal form, none of $|\phi_2\rangle, \ldots, |\phi_n\rangle$ have support on any superposition in state q_0. Therefore, statement 2 tells us that the next n steps of M' lead to superpositions $\pi(|\phi_{n-1}\rangle), \ldots, \pi(|\phi_1\rangle)$. Finally, statement 3 tells us that a single step of M' maps superposition $\pi(|c_x\rangle)$ to superposition $|c'_x\rangle$.

We define the transition function δ' to give a well-formed M' satisfying these three statements with the following transition rules.

1. $\delta'(q'_0, \sigma) = |\sigma\rangle|q_f\rangle|\bar{d}_{q_f}\rangle$.
2. For each $q \in Q - q_0$ and each $\tau \in \Sigma$,

$$\delta'(q, \tau) = \sum_{p,\sigma} \delta(p, \sigma, \tau, q, d_q)^* |\sigma\rangle|p\rangle|\bar{d}_p\rangle.$$

3. $\delta'(q_0, \sigma) = |\sigma\rangle|q'_f\rangle|d_{q_0}\rangle$.
4. $\delta'(q'_f, \sigma) = |\sigma\rangle|q'_0\rangle|R\rangle$.

The first and third rules can easily be seen to ensure statements 1 and 3. The second rule can be seen to ensure statement 2 as follows: Since M is in normal form, it maps a superposition $|\phi_1\rangle$ to a superposition $|\phi_2\rangle$ with no support on any

configuration in state q_0 if and only if $|\phi_1\rangle$ has no support on any configurations in state q_f. Therefore, the time evolution of M defines a unitary transformation from the space $\mathcal{S}_1$ of superpositions of configurations in states from the set $Q_1 = Q - q_f$ to the space of superpositions $\mathcal{S}_2$ of configurations in states from the set $Q_2 = Q - q_0$. This fact also tells us that the second rule above defines a linear transformation from space $\mathcal{S}_2$ back to space $\mathcal{S}_1$. Moreover, if M takes configuration c_1 with a state from Q_1 with amplitude α to configuration c_2 with a state from Q_2, then M' takes configuration $\pi(c_2)$ to configuration $\pi(c_1)$ with amplitude α^*. Therefore, the time evolution of M' on space $\mathcal{S}_2$ is the composition of π and its inverse around the conjugate transpose of the time evolution of M. Since this conjugate transpose must also be unitary, the second rule above actually gives a unitary transformation from the space $\mathcal{S}_2$ to the space $\mathcal{S}_1$ which satisfies statement 2 above.

Since M' is clearly in normal form, we complete the proof by showing that M' is well formed. To see this, just note that each of the four rules defines a unitary transformation to one of a set of four mutually orthogonal subspaces of the spaces of superpositions of M'. □

The synchronization theorem will allow us to take an existing QTM and put it inside a loop so that the machine can be run any specified number of times.

Building a reversible machine that loops indefinitely is trivial. However, if we want to loop some finite number of times, we need to carefully construct a reversible entrance and exit. As with the branching lemma above, the construction will proceed by building a reversible TM that accomplishes the desired looping and then inserting the QTM for a particular state in this looping machine. However, there are several difficulties. First, in this construction, as opposed to the branching construction, the reversible TM leaves an intermediate computation written on its tape while the QTM runs. This means that inserting a nonstationary QTM would destroy the proper functioning of the reversible TM. Second, even if we insert a stationary QTM, the second (and any subsequent) time the QTM is run, it may be started in superposition on inputs of different lengths and hence may not halt. Therefore, there is no general statement we are able to make about the behavior of the machine once the insertion is carried out. Instead, we describe here the reversible looping TM constructed in Appendix C on page 1470 and argue about specific QTMs resulting from this machine when they are constructed.

LEMMA 4.13 (looping lemma). *There is a stationary, normal form, reversible TM M and a constant c with the following properties. On input any positive integer k written in binary, M runs for time $O(k \log^c k)$ and halts with its tape unchanged. Moreover, M has a special state q^* such that on input k, M visits state q^* exactly k times, each time with its tape head back in the start cell.*

5. Changing the basis of a QTM's computation. In this section we introduce a fundamentally quantum mechanical feature of quantum computation, namely changing the computational basis during the evolution of a QTM. In particular, we will find it useful to change to a new orthonormal basis for the transition function in the middle of the computation—each state in the new state set is a linear combination of states from the original machine.

This will allow us to simulate a general QTM with one that is unidirectional. It will also allow us to prove that any partially defined quantum transition function which preserves length can be extended to give a well-formed QTM.

We start by showing how to change the basis of the states of a QTM. Then we give a set of conditions for a quantum transition function that is necessary and sufficient

for a QTM to be well formed. The last of these conditions, called separability, will allow us to construct a basis of the states of a QTM which will allow us to prove the unidirection and completion lemmas below.

5.1. The change of basis. If we take a well-formed QTM and choose a new orthonormal basis for the space of superpositions of its states, then translating its transition function into this new basis will give another well-formed QTM that evolves just as the first under the change of basis. Note that in this construction the new QTM has the same time evolution operator as the original machine. However, the states of the new machine differ from those of the old. This change of basis will allow us to prove Lemmas 5.5 and 5.7 below.

LEMMA 5.1. *Given a QTM $M = (\Sigma, Q, \delta)$ and a set of vectors B from $\tilde{C}^Q$ which forms an orthonormal basis for $\mathbf{C}^Q$, there is a QTM $M' = (\Sigma, B, \delta')$ which evolves exactly as M under a change of basis from Q to B.*

Proof. Let $M = (\Sigma, Q, \delta)$ be a QTM, and let B be an orthonormal basis for $\mathbf{C}^Q$.

Since B is an orthonormal basis, this establishes a unitary transformation from the space of superpositions of states in Q to the space of superpositions of states in B. Specifically, for each $p \in Q$ we have the mapping

$$|p\rangle \to \sum_{v \in B} \langle p||v\rangle \ |v\rangle.$$

Similarly, we have a unitary transformation from the space of superpositions of configurations with states in Q to the space of configurations with states in B. In this second transformation, a configuration with state p is mapped to the superposition of configurations, where the corresponding configuration with state v appears with amplitude $\langle p||v\rangle$.

Let us see what the time evolution of M should look like under this change of basis. In M a configuration in state p reading a σ evolves in a single time step to the superposition of configurations corresponding to the superposition $\delta(p, \sigma)$:

$$\delta(p, \sigma) = \sum_{\tau, q, d} \delta(p, \sigma, \tau, q, d) \ |\tau\rangle|q\rangle|d\rangle.$$

With the change of basis, the superposition on the right-hand side will instead be

$$\sum_{\tau, v, d} \left(\sum_{q} \langle q|v\rangle \ \delta(p, \sigma, \tau, q, d) \right) |\tau\rangle|v\rangle|d\rangle.$$

Now, since the state symbol pair $|v\rangle|\sigma\rangle$ in M' corresponds to the superposition

$$\sum_{p} \langle v|p\rangle \ |p\rangle|\sigma\rangle$$

in M, we should have in M'

$$\delta'(v, \sigma) = \sum_{p} \langle v|p\rangle \left(\sum_{\tau, v', d} \left(\sum_{q} \langle q|v'\rangle \ \delta(p, \sigma, \tau, q, d) \right) \right) |\tau\rangle|v'\rangle|d\rangle.$$

Therefore, M' will behave exactly as M under the change of basis if we define δ' by saying that for each $v, \sigma \in B \times \Sigma$,

$$\delta'(v, \sigma) = \sum_{\tau, v', d} \left(\sum_{p, q} \langle v|p\rangle \ \langle q|v'\rangle \ \delta(p, \sigma, \tau, q, d) \right) |\tau\rangle|v'\rangle|d\rangle.$$

Since the vectors in B are contained in $\tilde{C}^Q$, each amplitude of δ' is contained in $\tilde{C}$.

Finally, note that the time evolution of M' must preserve Euclidean length since it is exactly the time evolution of the well-formed M under the change of basis. □

5.2. Local conditions for QTM wellformedness. In our discussion of reversible TMs in Appendix B we find properties of a deterministic transition function which are necessary and sufficient for it to be reversible. Similarly, our next theorem gives three properties of a quantum transition function which together are necessary and sufficient for it to be well formed. The first property ensures that the time evolution operator preserves length when applied to any particular configuration. Adding the second ensures that the time evolution preserves the length of superpositions of configurations with their tape head in the same cell. The third property, which concerns "restricted" update superpositions, handles superpositions of configurations with tape heads in differing locations. This third property will be of critical use in the constructions of Lemmas 5.5 and 5.7 below.

DEFINITION 5.2. *We will also refer to the superposition of states*

$$\sum_{q \in Q} \delta(p, \sigma, \tau, q, d)|q\rangle$$

resulting from restricting $\delta(p, \sigma)$ to those pieces writing symbol τ and moving in direction d as a direction d-going restricted superposition*, denoted by $\delta(p, \sigma|\tau, d)$.*

THEOREM 5.3. *A QTM $M = (\Sigma, Q, \delta)$ is well formed iff the following conditions hold:*

unit length

$$\forall\ p, \sigma\ \in\ Q \times \Sigma \quad \|\delta(p, \sigma)\| = 1,$$

orthogonality

$$\forall\ (p_1, \sigma_1) \neq (p_2, \sigma_2)\ \in\ Q \times \Sigma \quad \delta(p_1, \sigma_1) \cdot \delta(p_2, \sigma_2) = 0,$$

separability

$$\forall\ (p_1, \sigma_1, \tau_1), (p_2, \sigma_2, \tau_2)\ \in\ Q \times \Sigma \times \Sigma \quad \delta(p_1, \sigma_1|\tau_1, L) \cdot \delta(p_2, \sigma_2|\tau_2, R) = 0.$$

Proof. Let U be the time evolution of a proposed QTM $M = (\Sigma, Q, \delta)$. We know M is well formed iff U^* exists and U^*U gives the identity or, equivalently, iff the columns of U have unit length and are mutually orthogonal. Clearly, the first condition specifies exactly that each column has unit length. In general, configurations whose tapes differ in a cell not under either of their heads, or whose tape heads are not either in the same cell or exactly two cells apart, cannot yield the same configuration in a single step. Therefore, such pairs of columns are guaranteed to be orthogonal, and we need only consider pairs of configurations for which this is not the case. The second condition specifies the orthogonality of pairs of columns for configurations that differ only in that one is in state p_1 reading σ_1 while the other is in state p_2 reading σ_2.

Finally, we must consider pairs of configurations with their tape heads two cells apart. Such pairs can only interfere in a single step if they differ at most in their states and in the symbol written in these cells. The third condition specifies the orthogonality of pairs of columns for configurations which are identical except that the second has its tape head two cells to the left, is in state p_2 instead of p_1, has a σ_2

instead of a τ_2 in the cell under its tape head, and has a τ_1 instead of a σ_1 two cells to the left. □

Now consider again unidirectional QTMs, those in which each state can be entered while moving in only one direction. When we considered this property for deterministic TMs, it meant that when looking at a deterministic transition function δ, we could ignore the direction update and think of δ as giving a bijection from the current state and tape symbol to the new symbol and state. Here, if δ is a unidirectional quantum transition function, then it certainly satisfies the separability condition since no left-going and right-going restricted superpositions have a state in common. Moreover, update triples will always share the same direction if they share the same state. Therefore, a unidirectional δ is well formed iff, ignoring the direction update, δ gives a unitary transformation from superpositions of current state and tape symbol to superpositions of new symbol and state.

5.3. Unidirection and completion lemmas. The separability condition of Theorem 5.3 allows us to simulate any QTM with a unidirectional QTM by applying a change of basis. The same change of basis will also allow us to complete any partially specified quantum transition function which preserves length.

It is straightforward to simulate a deterministic TM with one which is unidirectional. Simply split each state q into two states q_r and q_l, both of which are given the same transitions that q had, and then edit the transition function so that transitions moving right into q enter q_r and transitions moving left into q enter q_l. The resulting machine is clearly not reversible since the transition function operates the same on each pair of states q_r, q_l.

To simplify the unidirection construction, we first show how to interleave a series of quantum transition functions.

LEMMA 5.4. *Given k state sets $Q_0, \ldots, Q_{k-1}, Q_k = Q_0$ and k transition functions each mapping from one state set to the next*

$$\delta_i : Q_i \times \Sigma \to \tilde{\mathbf{C}}^{\Sigma \times Q_{i+1} \times \{L,R\}}$$

such that each δ_i preserves length, there is a well-formed QTM M with state set $\bigcup_i (Q_i, i)$ which alternates stepping according to each of the k transition functions.

Proof. Suppose we have k state sets transition functions as stated above. Then we let M be the QTM with the same alphabet, with state set given by the union of the individual state sets $\bigcup_i (Q_i, i)$, and with transition function according to the δ_i,

$$\delta((p,i),\sigma) = \sum_{\tau,q,d} \delta_i(p,\sigma,\tau,q,d)|\tau\rangle|q, i+1\rangle|d\rangle.$$

Clearly, the machine M alternates stepping according to $\delta_0, \ldots, \delta_{k-1}$. It is also easy to see that the time evolution of M preserves length. If $|\phi\rangle$ is a superposition of configurations with squared length α_i in the subspace with configurations with states from Q_i, then δ maps ϕ to a superposition with squared length α_i in the subspace with configurations with states from Q_{i+1}. □

LEMMA 5.5 (unidirection lemma). *Any QTM M is simulated, with slowdown by a factor of 5, by a unidirectional QTM M'. Furthermore, if M is well behaved and in normal form, then so is M'.*

Proof. The key idea is that the separability condition of a well-formed QTM allows a change of basis to a state set in which each state can be entered from only one direction.

The separability condition says that

$$\forall\ (p_1,\sigma_1,\tau_1),(p_2,\sigma_2,\tau_2)\ \in\ Q\times\Sigma\times\Sigma,$$

$$\delta(p_1,\sigma_1|\tau_1,L)\cdot\delta(p_2,\sigma_2|\tau_2,R)=0.$$

This means that we can split $\mathbf{C}^Q$ into mutually orthogonal subspaces $\mathbf{C}_L$ and $\mathbf{C}_R$ such that $span(\mathbf{C}_L,\mathbf{C}_R)=\mathbf{C}^Q$ and

$$\forall\ (p_1,\sigma_1,\tau_1)\in\ Q\times\Sigma\times\Sigma,$$

$$\delta(p_1,\sigma_1|\tau_1,d)\in\mathbf{C}_d.$$

Now, as shown in Lemma 5.1 above, under a change of basis from state set Q to state set B the new transition function is defined by

$$\delta'(v,\sigma,\tau,v',d)=\sum_{p,q}\langle v|p\rangle\langle q|v'\rangle\delta(p,\sigma,\tau,q,d).$$

So, choose orthonormal bases B_L and B_R for the spaces $\mathbf{C}_L$ and $\mathbf{C}_R$ and let $M'=(\Sigma,B_L\cup B_R,\delta')$ be the QTM constructed according to Lemma 5.1 which evolves exactly as M under a change of basis from state set Q to state set $B=B_L\cup B_R$. Then any state in M' can be entered in only one direction. To see this, first note that since $\delta(p,\sigma|\tau,d)\in B_d$ and $v=\sum_q\langle v|q\rangle|q\rangle$, the separability condition implies that for $v\in B_{\overline{d}}$,

$$\sum_q\delta(p,\sigma,\tau,q,d)\langle v|q\rangle^*=0.$$

Therefore, for any $v,\sigma,\tau,v'\in B\times\Sigma\times\Sigma\times B_{\overline{d}}$,

$$\delta'(v,\sigma,\tau,v',d)=\sum_{p,q}\langle v|p\rangle\langle q|v'\rangle\delta(p,\sigma,\tau,q,d)$$

$$=\sum_p\langle v|p\rangle\sum_q\langle q|v'\rangle\delta(p,\sigma,\tau,q,d)=0.$$

Therefore, any state in B can be entered while traveling in only one direction.

Unfortunately, this new QTM M' might not be able to simulate M. The problem is that the start state and final state of M might correspond under the change of basis isomorphism to superpositions of states in M', meaning that we would be unable to define the necessary start and final states for M'. To fix this problem, we use five time steps to simulate each step of M and interleave the five transition functions using Lemma 5.4 on page 1435.

1. Step right leaving the tape and state unchanged:

$$\delta_0(p,\sigma)=|\sigma\rangle|p\rangle|R\rangle.$$

2. Change basis from Q to B while stepping left:

$$\delta_1(p,\sigma)=\sum_{b\in B}\langle p|b\rangle|\sigma\rangle|b\rangle|L\rangle.$$

3. M' carries out a step of the computation of M. So, δ_2 is just the quantum transition function δ' from QTM M' constructed above.

4. Change basis back from B to Q while stepping left:

$$\delta_3(b,\sigma) = \sum_{p\in Q} \langle b|p\rangle|\sigma\rangle|p\rangle|L\rangle.$$

5. Step right leaving the tape and state unchanged:

$$\delta_4(p,\sigma) = |\sigma\rangle|p\rangle|R\rangle.$$

If we construct QTM M' with state set $(Q\times\{0,1,4\}\cup(B\times\{2,3\})$ using Lemma 5.4 on page 1435 and let M' have start and final states $(q_0,0)$ and $(q_f,0)$, then M' simulates M with slowdown by a factor of 5.

Next, we must show that each of the five transition functions obeys the well-formedness conditions and hence according to Lemma 5.4 that the interleaved machine is well formed.

The transition function $\delta_2 = \delta'$ certainly obeys the wellformedness conditions since M' is a well formed QTM. Also, δ_0 and δ_4 obey the three wellformedness conditions since they are deterministic and reversible. Finally, the transition functions δ_1 and δ_3 satisfy the unit length and orthogonality conditions since they implement a change of basis, and they obey the separability condition since they only move in one direction.

Finally, we must show that if M is well-behaved and in normal form, then we can make M' well behaved and in normal form.

So, suppose M is well behaved and in normal form. Then there is a T such that at time T the superposition includes only configurations in state q_f with the tape head back in the start cell, and at any time less than T the superposition contains no configuration in state q_f. But this means that when M' is run on input x, the superposition at time $5T$ includes only configurations in state $(q_f,0)$ with the tape head back in the start cell, and the superposition at any time less than $5T$ contains no configuration in state $(q_f,0)$. Therefore, M' is also well behaved.

Then for any input x there is a T such that M enters a series of $T-1$ superpositions of configurations all with states in $Q-\{q_0,q_f\}$ and then enters a superposition of configurations all in state q_f with the tape head back in the start cell. Therefore, on input x, M' enters a series of $5T-1$ superpositions of configurations all with states in $Q'-\{(q_f,0),(q_0,4)\}$ and then enters a superposition of configurations all in state $(q_f,0)$ with the tape head back in the start cell. Therefore, M' is well behaved. Also, swapping the outgoing transitions of $(q_f,0)$ and $(q_0,4)$, which puts M' in normal form, will not change the computation of M' on any input x. □

When we construct a QTM, we will often only be concerned with a subset of its transitions. Luckily, any partially defined transition function that preserves length can be extended to give a well-formed QTM.

DEFINITION 5.6. *A QTM M whose quantum transition function δ' is only defined for a subset $S\subseteq Q\times\Sigma$ is called a* partial *QTM. If the defined entries of δ' satisfy the three conditions of Theorem* 5.3 *on page* 1434 *then M is called a* well-formed partial *QTM.*

LEMMA 5.7 (completion lemma). *Suppose M is a well-formed partial QTM with quantum transition function δ. Then there is a well-formed QTM M' with the same state set and alphabet whose quantum transition function δ' agrees with δ wherever the latter is defined.*

Proof. We noted above that a unidirectional quantum transition function is well formed iff, ignoring the direction update, it gives a unitary transformation from superpositions of current state and tape symbol to superpositions of new symbol and state. So if our partial QTM is unidirectional, then we can easily fill in the undefined entries of δ by extending the set of update superpositions of δ to an orthonormal basis for the space of superpositions of new symbol and state.

For a general δ, we can use the technique of Lemma 5.1 on page 1433 to change the basis of δ away from Q so that each state can be entered while moving in only one direction, extend the transition function, and then rotate back to the basis Q.

We can formalize this as follows: let $M = (\Sigma, Q, \delta)$ be a well-formed partial QTM with δ defined on the subset $S \subseteq Q \times \Sigma$. Denote by $\overline{S}$ the complement of S in $Q \times \Sigma$.

As in the proof of the unidirection lemma above, the separability condition allows us to partition $\mathbf{C}^Q$ into mutually orthogonal subspaces $\mathbf{C}_L$ and $\mathbf{C}_R$ such that $span(\mathbf{C}_L, \mathbf{C}_R) = \mathbf{C}^Q$ and

$$\forall\ (p_1, \sigma_1, \tau_1) \in\ S \times \Sigma,$$

$$\delta(p_1, \sigma_1 | \tau_1, d) \in \mathbf{C}_d.$$

Then, we choose orthonormal bases B_L and B_R for $\mathbf{C}_L$ and $\mathbf{C}_R$ and consider the unitary transformation from superpositions of configurations with states in Q to the space of configurations with states in $B = B_L \cup B_R$, where any configuration with state p is mapped to the superposition of configurations where the corresponding configuration with state v appears with amplitude $\langle p|v\rangle$.

If we call δ' the partial function δ followed by this unitary change of basis, then we have

$$\delta'(p, \sigma) = \sum_{\tau, v, d} \left(\sum_q \langle q|v\rangle \delta(p, \sigma, \tau, q, d) \right) |\tau\rangle |v\rangle |d\rangle.$$

Since δ preserves length and δ' is δ followed by a unitary transformation, δ' also preserves length.

But now δ' can enter any state in B while moving in only one direction. To see this, first note that since $\delta(p, \sigma; \tau, d) \in B_d$ and $v = \sum_q \langle v|q\rangle |q\rangle$, the separability condition implies that for $v \in B_{\overline{d}}$,

$$\sum_q \delta(p, \sigma, \tau, q, d) \langle v|q\rangle^* = 0.$$

Therefore, for any $p, \sigma, \tau, v \in S \times \Sigma \times B_{\overline{d}}$,

$$\delta'(p, \sigma, \tau, v, d) = \sum_q \langle q|v\rangle \delta(p, \sigma, \tau, q,) = 0.$$

Therefore, any state in B can be entered while traveling in only one direction.

Then, since the direction is implied by the new state, we can think of δ' as mapping the current state and symbol to a superposition of new symbol and state. Since δ' preserves length, the set $\delta'(S)$ is a set of orthonormal vectors, and we can expand this set to an orthonormal basis of the space of superpositions of new symbol and state. Adding the appropriate direction updates and assigning these new vectors arbitrarily to $\delta'(\bar{S})$, we have a completely defined δ' that preserves length. Therefore, assigning $\delta(\bar{S})$ as $\delta'(\bar{S})$ followed by the inverse of the basis transformation gives a completion for δ that preserves length. ∎

6. An efficient QTM implementing any given unitary transformation. Suppose that the tape head of a QTM is confined to a region consisting of k contiguous tape cells (the tape is blank outside this region). Then the time evolution of the QTM can be described by a d-dimensional unitary transformation, where $d = k\text{card}(Q)\text{card}(\Sigma)^k$. In this section we show conversely that there is a QTM that, given any d-dimensional unitary transformation as input, carries out that transformation (on a region of its tape). To make this claim precise we must say how the d-dimensional unitary transformation is specified. We assume that an approximation to the unitary transformation is specified by a $d \times d$ complex matrix whose entries are approximations to the entries of the actual unitary matrix corresponding to the desired transformation. We show in Theorem 6.11 on page 1447 that there is a QTM that, on input ϵ and a d-dimensional transformation which is within distance $\frac{\epsilon}{2(10\sqrt{d})^d}$ of a unitary transformation, carries out a transformation which is an ϵ approximation to the desired unitary transformation. Moreover, the running time of the QTM is bounded by a polynomial in d and $\frac{1}{\epsilon}$.

In a single step, a QTM can map a single configuration into a superposition of a bounded number of configurations. Therefore, in order to carry out an (approximation to an) arbitrary unitary transformation on a QTM, we show how to approximate it by a product of simple unitary transformations—each such simple transformation acts as the identity in all but two dimensions. We then show that there is a particular simple unitary transformation such that any given simple transformation can be expressed as a product of permutation matrices and powers of this fixed simple matrix. Finally, we put it all together and show how to design a single QTM that carries out an arbitrary unitary transformation—this QTM is deterministic except for a single kind of quantum coin flip.

The decomposition of an arbitrary unitary transformation into a product of simple unitary transformations is similar to work carried out by Deutsch [21]. Deutsch's work, although phrased in terms of quantum computation networks, can be viewed as showing that a d-dimensional unitary transformation can be decomposed into a product of transformations where each applies a particular unitary transformation to three dimensions and acts as the identity elsewhere. We must consider here several issues of efficiency not addressed by Deutsch. First, we are concerned that the decomposition contains a number of transformations which is polynomial in the dimension of the unitary transformation and in the desired accuracy. Second, we desire that the decomposition can itself be efficiently computed given the desired unitary transformation as input. For more recent work on the efficient simulation of a unitary transformation by a quantum computation network see [5] and the references therein.

6.1. Measuring errors in approximated transformations. In this section, we will deal with operators (linear transformations) on finite-dimensional Hilbert spaces. It is often convenient to fix an orthonormal basis for the Hilbert space and describe the operator by a finite matrix with respect to the chosen basis. Let $e_1, \ldots, e_d$ be an orthonormal basis for Hilbert space $\mathcal{H} = \mathbf{C}^d$. Then we can represent an operator U on $\mathcal{H}$ by a $d \times d$ complex matrix M, whose i,jth entry $m_{i,j}$ is (Ue_j, e_i). The ith row of the matrix M is given by ${e_i}^T M$, and we will denote it by M_i. We denote by ${M_i}^*$ the conjugate transpose of M_i. The jth column of M is given by Me_j. U^*, the adjoint of U, is represented by the $d \times d$ matrix M^*. M is unitary iff $MM^* = M^*M = I$. It follows that if M is unitary then the rows (and columns) of M are orthonormal.

Recall that for a bounded linear operator U on a Hilbert space $\mathcal{H}$, the norm of U is defined as

$$|U| \;=\; \sup_{\|x\|=1} |Ux|.$$

If we represent U by the matrix M, then we can define the norm of the matrix M to be same as the norm of U. Thus, since we're working in a finite-dimensional space,

$$\|M\| \;=\; \max_{\|v\|=1} \|Mv\|.$$

FACT 6.1. *If U is unitary, then $\|U\| = \|U^*\| = 1$.*

Proof. $\forall x \in \mathcal{H}$, $\|Ux\|^2 = (Ux, Ux) = (x, U^*Ux) = (x, x) = \|x\|^2$. Therefore, $\|U\| = 1$, and similar reasoning shows $\|U^*\| = 1$. □

We will find it useful to keep track of how far our approximations are from being unitary. We will use the following simple measure of a transformation's distance from being unitary.

DEFINITION 6.2. *A bounded linear operator U is called* ϵ-close to unitary *if there is a unitary operator $\tilde{U}$ such that $\|U - \tilde{U}\| \leq \epsilon$. If we represent U by the matrix M, then we can equivalently say that M is* ϵ-close to unitary *if there is a unitary matrix $\tilde{M}$ such that $\|M - \tilde{M}\| \leq \epsilon$.*

Notice that, appealing to statement (2.3) in section 2 if U is ϵ-close to unitary, then $1 - \epsilon \leq \|U\| \leq 1 + \epsilon$. However, the converse is not true. For example the linear transformation $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ has norm 1, but is 1 away from unitary.

Next, we show that if M is close to unitary, then the rows of M are close to unit length and will be close to orthogonal.

LEMMA 6.3. *If a d-dimensional complex matrix M is ϵ-close to unitary, then*

$$1 - \epsilon \leq \|M_i\| \;\leq\; 1 + \epsilon, \tag{6.1}$$

$$\forall i \neq j, \quad \|M_i M_j{}^*\| \leq 2\epsilon + 3\epsilon^2. \tag{6.2}$$

Proof. Let $\tilde{M}$ be the unitary matrix such that $\|M - \tilde{M}\| \leq \epsilon$. Let $N = M - \tilde{M}$. Then we know that for each i, $\|\tilde{M}_i\| = 1$ and $\|N_i\| \leq \epsilon$.

So, for any i, we have $M_i = N_i + \tilde{M}_i$. Therefore, $1 - \epsilon \leq \|M_i\| \leq 1 + \epsilon$.

Next, since $\tilde{M}$ is unitary, it follows that for $i \neq j$, $\tilde{M}_i \tilde{M}_j^* = 0$. Therefore, $(M_i - N_i)(M_j^* - N_j^*) = 0$. Expanding this as a sum of four terms, we get

$$M_i M_j^* = M_i N_j^* + N_i M_j^* - N_i N_j^*.$$

Since $\|M\| \leq 1 + \epsilon$ and $\|N\| \leq \epsilon$, the Schwarz inequality tells us that $\|M_i N_j^*\| \leq (1 + \epsilon)\epsilon$, $\|N_i M_j^*\| \leq \epsilon(1 + \epsilon)$, and $\|N_i N_j^*\| \leq \epsilon^2$. Using the triangle inequality we conclude that $\|M_i M_j^*\| \leq 2(1 + \epsilon)\epsilon + \epsilon^2$. Therefore, $\|M_i M_j{}^*\| \leq 2\epsilon + 3\epsilon^2$. □

We will also use the following standard fact that a matrix with small entries must have small norm.

LEMMA 6.4. *If M is a d-dimensional square complex matrix such that $|m_{i,j}| \leq \epsilon$ for all i, j, then $\|M\| \leq d\epsilon$.*

Proof. If each entry of M has magnitude at most ϵ, then clearly each row M_i of M must have norm at most $\sqrt{d}\epsilon$. So, if v is a d-dimensional column vector with $|v| = 1$, we must have

$$\|Mv\|^2 = \sum_i \|M_i v\|^2 \leq \sum_i d\epsilon^2 = d^2\epsilon^2,$$

where the inequality follows from the Schwarz inequality. Therefore, $\|M\| \leq d\epsilon$. □

6.2. Decomposing a unitary transformation. We now describe a class of exceedingly simple unitary transformations which we will be able to carry out using a single QTM. These "near-trivial" transformations either apply a phase shift in one dimension or apply a rotation between two dimensions, while acting as the identity otherwise.

DEFINITION 6.5. *A $d \times d$ unitary matrix M is* near trivial *if it satisfies one of the following two conditions.*

1. *M is the identity except that one of its diagonal entries is $e^{i\theta}$ for some $\theta \in [0, 2\pi]$. For example, $\exists j\ m_{j,j} = e^{i\theta}\ \forall k \neq j\ m_{k,k} = 1$, and $\forall k \neq l\ m_{k,l} = 0$.*

2. *M is the identity except that the submatrix in one pair of distinct dimensions j and k is the rotation by some angle $\theta \in [0, 2\pi] : \left(\begin{smallmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{smallmatrix}\right)$. So, as a transformation M is near trivial if there exists θ and $i \neq j$ such that $Me_i = (\cos\theta)e_i + (\sin\theta)e_j$, $Me_j = -(\sin\theta)e_i + (\cos\theta)e_j$, and $\forall k \neq i, j\ Me_k = e_k$.*

We call a transformation which satisfies statement 1 a near-trivial phase shift*, and we call a transformation which satisfies Statement 2 a* near-trivial rotation.

We will write a near-trivial matrix M in the following way. If M is a phase shift of $e^{i\theta}$ in dimension j, then we will write down $[j, j, \theta]$ and if M is a rotation of angle θ between dimensions j and k we will write down $[j, k, \theta]$. This convention guarantees that the matrix that we are specifying is a near-trivial matrix and therefore a unitary matrix, *even if* for precision reasons we write down an approximation to the matrix that we really wish to specify. This feature will substantially simplify our error analyses.

Before we show how to use near-trivial transformations to carry out an arbitrary unitary transformation, we first show how to use them to map any particular vector to a desired target direction.

LEMMA 6.6. *There is a deterministic algorithm which on input a vector $v \in \mathbf{C}^d$ and a bound $\epsilon > 0$ computes near-trivial matrices $U_1, \ldots, U_{2d-1}$ such that*

$$\|U_1 \cdots U_{2d-1} v \ - \ \|v\| e_1\| \leq \epsilon,$$

where e_1 is the unit vector in the first coordinate direction. The running time of the algorithm is bounded by a polynomial in d, $\log \frac{1}{\epsilon}$ and the length of the input.

Proof. First, we use d phase shifts to map v into the space $\mathbb{R}^d$. We therefore want to apply the phase shift $\frac{v_i^*}{\|v_i\|}$ to each dimension i with $v_i \neq 0$. So, we let P_i be the near-trivial matrix which applies to dimension i the phase shift by angle ϕ_i, where $\phi_i = 0$ if $v_i = 0$ and otherwise $\phi_i = 2\pi - \cos^{-1} \frac{\mathrm{Re}(v_i)}{\|v_i\|}$ or $\cos^{-1} \frac{\mathrm{Re}(v_i)}{\|v_i\|}$ depending whether $\mathrm{Im}(v_i)$ is positive or negative. Then $P_1 \cdots P_d v$ is the vector with ith coordinate $\|v_i\|$.

Next, we use $d-1$ rotations to move all of the weight of the vector into dimension 1. So, we let R_i be the near-trivial matrix which applies the rotation by angle θ_i to dimensions i and $i+1$, where

$$\theta_i = \cos^{-1} \frac{\|v_i\|}{\sqrt{\sum_{j=i}^{d} \|v_j\|^2}}$$

if the sum in the denominator is not 0 and $\theta_i = 0$ otherwise. Then

$$R_1 \cdots R_{d-1} P_1 \cdots P_d v = \|v\| e_1.$$

Now, instead of producing these values ϕ_i, θ_i exactly, we can compute, in time polynomial in d and $\log \frac{1}{\epsilon}$ and the length of the input, values ϕ_i' and θ_i' which are

within $\delta = \frac{\epsilon}{(2d-1)\|v\|}$ of the desired values. Call P_i' and R_i' the near-trivial matrices corresponding to P_i and R_i but using these approximations. Then, since the distance between points at angle θ and θ' on the unit circle in the real plane is at most $|\theta - \theta'|$,

$$\|R_i - R_i'\| \leq \delta.$$

Thinking of the same inequality on the unit circle in the complex plane, we have

$$\|P_i - P_i'\| \leq \delta.$$

Finally, since each matrix P_i, P_i', R_i, R_i' is unitary, Fact 2.1 on page 1417 gives us

$$\|R_1' \cdots R_{d-1}' P_1' \cdots P_d' \ - \ R_1 \cdots R_{d-1} P_1 \cdots P_d\| \leq (2d-1)\delta$$

and therefore

$$\|R_1' \cdots R_{d-1}' P_1' \cdots P_d' v \ - \ \|v\| e_1\| \leq (2d-1)\delta\|v\| = \epsilon. \qquad \square$$

We now show how the ability to map a particular vector to a desired target direction allows us to approximate an arbitrary unitary transformation.

THEOREM 6.7. *There is a deterministic algorithm running in time polynomial in d and $\log 1/\epsilon$ and the length of the input which when given as input U, ϵ where $\epsilon > 0$ and U is a $d \times d$ complex matrix which is $\frac{\epsilon}{2(10\sqrt{d})^d}$-close to unitary, computes d-dimensional near-trivial matrices $U_1, \ldots, U_n$, with n polynomial in d such that $\|U \ - \ U_n \cdots U_1\| \leq \epsilon$.*

Proof. First we introduce notation to simplify the proof. Let U be a $d \times d$ complex matrix. Then we say U is *k-simple* if its first k rows and columns are the same as those of the d-dimensional identity. Notice that the product of two $d \times d$ k-simple matrices is also k-simple.

If U were d-simple, we would have $U = I$ and the desired computation would be trivial. In general, the U which our algorithm must approximate will not even be 1-simple. So, our algorithm will proceed through d phases such that during the ith phase the remaining problem is reduced to approximating a matrix which is $i+1$-simple.

Suppose we start to approximate a k-simple U with a series of near-trivial matrices with the product V. Then to approximate U we would still need to produce a series of near-trivial matrices whose product W satisfies $W \approx UV^*$. To reduce the problem we must therefore compute near-trivial matrices whose product V is such that UV^* is close to being $k+1$-simple. We can accomplish this by using the algorithm of Lemma 6.6 above.

So, let U be given which is k-simple and is δ-close to unitary, and let Z be the lower right $d-k \times d-k$ submatrix of Z. We invoke the procedure of Lemma 6.6 on inputs Z_1^T (the vector corresponding to the first row of Z) and δ. The output is a sequence of $d-k$-dimensional near trivial matrices $V_1, \ldots, V_{2d-2k-1}$ such that their product $V = V_1 \times \cdots \times V_{2d-2k-1}$ has the property that $\|VZ_1^T - \|Z_1\| e_1\| \leq \delta$.

Now suppose that we extend V and the V_i back to d-dimensional, k-simple matrices, and we let $W = UV^*$. Then clearly V is unitary and V and W are k-simple. In fact, since V is unitary and U is δ-close to unitary, W is also δ-close to unitary. Moreover, W is close to being $k+1$-simple as desired. We will show below that the $k+1$st row of W satisfies $\|W_{k+1} - e_{k+1}^T\| \leq 2\delta$ and that the entries of the $k+1$st column of W satisfy $\|w_{j,k+1}\| \leq 6\delta$ for $j \neq k+1$.

So, let X be the $d \times d$, $k+1$-simple matrix such that

$$x_{1,1} = 1,\ x_{j,1} = 0 \text{ for } j \neq 1,\ x_{1,j} = 0 \text{ for } j \neq 1,\ x_{j,l} = w_{j,l} \text{ for } j, l > l+1.$$

It follows from our bounds on the norm of the first row of W and on the entries of the first column of W that $\|W - X\| \leq 2\delta + 6\sqrt{d}\delta$. Since W is δ-close to unitary, we can then conclude that X is $3\delta + 6\sqrt{d}\delta$-close to unitary.

Unfortunately, we cannot compute the entries of $W = UV^*$ exactly. Instead, appealing to Lemma 6.4 on page 1440, we compute them to within $\frac{\delta}{d}$ to obtain a matrix $\hat{W}$ such that $\|\hat{W} - W\| \leq \delta$. Let's use the entries of $\hat{W}$ to define matrix $\hat{X}$ analogous to X. Using the triangle inequality, it is easy to see that $\|W - \hat{X}\| \leq 3\delta + 6\sqrt{d}\delta$ and $\hat{X}$ is $4\delta + 6\sqrt{d}\delta$-close to unitary. If we are willing to incur an error of $\|W - \hat{X}\| \leq 3\delta + 6\sqrt{d}\delta$, then we are left with the problem of approximating the $k+1$-simple $\hat{X}$ by a product of near-trivial matrices. Therefore, we have reduced the problem of approximating the k-simple matrix U by near-trivial matrices to the problem of approximating the $k+1$-simple matrix $\hat{X}$ by near-trivial matrices while incurring two sources of error:

1. an error of $\|W - \hat{X}\| \leq 3\delta + 6\sqrt{d}\delta$, since we are approximating $\hat{X}$ instead of W;
2. the new matrix $\hat{X}$ is only $4\delta + 6\sqrt{d}\delta$-close to unitary.

Let $\delta' = 10\sqrt{d}\delta$. Clearly δ' is an upper bound on both sources of error cited above. Therefore, the total error in the approximation is just $\sum_{j=1}^{d} (10\sqrt{d})^j \delta \leq 2(10\sqrt{d})^d \delta$. The last inequality follows since $10\sqrt{d} \geq 2$, and therefore the sum can be bounded by a geometric series. Therefore, the total error in the approximation is bounded by ϵ, since by assumption U is δ-close to unitary for $\delta = \frac{\epsilon}{2(10\sqrt{d})^d}$.

It is easy to see that this algorithm runs in time polynomial in d and $\log \frac{1}{\epsilon}$. Our algorithm consists of d iterations of first calling the algorithm from Lemma 6.6 on page 1441 to compute V and then computing the matrix $\hat{X}$. Since the each iteration takes time polynomial in d and $\log \frac{(10\sqrt{d})^d}{\epsilon}$, these d calls take a total time polynomial in d and $\log \frac{1}{\epsilon}$.

Finally, we show as required that the $k+1$st row of W satisfies $\|W_{k+1} - e_{k+1}^T\| \leq 2\delta$ and that the entries of the $k+1$st column of W satisfy $\|w_{j,k+1}\| \leq 6\delta$ for $j \neq k+1$. To see this, first recall that the lower dimension V satisfies $\|VZ_1^T - \|Z_1\|e_1\| \leq \delta$, where Z_1 is the first row of the lower right $k \times k$ submatrix of U. Therefore, the higher dimension V satisfies $\|VU_{k+1}^T - \|U_{k+1}\|e_{k+1}\| \leq \delta$. Then, since $1-\delta \leq \|U_{k+1}\| \leq 1+\delta$, it follows that $\|VU_{k+1}^T - e_{k+1}\| \leq 2\delta$. Therefore, the $k+1$st row of W satisfies $\|W_{k+1} - e_{k+1}^T\| \leq 2\delta$.

Next, we will show that this implies that the entries of the $k+1$st column of W satisfy $\|w_{j,k+1}\| \leq 6\delta$ for $j \neq k+1$. To see this, first notice that since V is unitary and U is delta close to unitary, W is also δ-close to unitary. This means that by statement (6.2) of Lemma 6.3 on page 1440, $\left|W_{k+1} W_j^*\right| \leq 2\delta + 3\delta^2$. Now let us use the condition $\|W_{k+1} - e_{k+1}^T\| \leq 2\delta$. This implies that $|w_{k+1,k+1}| \geq 1 - 2\delta$. Also, let us denote by $\hat{W}_j$ the $d-1$-dimensional row vector arrived at by dropping $w_{j,k+1}$ from W_j. Then the condition that W_{k+1} is close to e_{k+1}^T also implies that $\|\hat{W_{k+1}}\| \leq 2\delta$. Also, the fact that W is δ-close to unitary implies that $\|\hat{W}_j\| \leq 1 + \delta$. Putting all this together, we have $2\delta + 3\delta^2 \geq |W_{k+1} W_j^*| = |w_{k+1,k+1} w_{j,k+1}^* + \hat{W_{k+1}} \hat{W_j}^*| \geq |w_{k+1,k+1} w_{j,k+1}^*| - |\hat{W_{k+1}} \hat{W_j}^*|$. Therefore, $|w_{k+1,k+1} w_{j,k+1}^*| \leq 2\delta + 3\delta^2 + |\hat{W_{k+1}} \hat{W_j}^*| \leq 2\delta + 3\delta^2 + 2\delta(1+\delta) \leq 4\delta + 5\delta^2$. Therefore, $|w_{j,k+1}| \leq \frac{4\delta + 5\delta^2}{1-2\delta}$. Finally, since we may assume that $\delta \leq \frac{1}{10}$, we have $|w_{j,k+1}| \leq 6\delta$. □

6.3. Carrying out near-trivial transformations. In this section, we show how to construct a single QTM that can carry out, at least approximately, any specified near-trivial transformation. Since a near-trivial transformation can apply an arbitrary rotation, either between two dimensions or in the phase of a single dimension, we must first show how a fixed rotation can be used to efficiently approximate an arbitrary rotation. Note that a single copy of this fixed rotation gives the only "nonclassical" amplitudes (those other than $0, 1$) in the transition function of the universal QTM constructed below. See Adleman, DeMarrais, and Huang [1] and Solovay and Yao [40] for the constructions of universal QTMs whose amplitudes are restricted to a small set of rationals.

LEMMA 6.8. *Let* $\mathcal{R} = 2\pi \sum_{i=1}^{\infty} 2^{-2^i}$. *Then there is a deterministic algorithm taking time polynomial in* $\log \frac{1}{\epsilon}$ *and the length of the input which on input* θ, ϵ *with* $\theta \in [0, 2\pi]$ *and* $\epsilon > 0$ *produces integer output* k *bounded by a polynomial in* $\frac{1}{\epsilon}$ *such that*

$$|k\mathcal{R} - \theta| \bmod 2\pi \ \leq \ \epsilon.$$

Proof. First, we describe a procedure for computing such a k.

Start by calculating n, a power of 2, such that $\epsilon > \frac{2\pi}{2^{n-1}}$. Next, approximate $\frac{\theta}{2\pi}$ as a fraction with denominator 2^n. In other words, find an integer $m \in [1, 2^n]$ such that

$$\left| \frac{\theta}{2\pi} - \frac{m}{2^n} \right| \leq \frac{1}{2^n}.$$

Then we can let $k = m2^n$ because

$$\begin{aligned} m2^n\mathcal{R} \bmod 2\pi &= \left(2\pi m \sum_{i=1}^{\infty} 2^{n-2^i} \right) \bmod 2\pi \\ &= \left(2\pi m \sum_{i=\log n+1}^{\infty} 2^{n-2^i} \right) \bmod 2\pi \\ &= \left(\frac{2\pi m}{2^n} + 2\pi m \sum_{i=\log n+2}^{\infty} 2^{n-2^i} \right) \bmod 2\pi \end{aligned}$$

and since

$$\begin{aligned} m \sum_{i=\log n+2}^{\infty} 2^{n-2^i} &\leq m2^{n-4n+1} \\ &\leq 2^{n-3n+1} \\ &\leq 2^{-2n+1} \end{aligned}$$

we have

$$\begin{aligned} |m2^n\mathcal{R} - \theta| \bmod 2\pi &\leq \left| m2^n\mathcal{R} - \frac{2\pi m}{2^n} \right| \bmod 2\pi \ + \ \left| \frac{2\pi m}{2^n} - \theta \right| \\ &\leq \frac{2\pi}{2^{2n-1}} + \frac{2\pi}{2^n} \\ &< \frac{2\pi}{2^{n-1}} \\ &< \epsilon. \qquad \square \end{aligned}$$

At this point it should be clear that a single QTM can carry out any sequence of near-trivial transformations to any desired accuracy ϵ. We formalize this notion below by showing that there is a QTM that accepts as input the descriptions of a sequence of near-trivial transformations and an error bound ϵ and applies an ϵ approximation of the product of these transformations on any given superposition. The formalization is quite tedious, and the reader is encouraged to skip the rest of the subsection if the above statement is convincing.

Below, we give a formal definition of what it means for a QTM to carry out a transformation.

DEFINITION 6.9. *Let $\Sigma \cup \#$ be the alphabet of the first track of QTM M. Let $\mathcal{V}$ be the complex vector space of superpositions of k length strings over Σ. Let U be a linear transformation on $\mathcal{V}$, and let x_U be a string that encodes U (perhaps approximately). We say that x_U* causes *M to carry out the k cell transformation U with accuracy ϵ in time T if for every $|\phi\rangle \in \mathcal{V}$, on input $|\phi\rangle|x_U\rangle|\epsilon\rangle$, M halts in exactly T steps with its tape head back in the start cell and with final superposition $(U'|\phi\rangle)|x\rangle$, where U' is a unitary transformation on $\mathcal{V}$ such that $\|U - U'\| \leq \epsilon$. Moreover, for a family A of transformations, we say that M carries out these transformations in polynomial time if T is bounded by a polynomial in $\frac{1}{\epsilon}$ and the length of the input.*

In the case that A contains transformations that are not unitary, we say that M carries out *the set of transformations A with* closeness factor *c if for any $\epsilon > 0$ and any $U \in A$ which is $c\epsilon$-close to unitary, there is a unitary transformation U' with $\|U' - U\| \leq \epsilon$ such that $|x_U\rangle|\epsilon\rangle$ causes M to carry out the transformation U' in time which is polynomial in $\frac{1}{\epsilon}$ and the length of its input.*

Recall that a near-trivial transformation written as x, y, θ calls for a rotation between dimensions x and y of angle θ if $x \neq y$ and a phase shift of $e^{i\theta}$ to dimension x otherwise. So, we want to build a stationary, normal form QTM that takes as input $w; x, y, \theta; \epsilon$ and transforms w according to the near-trivial transformation described by x, y, θ with accuracy ϵ. We also need this QTM's running time to depend only on the length of w but not its value. If this is the case then the machine will also halt on an initial superposition of the form $|\phi\rangle|x, y, \theta\rangle|\epsilon\rangle$ where $|\phi\rangle$ is a superposition of equal-length strings w.

LEMMA 6.10. *There is a stationary, normal form QTM M with first track alphabet $\{\#, 0, 1\}$ that carries out the set of near-trivial transformations on its first track in polynomial time.*

Proof. Using the encoding x, y, θ for near-trivial transformations described above in section 6.2, we will show how to construct QTMs M_1 and M_2 with first track alphabet $\{\#, 0, 1\}$ such that M_1 carries out the set of near-trivial rotations on its first track in polynomial time, and M_2 carries out the set of near-trivial phase shifts on its first track in polynomial time. Using the branching lemma (page 1429) on these two machines, we can construct a QTM that behaves correctly provided there is an extra track containing a 1 when $x = y$ and we have a phase shift to perform, and containing a 0 when $x \neq y$ and we have a rotation to perform. We can then construct the desired QTM M by dovetailing before and after with machines that compute and erase this extra bit based on x, y. The synchronization theorem lets us construct two such stationary, normal form QTMs whose running times depend only on the lengths of x and y. Therefore, this additional computation will not disturb the synchronization of the computation.

Next, we show how to construct the QTM M_1 to carry out near-trivial rotations.

It is easy to construct a QTM which on input b applies a rotation by angle θ between $|0\rangle$ and $|1\rangle$, while leaving b alone if $b = \#$. But Lemma 6.8 above tells us that we can achieve any rotation by applying the single rotation $\mathcal{R}$ at most a polynomial number of times. Therefore, the following five-step process will allow us to apply a near-trivial rotation. We must be careful that when we apply the rotation, the two computational paths with $b \in \{0, 1\}$ differ only in b since otherwise they will not interfere.

1. Calculate k such that $k\,\mathcal{R}\mathrm{mod}\,2\pi \in [\theta - \epsilon, \theta + \epsilon]$.
2. Transform w, x, y into b, x, y, z, where $b = 0$ if $w = x$, $b = 1$ if $w = y$ and $w \neq x$, and $b = \#$ otherwise and where $z = w$ if $b = \#$ and z is the empty string otherwise.
3. Run the rotation applying machine k times on the first bit of z.
4. Reverse step 2 transforming $\#, x, y, w$ with $w \neq x, y$ into w, x, y, transforming $0, x, y$ into x, x, y, and transforming $1, x, y$ with $x \neq y$ into y, x, y.
5. Reverse step 1 erasing k.

We build the desired QTM M by constructing a QTM for each of these five steps and then dovetailing them together.

First, notice that the length of the desired output of steps 1, 2, 4, and 5 can be computed just from the length of the input. Therefore, using Lemma 6.8 from page 1444 and the synchronization theorem from page 1428, we can build polynomial time, stationary, normal form QTMs for steps 1, 2, 4, and 5 which run in time that depend on the lengths of w, x, y but not their particular values.

To complete the construction we must build a machine for the rotation with these same properties. The stationary, normal form QTM R with alphabet $\{\#, 0, 1\}$, state set $\{q_0, q_1, q_f\}$, and transition function defined by

	$\#$	0	1
q_0	$\|\#\rangle\|q_1\rangle\|L\rangle$	$\cos\mathcal{R}\|0\rangle\|q_1\rangle\|L\rangle$ $+\sin\mathcal{R}\|1\rangle\|q_1\rangle\|L\rangle$	$-\sin\mathcal{R}\|0\rangle\|q_1\rangle\|L\rangle$ $+\cos\mathcal{R}\|1\rangle\|q_1\rangle\|L\rangle$
q_1	$\|\#\rangle\|q_f\rangle\|R\rangle$		
q_f	$\|\#\rangle\|q_0\rangle\|R\rangle$	$\|0\rangle\|q_0\rangle\|R\rangle$	$\|1\rangle\|q_0\rangle\|R\rangle$

runs for constant time and applies rotation $\mathcal{R}$ between start cell contents $|0\rangle$ and $|1\rangle$ while leaving other inputs unchanged. Inserting R for the special state in the reversible TM from the looping lemma, we can construct a normal form QTM which applies rotation $k\mathcal{R}$ between inputs $|0, k\rangle$ and $|1, k\rangle$ while leaving input $|\#w, k\rangle$ unchanged. Since the machine we loop on is stationary and takes constant time regardless of its input, the resulting looping machine is stationary and takes time depending only on k.

Finally, with appropriate use of Lemmas 4.4 and 4.5 on page 1428 we can dovetail these five stationary, normal form QTMs to achieve a stationary, normal form QTM M that implements the desired computation. Since, the phase application machine run in time which is independent of w, x, and y, and the other four run in time which depends only on the length of w, x, and y, the running time of M depends on the length of w, x, and y but not on their particular values. Therefore, it halts with the proper output not only if run on an input with a single string w but also if run on an initial superposition with different strings w and with the same near-trivial transformation and ϵ.

The QTM M_2 to carry out near-trivial phase shifts is the same as M_1 except that we replace the transition function of the simple QTM which applies a phase shift

rotation by angle $\mathcal{R}$ as follows giving a stationary QTM which applies phase shift $e^{i\mathcal{R}}$ if b is a 0 and phase shift 1 otherwise.

	#	0	1
q_0	$\|\#\rangle\|q_1\rangle\|L\rangle$	$e^{i\mathcal{R}}\|0\rangle\|q_1\rangle\|L\rangle$	$\|1\rangle\|q_1\rangle\|L\rangle$
q_1	$\|\#\rangle\|q_f\rangle\|R\rangle$		
q_f	$\|\#\rangle\|q_0\rangle\|R\rangle$	$\|0\rangle\|q_0\rangle\|R\rangle$	$\|1\rangle\|q_0\rangle\|R\rangle$

The proof that this gives the desired M_2 is identical to the proof for M_1 and is omitted. □

6.4. Carrying out a unitary transformation on a QTM. Now that we can approximate a unitary transformation by a product of near-trivial transformations, and we have a QTM to carry out the latter, we can build a QTM to apply an approximation of a given unitary transformation.

THEOREM 6.11 (unitary transformation theorem). *There is a stationary, normal form QTM M with first track alphabet $\{\#, 0, 1\}$ that carries out the set of all transformations on its first track in polynomial time with required closeness factor $\frac{1}{2(10\sqrt{d})^d}$ for transformations of dimension d.*

Proof. Given an $\epsilon > 0$ and a transformation U of dimension $d = 2^k$, which is $\frac{\epsilon}{2(10\sqrt{d})^d}$-close to unitary, we can carry out U to within ϵ on the first k cells of the first track using the following steps.

1. Calculate and write on clean tracks $\frac{\epsilon}{2n}$ and a list of near-trivial $U_1, \ldots, U_n$ such that $\|U - U_n \cdots U_1\| \leq \frac{\epsilon}{2}$ and such that n is polynomial in 2^k.
2. Apply the list of transformations $U_1, \ldots, U_n$, each to within $\frac{\epsilon}{2n}$.
3. Erase $U_1, \ldots, U_n$ and $\frac{\epsilon}{2n}$.

We can construct a QTM to accomplish these steps as follows. First, using Theorem 6.7 on page 1442 and the synchronization theorem on page 1428, we can build polynomial time, stationary, normal form QTMs for steps 1 and 3 that run in time, which depend only on U and ϵ. Finally, we can build a stationary, normal form QTM to accomplish step 2 in time, which is polynomial in 2^k, and $\frac{1}{\epsilon}$ as follows. We have a stationary, normal form QTM constructed in Lemma 6.10 on page 1445 to apply any specified near-trivial transformation to within a given bound ϵ. We dovetail this with a machine, constructed using the synchronization theorem on page 1428, that rotates U_1 around to the end of the list of transformations. Since the resulting QTM is stationary and takes time that depends only on ϵ and the U_i, we can insert it for the special state in the machine of the looping lemma to give the desired QTM for step 2.

With the appropriate use of Lemmas 4.4 and 4.5 on page 1428, dovetailing the QTMs for these three steps gives the desired M'. Since the running times of the three QTMs are independent of the contents of the first track, so is the running time of M'. Finally, notice that when we run M' we apply to the first k cells of the first track, we apply a unitary transformation U' with

$$\|U' - U\| \leq \|U' - U_n \cdots U_1\| + \|U_n \cdots U_1 - U\| \leq n\frac{\epsilon}{2n} + \frac{\epsilon}{2} \leq \epsilon$$

as desired. □

7. Constructing a universal QTM. A universal QTM must inevitably decompose one step of the simulated machine using many simple steps. A step of the

simulated QTM is a mapping from the computational basis to a new orthonormal basis. A single step of the universal QTM can only map some of the computational basis vectors to their desired destinations. In general this partial transformation will not be unitary, because the destination vectors will not be orthogonal to the computational bases which have not yet been operated on. The key construction that enables us to achieve a unitary decomposition is the unidirection lemma. Applying this lemma, we get a QTM whose mapping of the computational basis has the following property: there is a decomposition of the space into subspaces of constant dimension such that each subspace gets mapped onto another.

More specifically, we saw in the previous section that we can construct a QTM that carries out to a close approximation any specified unitary transformation. On the other hand, we have also noted that the transition function of a unidirectional QTM specifies a unitary transformation from superpositions of current state and tape symbol to superpositions of new symbol and state. This means a unidirectional QTM can be simulated by repeatedly applying this fixed-dimensional unitary transformation followed by the reversible deterministic transformation that moves the simulated tape head according to the new state. So, our universal machine will first convert its input to a unidirectional QTM using the construction of the unidirection lemma and then simulate this new QTM by repeatedly applying this unitary transformation and reversible deterministic transformation.

Since we wish to construct a single machine that can simulate every QTM, we must build it in such a way that every QTM can be provided as input. Much of the definition of a QTM is easily encoded: we can write down the size of the alphabet Σ, with the first symbol assumed to be the blank symbol, and we can write down the size of the state set Q, with the first state assumed to be the start state. To complete the specification, we need to describe the transition function δ by giving the $2\ \mathrm{card}(\Sigma)^2\ \mathrm{card}(Q)^2$ amplitudes of the form $\delta(i_1, i_2, i_3, i_4, d)$. If we had restricted the definition of a QTM to include only machines with rational transition amplitudes, then we could write down each amplitude explicitly as the ratio of two integers. However, we have instead restricted the definition of a QTM to include only those machines with amplitudes in $\tilde{\mathbf{C}}$, which means that for each amplitude there is a deterministic algorithm which computes the amplitude's real and imaginary parts to within 2^{-n} in time polynomial in n. We will therefore specify δ by giving a deterministic algorithm that computes each transition amplitude to within 2^{-n} in time polynomial in n.

Since the universal QTM that we will construct returns its tape head to the start cell after simulating each step of the desired machine, it will incur a slowdown which is (at least) linear in T. We conjecture that with more care a universal QTM can be constructed whose slowdown is only polylogarithmic in T.

THEOREM 7.1. *There is a normal form QTM $\mathcal{M}$ such that for any well-formed QTM M, any $\epsilon > 0$, and any T, $\mathcal{M}$ can simulate M with accuracy ϵ for T steps with slowdown polynomial in T and $\frac{1}{\epsilon}$.*

Proof. As described above, our approach will be first to use the construction of the unidirection lemma to build a unidirectional QTM M', which simulates M with slowdown by a factor of 5, and then to simulate M'. We will first describe the simulation of M' and then return to describe the easily computable preprocessing that needs to be carried out.

So, suppose $M = (\Sigma, Q, \delta)$ is a unidirectional QTM that we wish to simulate on our universal QTM.

We start by reviewing the standard technique of representing the configuration of a target TM on the tape of a universal TM. We will use one track of the tape of our universal QTM to simulate the current configuration of M. Since the alphabet and state set of M could have any fixed size, we will use a series of $\log\ \mathrm{card}(Q \times \Sigma)$ cells of our tape, referred to as a "supercell," to simulate each cell of M. Each supercell holds a pair of integers p, σ, where $\sigma \in [1, \mathrm{card}(\Sigma)]$ represents the contents of the corresponding cell of M, and $p \in [0, \mathrm{card}(Q)]$ represents the state of M if its tape head is scanning the corresponding cell and $p = 0$ otherwise. Since the tape head of M can only move distance T away from the start cell in time T, we only need supercells for the $2T + 1$ cells at the center of M's tape (and we place markers to denote the ends).

Now we know that if we ignore the update direction, then δ gives a unitary transformation U of dimension $d = \mathrm{card}(Q \times \Sigma)$ from superpositions of current state and tape symbol to superpositions of new state and symbol. So, we can properly update the superposition on the simulation tracks if we first apply U to the current state and symbol of M and then move the new state specification left or right one supercell according to the direction in which that state of M can be entered.

We will therefore build a QTM $STEP$ that carries out one step of the simulation as follows. In addition to the simulation track, this machine is provided as input a desired accuracy γ, a specification of U (which is guaranteed to be $\frac{\gamma}{2(10\sqrt{d})^d}$-close to the unitary), and a string $s \in \{0,1\}^{\mathrm{card}(Q)}$, which gives the direction in which each state of M can be entered. The machine $STEP$ operates as follows.

1. Transfer the current state and symbol $p; \sigma$ to empty workspace near the start cell, leaving a special marker in their places.
2. Apply U to $p; \sigma$ to within γ, transforming $p; \sigma$ into a superposition of new state and symbol $q; \tau$.
3. Reverse step 1, transferring q, τ back to the marked, empty supercell (and emptying the workspace).
4. Transfer the state specification q one supercell to the right or left depending whether the qth bit of s is a 0 or 1.

Using the synchronization theorem on page 1428, we can construct stationary, normal form QTMs for steps 1, 3, and 4 that take time which is polynomial in T and (for a fixed M) depend only on T. Step 2 can be carried out in time polynomial in $\mathrm{card}(\Sigma), \mathrm{card}(Q)$, and γ with the unitary transformation applying QTM constructed in the unitary transformation theorem. With appropriate use of Lemmas 4.4 and 4.5 on page 1428, dovetailing these four normal form QTMs gives us the desired normal form QTM $STEP$.

Since each of the four QTMs takes time that depends (for a fixed M) only on T and ϵ, so does $STEP$. Therefore, if we insert $STEP$ for the special state in the reversible TM constructed in the looping lemma and provide additional input T, the resulting QTM $STEP'$ will halt after time polynomial in T and $\frac{1}{\epsilon}$ after simulating T steps of M with accuracy $T\epsilon$.

Finally, we construct the desired universal QTM $\mathcal{M}$ by dovetailing $STEP'$ after a QTM which carries out the necessary preprocessing. In general, the universal machine must simulate QTMs that are not unidirectional. So, the preprocessing for desired QTM M, desired input x, and desired simulation accuracy ϵ consists of first carrying out the construction of the unidirection lemma to build a unidirectional QTM M'

which simulates M with slowdown by a factor of 5.[3] The following inputs are then computed for $STEP'$:

1. the proper $2T+1$ supercell representation of the initial configuration of M' with input x,
2. the d-dimensional transformation U for M' with each entry written to accuracy $\frac{\epsilon}{40T(10\sqrt{d})^{d+2}}$,
3. the string of directions s for M',
4. the desired number of simulation steps $5T$ and the desired accuracy $\gamma = \frac{\epsilon}{40T}$.

It can be verified that each of these inputs to $\mathcal{M}$ can be computed in deterministic time which is polynomial in T, $\frac{1}{\epsilon}$, and the length of the input. If the transformation U is computed to the specified accuracy, the transformation actually provided to $STEP$ will be within $\frac{\epsilon}{40T(10\sqrt{d})^d}$ of the desired unitary U and therefore will be $\frac{\epsilon}{40T(10\sqrt{d^d})}$-close to unitary as required for the operation of $STEP$. So, each time $STEP$ runs with accuracy $\frac{\epsilon}{40T}$, it will have applied a unitary transformation which is within $\frac{\epsilon}{20T}$ of U. Therefore, after $5T$ runs of $STEP$, we will have applied a unitary transformation which is within $\frac{\epsilon}{4}$ of the $5T$ step transformation of M'. This means that observing the simulation track of $\mathcal{M}$ after it has been completed will give a sample from a distribution which is within total variation distance ϵ of the distribution sampled by observing M on input x at time T. □

8. The computational power of QTMs. In this section, we explore the computational power of QTMs from a complexity theoretic point of view. It is natural to define quantum analogues of classical complexity classes [13]. In classical complexity theory, **BPP** is regarded as the class of all languages that is efficiently computable on a classical computer. The quantum analogue of BPP—BQP (bounded-error quantum polynomial time)—should similarly be regarded as the class of all languages that is efficiently computable on a QTM.

8.1. Accepting languages with QTMs.

DEFINITION 8.1. *Let M be a stationary, normal form, multitrack QTM M whose last track has alphabet $\{\#, 0, 1\}$. If we run M with string x on the first track and the empty string elsewhere, wait until M halts,*[4] *and then observe the last track of the start cell, we will see a* 1 *with some probability p. We will say that M* accepts *x with probability p and* rejects *x with probability $1-p$.*

Consider any language $\mathcal{L} \subseteq (\Sigma - \#)^*$.

We say that QTM M exactly accepts *the $\mathcal{L}$ if M accepts every string $x \in \mathcal{L}$ with probability* 1 *and rejects every string $x \in (\Sigma - \#)^* - \mathcal{L}$ with probability* 1.

We define the class **EQP** (exact *or* error-free quantum polynomial time) *as the set of languages which are exactly accepted by some polynomial time QTM. More generally, we define the class* **EQTime** $(T(n))$ *as the set of languages which are exactly accepted by some QTM whose running time on any input of length n is bounded by $T(n)$.*

A QTM accepts *the language $\mathcal{L} \subseteq (\Sigma - \#)^*$* with probability *$p$ if M accepts with probability at least p every string $x \in \mathcal{L}$ and rejects with probability at least p every string $x \in (\Sigma - \#)^* - \mathcal{L}$. We define the class* **BQP** *as the set of languages that are*

[3] Note that the transition function of M' is again specified with a deterministic algorithm, which depends on the algorithm for the transition function of M.

[4] This can be accomplished by performing a measurement to check whether the machine is in the final state q_f. Making this partial measurement does not have any other effect on the computation of the QTM.

accepted with probability $\frac{2}{3}$ *by some polynomial time QTM. More generally, we define the class* $\mathbf{BQTime}(T(n))$ *as the set of languages that are accepted with probability* $\frac{2}{3}$ *by some QTM whose running time on any input of length* n *is bounded by* $T(n)$.

The limitation of considering only stationary, normal form QTMs in these definitions is easily seen to not limit the computational power by appealing to the stationary, normal form universal QTM constructed in Theorem 7.1 on page 1448.

8.2. Upper and lower bounds on the power of QTMs. Clearly $\mathbf{EQP} \subseteq \mathbf{BQP}$. Since reversible TMs are a special case of QTMs, Bennett's results imply that $\mathbf{P} \subseteq \mathbf{EQP}$ and $\mathbf{BPP} \subseteq \mathbf{BQP}$. We include these two simple proofs for completeness.

THEOREM 8.2. $\mathbf{P} \subseteq \mathbf{EQP}$.

Proof. Let $\mathcal{L}$ be a language in $\mathbf{P}$. Then there is some polynomial time deterministic algorithm that on input x produces output 1 if $x \in \mathcal{L}$ and 0 otherwise. Appealing to the synchronization theorem on page 1428, there is therefore a stationary, normal form QTM running in polynomial time which on input x produces output $x;1$ if $x \in \mathcal{L}$ and $x;0$ otherwise. This is an $\mathbf{EQP}$ machine accepting $\mathcal{L}$. ∎

THEOREM 8.3. $\mathbf{BPP} \subseteq \mathbf{BQP}$.

Proof. Let $\mathcal{L}$ be a language in $\mathbf{BPP}$. Then there must be a polynomial $p(n)$ and a polynomial time deterministic TM M with $0,1$ output which satisfy the following. For any string x of length n, if we call S_x the set of $2^{p(n)}$ bits computed by M on the inputs $x;y$ with $y \in \{0,1\}^{p(n)}$, then the proportion of 1's in S_x is at least $\frac{2}{3}$ when $x \in \mathcal{L}$ and at most $\frac{1}{3}$ otherwise. We can use a QTM to decide whether a string x is in the language $\mathcal{L}$ by first breaking into a superposition split equally among all $|x\rangle|y\rangle$ and then running this deterministic algorithm.

First, we dovetail a stationary, normal form QTM that takes input x to output $x;0^{p(n)}$ with a stationary, normal form QTM constructed as in Theorem 8.8 below which applies a Fourier transform to the contents of its second track. This gives us a stationary, normal form QTM which on input x produces the superposition $\sum_{y\in\{0,1\}^{p(n)}} \frac{1}{2^{p(n)/2}}|x\rangle|y\rangle$. Dovetailing this with a synchronized, normal form version of M built according to the synchronization theorem on page 1428 gives a polynomial time QTM which on input x produces a final superposition

$$\sum_{y\in\{0,1\}^{p(n)}} \frac{1}{2^{p(n)/2}}|x\rangle|y\rangle|M(x;y)\rangle.$$

Since the proportion of 1's in S_x is at least $\frac{2}{3}$ if $x \in \mathcal{L}$ and at most $\frac{1}{3}$ otherwise, observing the bit on the third track will give the proper classification for string x with probability at least $\frac{2}{3}$. Therefore, this is a $\mathbf{BQP}$ machine accepting the language $\mathcal{L}$. ∎

Clearly $\mathbf{BQP}$ is in exponential time. The next result gives the first nontrivial upper bound on $\mathbf{BQP}$.

THEOREM 8.4. $\mathbf{BQP} \subseteq \mathbf{PSPACE}$.

Proof. Let $M = (\Sigma, Q, \delta)$ be a $\mathbf{BQP}$ machine with running time $p(n)$.

According to Theorem 3.9 on page 1423, any QTM M' which is

$$\frac{\epsilon}{24\ \mathrm{card}(\Sigma)\ \mathrm{card}(Q)p(n)}\text{-close}$$

to M will simulate M for $p(n)$ steps with accuracy ϵ. If we simulate M with accuracy $\frac{1}{12}$, then the success probability will still be at least $\frac{7}{12}$. Therefore, we need only work

with the QTM M' where each transition amplitude from M is computed to its first $\log(288\ \mathrm{card}(\Sigma)\ \mathrm{card}(Q)p(n))$ bits.

Now the amplitude of any particular configuration at time T is the sum of the amplitudes of each possible computational path of M' of length T from the start configuration to the desired configuration. The amplitude of each such path can be computed exactly in polynomial time. So, if we maintain a stack of at most $p(n)$ intermediate configurations we can carry out a depth-first search of the computational tree to calculate the amplitude of any particular configuration using only polynomial space (but exponential time).

Finally, we can determine whether a string x of length n is accepted by M by computing the sum of squared magnitudes at time $p(n)$ of all configurations that M' can reach that have a 1 in the start cell and comparing this sum to $\frac{7}{12}$. Clearly the only reachable "accepting" configurations of M' are those with a 1 in the start cell and blanks in all but the $2p(n)$ cells within distance $p(n)$ of the start cell. So, using only polynomial space, we can step through all of these configurations computing a running sum of the their squared magnitudes. □

Following Valiant's suggestion [43], the upper bound can be further improved to $\mathbf{P}^{\sharp\mathbf{P}}$. This proof can be simplified by using a theorem from [9] that shows how any **BQP** machine can be turned into a "clean" version M that on input x produces a final superposition with almost all of its weight on $x; M(x)$ where $M(x)$ is a 1 if M accepts x and a 0 otherwise. This means we need only estimate the amplitude of this one configuration in the final superposition of M.

Now, the amplitude of a single configuration can be broken down into not only the sum of the amplitudes of all of the computational paths that reach it but also into the sum of positive real contributions, the sum of negative real contributions, the sum of positive imaginary contributions, and the sum of negative imaginary contributions. We will show that each of these four pieces can be computed using a $\sharp\mathbf{P}$ machine.

Recall that $\sharp\mathbf{P}$ is the set of functions f mapping strings to integers for which there exists a polynomial $p(n)$ and a language $\mathcal{L} \in \mathbf{P}$ such that for any string x, the value $f(x)$ is the number of strings y of length $p(|x|)$ for which xy is contained in $\mathcal{L}$.

THEOREM 8.5. *If the language $\mathcal{L}$ is contained in the class* $\mathbf{BQTime}(T(n))$ *with $T(n) > n$, with $T(n)$ time constructible, then for any $\epsilon > 0$ there is a QTM M' which accepts $\mathcal{L}$ with probability $1 - \epsilon$ and has the following property. When run on input x of length n, M' runs for time bounded by $cT(n)$, where c is a polynomial in $\log \frac{1}{\epsilon}$ and produces a final superposition in which $|x\rangle|\mathcal{L}(x)\rangle$, with $\mathcal{L}(x) = 1$ if $x \in \mathcal{L}$ and 0 otherwise, has squared magnitude at least $1 - \epsilon$.*

THEOREM 8.6. $\mathbf{BQP} \subseteq \mathbf{P}^{\sharp\mathbf{P}}$.

Proof. Let $M = (\Sigma, Q, \delta)$ be a **BQP** machine with observation time $p(n)$. Appealing to Theorem 8.5 above, we can conclude without loss of generality that M is a "clean **BQP** machine." For example, on any input x at time $p(|x|)$, the squared magnitude of the final configuration with output $x; M(x)$ is at least $\frac{2}{3}$.

Now as above, we will appeal to Theorem 3.9 on page 1423 which tells us that if we work with the QTM M', where each amplitude of M is computed to its first $b = \log(288\ \mathrm{card}(\Sigma)\ \mathrm{card}(Q)p(n))$ bits, and we run M' on input x, then the squared magnitude at time $p(|x|)$ of the final configuration with output $x; M(x)$ will be at least $\frac{7}{12}$.

We will carry out the proof by showing how to use an oracle for the class $\sharp\mathbf{P}$ to efficiently compute with error magnitude less than $\frac{1}{36}$ the amplitude of the final configuration $x; 1$ of M' at time T. Since the true amplitude has magnitude at most

1, the squared magnitude of this approximated amplitude must be within $\frac{1}{12}$ of the squared magnitude of the true amplitude. To see this, just note that if α is the true amplitude and $\|\alpha' - \alpha\| < \frac{1}{36}$, then

$$\left| \|\alpha\|^2 - \|\alpha'\|^2 \right| \leq \|\alpha' - \alpha\|^2 + 2\|\alpha\|\|\alpha' - \alpha\| \leq \frac{1}{36}\left(2 + \frac{1}{36}\right) < \frac{1}{12}.$$

Since the success probability of M' is at least $\frac{7}{12}$, comparing the squared magnitude of this approximated amplitude to $\frac{1}{2}$ lets us correctly classify the string x.

We will now show how to approximate the amplitude described above. First, notice that the amplitude of a configuration at time T is the sum of the amplitudes of each computational path of length T from the start configuration to the desired configuration. The amplitude of any particular path is the product of the amplitudes in the transition function of M' used in each step along the path. Since each amplitude consists of a real part and an imaginary part, we can think of this product as consisting of the sum of 2^T terms each of which is either purely real or purely imaginary. So, the amplitude of the desired configuration at time T is the sum of these 2^T terms over each path. We will break this sum into four pieces, the sum of the positive real terms, the sum of the negative real terms, the sum of the positive imaginary terms, and the sum of the negative imaginary terms, and we will compute each to within error magnitude less than $\frac{1}{144}$ with the aid of a $\sharp\mathbf{P}$ algorithm. Taking the difference of the first two and the last two will then give us the amplitude of the desired configuration to within an error of magnitude at most $\frac{1}{36}$ as desired.

We can compute the sum of the positive real contributions for all paths as follows. Suppose for some fixed constant c which is polynomial in T we are given the following three inputs, all of length polynomial in T: a specification of a T step computational path p of M', a specification t of one of the 2^T terms, and an integer w between 0 and 2^{cT}. Then it is easy to see that we could decide in deterministic time polynomial in T whether p is really a path from the start configuration of M on x to the desired final configuration, whether the term t is real and positive, and whether the tth term of the amplitude for path p is greater than $w/2^{cT}$. If we fix a path p and term t satisfying these constraints, then the number of w for which this algorithm accepts, divided by 2^{cT}, is within $1/2^{cT}$ of the value of the tth for path p. So, if we fix only a path p satisfying these constraints, then the number of t, w for which the algorithm accepts, divided by 2^{cT}, is within $1/2^{(c-1)T}$ of the sum of the positive real terms for path p. Therefore, the number of p, t, w for which the algorithm accepts, divided by 2^{cT}, is within $N/2^{(c-1)T}$ of the sum of all of the positive real terms of all of the T step paths of M' from the start configuration to the desired configuration. Since there are at most $2\text{card}(\Sigma)\text{card}(Q)$ possible successors for any configuration in a legal path of M, choosing $c > 1 + \frac{\log 144}{T} + \frac{2\text{card}(\Sigma)\text{card}(Q)\log T}{T}$ gives $N/2^{(c-1)T} < \frac{1}{144}$ as desired. Similar reasoning gives $\mathbf{P}^{\sharp\mathbf{P}}$ algorithms for approximating each of the remaining three sums of terms. □

8.3. Oracle QTMs. In this subsection and the next, we will assume without loss of generality that the TM alphabet for each track is $\{0, 1, \#\}$. Initially all tracks are blank except that the input track contains the actual input surrounded by blanks. We will use Σ to denote $\{0, 1\}$.

In the classical setting, an oracle may be described informally as a device for evaluating some Boolean function $f : \Sigma^* \to \Sigma$, on arbitrary arguments, at unit cost per evaluation. This allows us to formulate questions such as, "if f were efficiently computable by a TM, which other functions (or languages) could be efficiently computed

by TMs?" In this section we define oracle QTMs so that the equivalent question can be asked in the quantum setting.

An oracle QTM has a special *query track* on which the machine will place its questions for the oracle. Oracle QTMs have two distinguished internal states: a prequery state q_q and a postquery state q_a. A query is executed whenever the machine enters the prequery state with a single (nonempty) block of nonblank cells on the query track.[5] Assume that the nonblank region on the query tape is in state $|x \cdot b\rangle$ when the prequery state is entered, where $x \in \Sigma^*$, $b \in \Sigma$, and "$\cdot$" denotes concatenation. Let f be the Boolean function computed by the oracle. The result of the oracle call is that the state of the query tape becomes $|x \cdot b \oplus f(x)\rangle$, where "$\oplus$" denotes the exclusive-or (addition modulo 2), after which the machine's internal control passes to the postquery state. Except for the query tape and internal control, other parts of the oracle QTM do not change during the query. If the target bit $|b\rangle$ was supplied in initial state $|0\rangle$, then its final state will be $|f(x)\rangle$, just as in a classical oracle machine. Conversely, if the target bit is already in state $|f(x)\rangle$, calling the oracle will reset it to $|0\rangle$, a process known as "uncomputing," which is essential for proper interference to take place.

The power of quantum computers comes from their ability to follow a coherent superposition of computation paths. Similarly, oracle QTMs derive great power from the ability to perform superpositions of queries. For example, an oracle for Boolean function f might be called when the query tape is in state $|\psi, 0\rangle = \sum_x \alpha_x |x, 0\rangle$, where α_x are complex coefficients, corresponding to an arbitrary superposition of queries with a constant $|0\rangle$ in the target bit. In this case, after the query, the query string will be left in the entangled state $\sum_x \alpha_x |x, f(x)\rangle$.

That the above definition of oracle QTMs yields unitary evolutions is self-evident if we restrict ourselves to machines that are well formed in other respects, in particular evolving unitarily as they enter the prequery state and leave the postquery state.

Let us define $\mathbf{BQTime}\,(T(n))^O$ as the sets of languages accepted with probability at least $\frac{2}{3}$ by some oracle QTM M^O whose running time is bounded by $T(n)$. This bound on the running time applies to each individual input, not just on the average. Notice that whether or not M^O is a **BQP**-machine might depend upon the oracle O; thus M^O might be a **BQP**-machine while $M^{O'}$ might not be one.

We have carefully defined oracle QTMs so that the same technique used to reverse a QTM in the reversal lemma can also be used to reverse an oracle QTM.

LEMMA 8.7. *If M is a normal form, unidirectional oracle QTM, then there is a normal form oracle QTM M' such that for any oracle O, M'^O reverses the computation of M^O while taking two extra time steps.*

Proof. Let $M = (\Sigma, Q, \delta)$ be a normal form, unidirectional oracle QTM with initial and final states q_0 and q_f and with query states $q_q \neq q_f$ and q_a. We construct M' from M exactly as in the proof of the reversal lemma. We further give M' the same query states q_q, q_a as M but with roles reversed. Since M is in normal form, and $q_q \neq q_f$, we must have $q_a \neq q_0$. Recall that the transition function of M' is defined so that for $q \neq q_0$

$$\delta'(q, \tau) = \sum_{p,\sigma} \delta(p, \sigma, \tau, q, d_q)^* |\sigma\rangle |p\rangle |\hat{d}_p\rangle.$$

[5] Since a QTM must be careful to avoid leaving around intermediate computations on its tape, requiring that the query track contains only the query string, adds no further difficulty to the construction of oracle QTMs.

Therefore, since state q_q always leads to state q_a in M, state q_a always leads to state q_q in M'. Therefore, M' is an oracle QTM.

Next, note that since the tape head position is irrelevant for the functioning of the oracle, the operation of the oracle in M'^O reverses the operation of the oracle in M^O. Finally, the same argument used in the reversal lemma can be used again here to prove that M'^O is well formed and that M'^O reverses the computation of M^O while taking two extra time steps. □

The above definition of a quantum oracle for an arbitrary Boolean function will suffice for the purposes of the present paper, but the ability of quantum computers to perform general unitary transformations suggests a broader definition, which may be useful in other contexts. For example, oracles that perform more general, non-Boolean unitary operations have been considered in computational learning theory [15] and have been used to obtain large separations [32] between quantum and classical relativized complexity classes. See [9] for a discussion of more general definitions of oracle quantum computing.

8.4. Fourier sampling and the power of QTMs. In this section we give evidence that QTMs are more powerful than bounded-error probabilistic TMs. We define the recursive Fourier sampling problem which on input the program for a Boolean function takes on value 0 or 1. We show that the recursive Fourier sampling problem is in **BQP**. On the other hand, we prove that if the Boolean function is specified by an oracle, then the recursive Fourier sampling problem is not in $\mathbf{BQTime}(n^{o(\log n)})$. This result provided the first evidence that QTMs are more powerful than classical probabilistic TMs with bounded error probability [11].

One could ask what the relevance of these oracle results is, in view of the non-relativizing results on probabilistically checkable proofs [36, 3]. Moreover, Arora, Impagliazzo, and Vazirani [2] make the case that the fact that the **P** versus **NP** question relativizes does not imply that the question "cannot be resolved by current techniques in complexity theory." On the other hand, in our oracle results (and also in the subsequent results of [39]), the key property of oracles that is exploited is their black-box nature, and for the reasons sketched below, these results did indeed provide strong evidence that $\mathbf{BQP} \neq \mathbf{BPP}$ (of course, the later results of [36] gave even stronger evidence). This is because if one assumes that $\mathbf{P} \neq \mathbf{NP}$ and the existence of one-way functions (both these hypotheses are unproven but widely believed by complexity theorists), it is also reasonable to assume that, in general, it is impossible to (efficiently) figure out the function computed by a program by just looking at its text (i.e., without explicitly running it on various inputs). Such a program would have to be treated as a black box. Of course, such assumptions cannot be made if the question at hand is whether $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Fourier sampling. Consider the vector space of complex-valued functions $f : Z_2^n \to \mathbf{C}$. There is a natural inner product on this space given by

$$(f, g) = \sum_{x \in Z_2^n} f(x) g(x)^*.$$

The standard orthonormal basis for the vector space is the set of delta functions $\delta_y : Z_2^n \to \mathbf{C}$, given by $\delta_y(y) = 1$ and $\delta_y(x) = 0$ for $x \neq y$. Expressing a function in this basis is equivalent to specifying its values at each point in the domain. The characters of the group Z_2^n yield a different orthonormal basis consisting of the parity basis functions $\chi_s : Z_2^n \to \mathbf{C}$, given by $\chi_s(x) = \frac{-1^{s \cdot x}}{2^{n/2}}$, where $s \cdot x = \sum_{i=1}^n s_i x_i$. Given

any function $f : Z_2^n \to \mathbf{C}$, we may write it in the new parity basis as $f = \sum_s \hat{f}(s)\chi_s$, where $\hat{f} : Z_2^n \to \mathbf{C}$ is given by $\hat{f}(s) = (f, \chi_s)$. The function $\hat{f}$ is called the discrete Fourier transform or Hadamard transform of f. Here the latter name refers to the fact that the linear transformation that maps a function f to its Fourier transform $\hat{f}$ is the Hadamard matrix $\mathcal{H}_n$. Clearly this transformation is unitary, since it is just effecting a change of basis from one orthonormal basis (the delta function basis) to another (the parity function basis). It follows that $\sum_x |f(x)|^2 = \sum_s |\hat{f}(s)|^2$.

Moreover, the discrete Fourier transform on Z_2^n can be written as the Kronecker product of the transform on each of n copies of Z_2—the transform in that case is given by the matrix

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}.$$

This fact can be used to write a simple and efficient QTM that affects the discrete Fourier transformation in the following sense: suppose the QTM has tape alphabet $\{0, 1, \#\}$ and the cells initially that have only n tape contain nonblank symbols. Then we can express the initial superposition as $\sum_{x\in\{0,1\}^n} f(x)|x\rangle$, where $f(x)$ is the amplitude of the configuration with x in the nonblank cells. Then there is a QTM that affects the Fourier transformation by applying the above transform on each of the n bits in the n nonblank cells. This takes $O(n)$ steps and results in the final configuration $\sum_{s\in\{0,1\}^n} \hat{f}(s)|s\rangle$ (Theorem 8.8 below). Notice that this Fourier transformation is taking place over a vector space of dimension 2^n. On the other hand, the result of the Fourier transform $\hat{f}$ resides in the amplitudes of the quantum superposition and is thus not directly accessible. We can, however, perform a measurement on the n nonblank tape cells to obtain a sample from the probability distribution P such that $P[s] = |\hat{f}(s)|^2$. We shall refer to this operation as Fourier sampling. As we shall see, this is a very powerful operation. The reason is that each value $\hat{f}(s)$ depends upon all the (exponentially many values) of f, and it does not seem possible to anticipate which values of s will have large probability (constructive interference) without actually carrying out an exponential search.

Given a Boolean function $g : Z_2^n \to \{1, -1\}$, we may define a function $f : Z_2^n \to \mathbf{C}$ of norm 1 by letting $f(x) = \frac{g(x)}{2^{n/2}}$. Following Deutsch and Jozsa [22], if g is a polynomial time computable function, there is a QTM that produces the superposition $\sum_{x\in\{0,1\}^n} g(x)|x\rangle$ in time polynomial in n. Combining this with the Fourier sampling operation above, we get a polynomial time QTM that samples from a certain distribution related to the given polynomial time computable function g (Theorem 8.9 below). We shall call this composite operation Fourier sampling with respect to g.

THEOREM 8.8. *There is a normal form QTM which when run on an initial superposition of n-bit strings $|\phi\rangle$ halts in time $2n + 4$ with its tape head back in the start cell and produces final superposition $\mathcal{H}_n|\phi\rangle$.*

Proof. We can construct the desired machine using the alphabet $\{\#, 0, 1\}$ and the set of states $\{q_0, q_a, q_b, q_c, q_f\}$. The machine will operate as follows when started with a string of length n as input. In state q_0, the machine steps right and left and enters state q_b. In state q_b, the machine steps right along the input string until it reaches the # at the end and enters state q_c stepping back left to the string's last symbol. During this rightward scan, the machine applies the transformation

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

to the bit in each cell. In state q_c, the machine steps left until it reaches the # to the left of the start cell, at which point it steps back right and halts. The following gives the quantum transition function of the desired machine.

	#	0	1
q_0		$\|0\rangle\|q_a\rangle\|R\rangle$	$\|1\rangle\|q_a\rangle\|R\rangle$
q_a	$\|\#\rangle\|q_b\rangle\|L\rangle$		
q_b	$\|\#\rangle\|q_c\rangle\|L\rangle$	$\frac{1}{\sqrt{2}}\|0\rangle\|q_b\rangle\|R\rangle + \frac{1}{\sqrt{2}}\|1\rangle\|q_b\rangle\|R\rangle$	$\frac{1}{\sqrt{2}}\|0\rangle\|q_b\rangle\|R\rangle - \frac{1}{\sqrt{2}}\|1\rangle\|q_b\rangle\|R\rangle$
q_c	$\|\#\rangle\|q_f\rangle\|L\rangle$	$\|0\rangle\|q_c\rangle\|L\rangle$	$\|1\rangle\|q_c\rangle\|L\rangle$
q_f	$\|\#\rangle\|q_0\rangle\|R\rangle$	$\|0\rangle\|q_0\rangle\|R\rangle$	$\|1\rangle\|q_0\rangle\|R\rangle$

It is can be easily verified that the completion lemma can be used to extend this machine to a well-formed QTM and that this machine has the desired behavior described above. □

THEOREM 8.9. *For every polynomial time computable function $g : \{0,1\}^n \to \{-1,1\}$, there is a polynomial time, stationary QTM where observing the final superposition on input 0^n gives each n-bit string s with probability $|\hat{f}(s)|^2$, where $f(x) = \frac{g(x)}{2^{n/2}}$.*

Proof. We build a QTM to carry out the following steps.

1. Apply a Fourier transform to 0^n to produce $\sum_{i\in\{0,1\}^n} \frac{1}{2^{n/2}}|i\rangle$.
2. Compute f to produce $\sum_{i\in\{0,1\}^n} \frac{1}{2^{n/2}}|i\rangle|f(i)\rangle$.
3. Apply a phase-applying machine to produce $\sum_{i\in\{0,1\}^n} \frac{1}{2^{n/2}}f(i)|i\rangle|f(i)\rangle$.
4. Apply the reverse of the machine in step 2 to produce $\sum_{i\in\{0,1\}^n} \frac{1}{2^{n/2}}f(i)|i\rangle$.
5. Apply another Fourier transform to produce $\sum_{i\in\{0,1\}^n} \frac{1}{2^{n/2}}\hat{f}(s)|s\rangle$ as desired.

We have already constructed above the Fourier transform machine for steps 1 and 5. Since f can be computed in deterministic polynomial time, the synchronization theorem on page 1428 lets us construct polynomial time QTMs for steps 2 and 4 which take input $|i\rangle$ to output $|i\rangle|f(i)\rangle$ and vice versa, with running time independent of the particular value of i.

Finally, we can apply phase according to $f(i)$ in step 3 by extending to two tracks the stationary, normal form QTM with alphabet $\{\#, -1, 1\}$ defined by

	#	-1	1
q_0	$\|\#\rangle\|q_1\rangle\|R\rangle$	$-\|-1\rangle\|q_1\rangle\|R\rangle$	$\|1\rangle\|q_1\rangle\|R\rangle$
q_1	$\|\#\rangle\|q_f\rangle\|L\rangle$	$\|-1\rangle\|q_f\rangle\|L\rangle$	$\|1\rangle\|q_f\rangle\|L\rangle$
q_f	$\|\#\rangle\|q_0\rangle\|R\rangle$	$\|-1\rangle\|q_0\rangle\|R\rangle$	$\|1\rangle\|q_0\rangle\|R\rangle$

Dovetailing these five stationary, normal form QTMs gives the desired machine. □

Remarkably, this quantum algorithm performs Fourier sampling with respect to f while calling the algorithm for f only twice. To see more clearly why this is remarkable, consider the special case of the sampling problem where the function f is one of the (unnormalized) parity functions. Suppose f corresponds to the parity function χ_k; i.e., $f(i) = (-1)^{i\cdot k}$, where $i, k \in \{0,1\}^n$. Then the result of Fourier sampling with respect to f is always k. Let us call this promise problem the parity problem: on input a program that computes f where f is an (unnormalized) parity function, determine k. Notice that the QTM for Fourier sampling extracts n bits of information (the value of k) using just two invocations of the subroutine for computing Boolean function f. It is not hard to show that if f is specified by an oracle, then in a probabilistic setting, extracting n bits of information must require at least n invocations of the Boolean function. We now show how to amplify this advantage of quantum computers over

probabilistic computers by defining a recursive version of the parity problem: the recursive Fourier sampling problem.

To ready this problem for recursion, we first need to turn the parity problem into a problem with range $\{-1, 1\}$. This is easily done by adding a second function g and requiring the answer $g(k)$ rather than k itself.

Now, we will make the problem recursive. For each problem instance of size n, we will replace the 2^n values of f with 2^n independent recursive subproblems of size $\frac{n}{2}$, and so on, stopping the recursion with function calls at the bottom. Since a QTM needs only two calls to f to solve the parity problem, it will be able to solve an instance of the recursive problem of size n recursively in time $T(n)$, where

$$T(n) \;\leq\; poly(n) + 4T\left(\frac{n}{2}\right) \qquad \Longrightarrow \qquad T(n) \leq poly(n).$$

However, since a PTM needs n calls to f to solve the parity problem, the straightforward recursive solution to the recursive problem on a PTM will require time $T(n)$, where

$$T(n) \;\geq\; nT\left(\frac{n}{2}\right) \qquad \Longrightarrow \qquad T(n) \;\geq\; n^{\log n}.$$

To allow us to prove a separation between quantum and classical machines, we will replace the functions with an oracle. For any "legal" oracle O, any oracle satisfying some special constraints to be described below, we will define the language $\mathcal{R}_O$. We will show that there is a QTM which, given access to any legal oracle O, accepts $\mathcal{R}_O$ in time $O(n \log n)$ with success probability 1 but that there is a legal oracle O such that $\mathcal{R}_O$ is not contained in $\mathbf{BPTime}\left(n^{o(\log n)}\right)^O$.

Our language will consist of strings from $\{0, 1\}^*$ with length a power of 2. We will call all such strings *candidates*. The decision whether a candidate x is contained in the language will depend on a recursive tree. If candidate x has length n, then it will have $2^{n/2}$ children in the tree, and in general a node at level $l \geq 0$ (counting from the bottom with leaves at level -1 for convenience) will have 2^{2^l} children. So, the root of the tree for a candidate x of length n is identified by the string x, its $2^{n/2}$ children are each identified by a string $x\$x_1$ where $x_1 \in \{0, 1\}^{n/2}$, and, in general, a descendent in the tree at level l is identified by a string of the form $x\$x_1\$x_2\$ \ldots \$x_m$ with $|x| = n, |x_1| = \frac{n}{2}, \ldots, |x_m| = 2^{l+1}$. Notice that any string of this form for some n a power of 2 and $l \in [0, \log n - 1]$ defines a node in some tree. So, we will consider oracles O which map queries over the alphabet $\{0, 1, \$\}$ to answers $\{-1, +1\}$, and we will use each such oracle O to define a function V_O that gives each node a value $\{-1, +1\}$. The language $\mathcal{R}_O$ will contain exactly those candidates x for which $V_O(x) = 1$.

We will define V_O for leaves (level 0 nodes) based directly on the oracle O. So, if x is a leaf, then we let $V_O(x) = O(x)$.

We will define V_O for all other nodes by looking at the answer of O to a particular query which is chosen depending on the values of V_O for its children. Consider node x at level $l \geq 0$. We will insist that the oracle O be such that there is some $k_x \in \{0, 1\}^{2^l}$ such that the children of x all have values determined by their parity with k_x

$$\forall y \in \{0, 1\}^{2^l}, \quad V_O(x\$y) = (-1)^{y \cdot k_x},$$

and we will then give $V_O(x)$ the value $O(x\$k_x)$. We will say that the oracle O is *legal* if this process allows us to successfully define V_O for all nodes whose level is ≥ 0. Any

query of the form $x\$k$ with x a node at level $l \geq 0$ and $k \in \{0,1\}^{2^l}$, or of the form x with x a leaf, is called a *query at node* x. A query which is located in the same recursive tree as node x, but not in the subtree rooted at x, is called *outside of* x. Notice that for any candidate x, the values of V_O at nodes in the tree rooted at x and the decision whether x is in $\mathcal{R}_O$ all depend only on the answers of queries located at nodes in the tree rooted at x.

THEOREM 8.10. *There is an oracle QTM M such that for every legal oracle O, M^O runs in polynomial time and accepts the language $\mathcal{R}_O$ with probability* 1.

Proof. We have built the language $\mathcal{R}_O$ so that it can be accepted efficiently using a recursive quantum algorithm. To avoid working through the details required to implement a recursive algorithm reversibly, we will instead implement the algorithm with a machine that writes down an iteration that "unwraps" the desired recursion. Then the looping lemma will let us build a QTM to carry out this iteration.

First consider the recursive algorithm to compute V_O for a node x. If x is a leaf, then the value $V_O(x)$ can be found by querying the string $x\$$. If x is a node at level $l \geq 0$, then calculate V_O as follows.

1. Split into an equal superposition of the 2^{2^l} children of x.
2. Recursively compute V_O for these children in superposition.
3. Apply phase to each child given by that child's value V_O.
4. Reverse the computation of step 2 to erase the value V_O.
5. Apply the Fourier transform converting the superposition of children of x into a superposition consisting entirely of the single string k_x.
6. Query $x\$k_x$ to find the value V_O for x.
7. Reverse steps 1–5 to erase k_x (leaving only x and $V_O(x)$).

Notice that the number of steps required in the iteration obeys the recursion discussed above and hence is polynomial in the length of x. So, we can use the synchronization theorem on page 1428 to construct a polynomial time QTM which, for any particular x, writes a list of the steps which must be carried out. Therefore, we complete the proof by constructing a polynomial time QTM to carry out any such list of steps.

Since our algorithm requires us to recursively run both the algorithm and its reverse, we need to see how to handle each step and its reverse. Before we start, we will fill out the node x to a description of a leaf by adding strings of 0's. Then, steps 1 and 5 at level $l \geq 0$ just require applying the Fourier transform QTM to the 2^l bit string at level l in the current node description. Since the Fourier transform is its own reverse, the same machine also reverses steps 1 and 5. Step 3 is handled by the phase-applying machine already constructed above as part of the Fourier sampling QTM in Theorem 8.9. Again, the transformation in step 3 is its own reverse, so the same machine can be used to reverse step 3. Step 6 and its reverse can be handled by a reversible TM that copies the relevant part of the current node description, queries the oracle, and then returns the node description from the query tape. Notice that each of these machines takes time which depends only on the length of its input.

Since we have stationary, normal form QTMs to handle each step at level l and its reverse in time bounded by a polynomial in 2^l, we can use the branching lemma to construct a stationary, normal form QTM to carry out any specified step of the computation. Dovetailing with a machine which rotates the first step in the list to the end and inserting the resulting machine into the reversible TM of the looping lemma gives the desired QTM. □

Computing the function V_O takes time $\Omega(n^{\log n})$ even for a probabilistic computer that is allowed a bounded probability of error. We can see this with the following intuition. First, consider asking some set of queries of a legal O that determine the value of V_O for a node x described by the string $x_1\$ \dots \$x_m$ at level l. There are two ways that the asked queries might fix the value at x. The first is that the queries outside of the subtree rooted at x might be enough to fix V_O for x. If this happens, we say that $V_O(x)$ is *fixed by constraint*. An example of this is that if we asked all of the queries in the subtrees rooted at all of the siblings of x, then we have fixed V_O for all of the siblings, thereby fixing the string k such that $V_O(x_1\$ \dots \$x_m\$y)$ equals $(-1)^{y \cdot k}$.

The only other way that the value of the node x might be fixed is the following. If the queries fix the value of V_O for some of the children of x then this will restrict the possible values for the string k_x such that $V_O(x\$y)$ always equals $(-1)^{y \oplus k_x}$ and such that $V_O(x) = O(x\$k_x)$. If the query $x\$k_x$ for each possible k_x has been asked and all have the same answers, then this fixes the value V_O at x. If the query $x\$k_x$ for the correct k_x has been asked, then we call x a *hit*.

Now notice that fixing the values of a set of children of a level l node x restricts the value of k_x to a set of $2^{2^l - c}$ possibilities, where c is the maximum size of a linearly independent subset of the children whose values are fixed. This can be used to prove that it takes $n \cdot \frac{n}{2} \cdots 1 = n^{\Omega(\log n)}$.

However, this intuition is not yet enough since we are interested in arguing against a probabilistic TM rather than a deterministic TM. So, we will argue not just that it takes $n^{\Omega(\log n)}$ queries to fix the value of a candidate of length n but that if fewer than $n^{\Omega(\log n)}$ queries are fixed, then choosing a random legal oracle consistent with those queries gives to a candidate of length n the value 1 with probability extremely close to $\frac{1}{2}$. This will give the desired result. To see this, call the set of queries actually asked and the answers given to those queries a *run* of the probabilistic TM. We will have shown that if we take any run on a candidate of length n with fewer than $n^{\Omega(\log n)}$ queries, then the probability that a random legal oracle agreeing with the run assigns to x the value 1 is extremely close to $\frac{1}{2}$. This means that a probabilistic TM whose running time is $n^{o(\log n/2)}$ will fail with probability 1 to accept $\mathcal{R}_O$ for a random legal oracle O.

DEFINITION 8.11. *A* run of size k *is defined as a pair* S, f *where* S *is a set of* k *query strings and* f *is map from* S *to* $\{0,1\}$ *such that there is at least one legal oracle agreeing with* f.

Let r *be a run, let* y *be a node at level* $l \geq 2$, *and let* O *be a legal oracle at and below* y *which agrees with* r. *Then* O *determines the string* k_y *for which* $V_O(y) = O(y\$k_y)$. *If* $y\$k_y$ *is a query in* r, *then we say* O makes y a hit for r. *Suppressing the dependency on* r *in the notation, we define* $P(y)$ *as the probability that* y *is a hit for* r *when we choose a legal oracle at and below* y *uniformly at random from the set of all such oracles at and below* y *which agree with* r. *Similarly, for* x *an ancestor of* y, *we define* $P_x(y)$ *as the probability that* y *is a hit when a legal oracle is chosen at and below* x *uniformly at random from the set of all oracles at and below* x *which agree with* r.

LEMMA 8.12. $P_x(y) \leq 2P(y)$.

Proof. Let S be the set of legal oracles at and below y which agree with r. We can write S as the disjoint union of S_h and S_n, where the former is the set of those oracles in S that make y a hit for r. Further splitting S_h and S_n according to the value $V_O(y)$, we can write S as the disjoint union of four sets $S_{h+}, S_{h-}, S_{n+}, S_{n-}$.

Using this notation, we have $P(y) = \frac{\text{card}(S_h)}{\text{card}(S_h)+\text{card}(S_n)}$. It is easy to see that, since the oracles in S_n do not make y a hit for r, $\text{card}(S_{n+}) = \text{card}(S_{n-}) = \frac{\text{card}(S_n)}{2}$.

Next consider the set T of all legal oracles defined at and below x, but outside y, which agree with r. Each oracle $O \in T$ determines by constraint the value $V_O(y)$ but leaves the string k_y completely undetermined. If we again write T as the disjoint union of T_+ and T_- according to the constrained value $V_O(y)$, we notice that the set of legal oracles at and below x is exactly $(T_+ \times S_+) \cup (T_- \times S_-)$. So, we have

$$\begin{aligned} P_x(y) &= \frac{\text{card}(T_+)\text{card}(S_{h+}) + \text{card}(T_-)\text{card}(S_{h-})}{\text{card}(T_+)\text{card}(S_+) + \text{card}(T_-)\text{card}(S_-)} \\ &= \frac{\text{card}(T_+)\text{card}(S_{h+}) + \text{card}(T_-)\text{card}(S_{h-})}{\text{card}(T_+)\text{card}(S_{h+}) + \text{card}(T_-)\text{card}(S_{h-}) + \text{card}(T)\text{card}(S_n)/2}. \end{aligned}$$

Without loss of generality, let $\text{card}(T_+) \geq \text{card}(T_-)$. Then since $\frac{n}{n+c}$ with $c, n > 0$ increases with n, we have

$$\begin{aligned} P_x(y) &\leq \frac{\text{card}(T_+)\text{card}(S_h)}{\text{card}(T_+)\text{card}(S_h) + \text{card}(T)\text{card}(S_n)/2} \\ &\leq \frac{\text{card}(T_+)\text{card}(S_h)}{\text{card}(T_+)\text{card}(S_h) + \text{card}(T_+)\text{card}(S_n)/2} \\ &= \frac{2\text{card}(S_h)}{2\text{card}(S_h) + \text{card}(S_n)} \;\leq\; 2P(y). \qquad \square \end{aligned}$$

For a positive integer n, we define $\gamma(n) = n(\frac{n}{2}) \cdots 1$. Notice that $\gamma(n) > n^{(\log n)/2}$.

THEOREM 8.13. *Suppose r is a run, y is a node at level $l \geq 2$ with q queries from r at or below y, and x is an ancestor of y. Then $P_x(y) \leq \frac{q}{\gamma(n/4)}$ where $n = 2^l$.*

Proof. We prove the theorem by induction on l.

So, fix a run r and a node y at level 2 with q queries from r at or below y. If $q = 0$, y can never be a hit. So, certainly the probability that y is a hit is at most q as desired.

Next, we perform the inductive step. So, assume the theorem holds true for any r and y at level less than l with $l \geq 2$. Then, fix a run r and a node y at level l with q queries from r at or below y. Let $n = 2^l$. We will show that $P(y) \leq \frac{q}{2\gamma(n/4)}$, and then the theorem will follow from Lemma 8.12. So, for the remainder of the proof, all probabilities are taken over the choice of a legal oracle at and below y uniformly at random from the set of all those legal oracles at and below y which agree with r.

Now, suppose that q' of the q queries are actually at y. Clearly, if we condition on there being no hits among the children of y, then k_y will be chosen uniformly among all n-bit strings, and hence the probability y is a hit would be $\frac{q'}{2^n}$. If we instead condition on there being exactly c hits among the 2^n children of y, then the probability that y is a hit must be at most $\frac{q'}{2^{n-c}}$. Therefore, the probability y is a hit is bounded above by the sum of $\frac{q'}{2^{n/2}}$ and the probability that at least $\frac{n}{2}$ of the children of y are hits.

Now consider any child z of y. Applying the inductive hypothesis with y and z taking the roles of x and y, we know that if r has q_z queries at and below z, then $P_y(z) \leq \frac{q_z}{\gamma(n/8)}$. Therefore the *expected* number of hits among the children of y is at most $\frac{(q-q')}{\gamma(n/8)}$. This means that the probability that at least $\frac{n}{2}$ of the children of y are

hits is at most

$$\frac{(q-q')}{\gamma(n/8)n/2} \;=\; \frac{(q-q')}{2\gamma(n/4)}.$$

Therefore,

$$P(y) \;\leq\; \frac{q-q'}{2\gamma(n/4)} + \frac{q'}{2^{n/2}} \;\leq\; \frac{q}{2\gamma(n/4)}$$

since $2\gamma(\frac{n}{4}) < 2^{n/2}$ for $n > 4$. □

COROLLARY 8.14. *For any $T(n)$ which is $n^{o(\log n)}$ relative to a random legal oracle O, with probability 1, $\mathcal{R}_O$ is not contained in* $\mathbf{BPTime}(T(n))$.

Proof. Fix $T(n)$ which is $n^{o(\log n)}$.

We will show that for any probabilistic TM M, when we pick a random legal oracle O, with probability 1, M^O either fails to run in time $cn^{o(\log n)}$ or it fails to accept $\mathcal{R}_O$ with error probability bounded by $\frac{1}{3}$. Then since there are a countable number of probabilistic TMs and the intersection of a countable number of probability 1 events still has probability 1, we conclude that with probability 1, $\mathcal{R}_O$ is not contained in $\mathbf{BPTime}(n^{o(\log n)})$.

We prove the corollary by showing that, for large enough n, the probability that M^O runs in time greater than $T(n)$ or has error greater than $\frac{1}{3}$ on input 0^n is at least $\frac{1}{8}$ for every way of fixing the oracle answers for trees other than the tree rooted at 0^n. The probability is taken over the random choices of the oracle for the tree rooted at 0^n.

Arbitrarily fix a legal behavior for O on all trees other than the one rooted at 0^n. Then consider picking a legal behavior for O for the tree rooted at 0^n uniformly at random and run M^O on input 0^n. We can classify runs of M^O on input 0^n based on the run r that lists the queries the machines asks and the answers it receives. If we take all probabilities over both the randomness in M and the choice of oracle O, then the probability that M^O correctly classifies 0^n in time $T(n)$ is

$$\sum_r \Pr[r]\Pr[\text{correct} \mid r],$$

where $\Pr[r]$ is the probability of run r, where $\Pr[\text{correct} \mid r]$ is the probability the answer is correct given run r, and where r ranges over all runs with at most $T(n)$ queries. Theorem 8.13 tells us that if we condition on any run r with fewer than $\frac{1}{12}\gamma(\frac{n}{4})$ queries, then the probability 0^n is a hit is less than $\frac{1}{12}$. This means that the probability the algorithm correctly classifies 0^n, conditioned on any particular run r with fewer than $\frac{1}{12}\gamma(\frac{n}{4})$ queries, is at most $\frac{7}{12}$. Therefore, for n large enough that $T(n)$ is less than $\frac{1}{12}\gamma(n)$, the probability M^O correctly classifies 0^n in time $T(n)$ is at most $\frac{7}{12}$. So, for sufficiently large n, when we choose O, then with probability at least $\frac{1}{8}$ M^O either fails to run in time $T(n)$ or has success probability less than $\frac{2}{3}$ on input 0^n. □

Appendix A. A QTM is well formed iff its time evolution is unitary. First, we note that the time evolution operator of a QTM always exists. Note that this is true even if the QTM is not well formed.

LEMMA A.1. *If M is a QTM with time evolution operator U, then U has an adjoint operator U^* in the inner-product space of superpositions of the machine M.*

Proof. Let M be a QTM with time evolution operator U. Then the adjoint of U is the operator U' and is the operator whose matrix element in any pair of dimensions i, j is the complex conjugate of the matrix element of U in dimensions j, i. The operator U' defined in this fashion still maps all superpositions to other superpositions (= finite linear combinations of configurations), since any particular configuration of M can be reached with nonzero weight from only a finite number of other configurations. It is also easy to see that for any superpositions ϕ, ψ,

$$\langle U'\phi|\psi\rangle = \langle\phi|U\psi\rangle$$

as desired. □

We include the proofs of the following standard facts for completeness. We will use them in the proof of the theorem below.

FACT A.2. *If U is a linear operator on an inner-product space V and U^* exists, then U preserves norm iff $U^*U = I$.*

Proof. For any $x \in V$, the square of the norm of Ux is $(Ux, Ux) = (x, U^*Ux)$. It clearly follows that if $U^*U = I$, then U preserves norm. For the converse, let $B = U^*U - I$. Since U preserves norm, for every $x \in V$, $(x, U^*Ux) = (x, x)$. Therefore, for every $x \in V$, $(x, Bx) = 0$. It follows that $B = 0$ and therefore $U^*U = I$. □

This further implies the following useful fact.

FACT A.3. *Suppose U is an linear operator in an inner-product space V and U^* exists. Then*

$$\forall x \in V \quad \|Ux\| = \|x\| \qquad \leftrightarrow \qquad \forall x, y \in V \quad (Ux, Uy) = (x, y).$$

Proof. Since $\|Ux\| = \|x\| \leftrightarrow (Ux, Ux) = (x, x)$, one direction follows by substituting $x = y$. For the other direction, if U preserves norm then by Fact A.2, $U^*U = I$. Therefore, $(Ux, Uy) = (x, U^*Uy) = (x, y)$. □

We need to establish one additional fact about norm-preserving operators before we can prove our theorem.

FACT A.4. *Let V be a countable inner-product space, and let $\{|i\rangle\}_{i\in I}$ be an orthonormal basis for V. If U is a norm-preserving linear operator on V and U^* exists, then $\forall i \in I$ $\|U^*|i\rangle\| \leq 1$. Moreover, if $\forall i \in I$ $\|U^*|i\rangle\| = 1$ then U is unitary.*

Proof. Since U preserves norm, $\|UU^*|i\rangle\| = \|U^*|i\rangle\|$. But the projection of $UU^*|i\rangle$ on $|i\rangle$ has norm $|\langle i|UU^*|i\rangle| = \|U^*|i\rangle\|^2$. Therefore $\|U^*|i\rangle\| \geq \|U^*|i\rangle\|^2$, and therefore $\|U^*|i\rangle\| \leq 1$.

Moreover, if $\|U^*|i\rangle\| = 1$, then, since U is norm preserving, $\|UU^*|i\rangle\| = 1$. On the other hand, the projection of $UU^*|i\rangle$ on $|i\rangle$ has norm $\|U^*|i\rangle\|^2 = 1$. It follows that for $j \neq i$, the projection of $UU^*|i\rangle$ on $|j\rangle$ must have norm 0. Thus $|\langle j|UU^*|i\rangle| = 0$. It follows that $UU^* = I$. □

If V is finite dimensional, then $U^*U = I$ implies $UU^* = I$, and therefore an operator is norm preserving if and only if it is unitary. However, if $\mathcal{H}$ is infinite dimensional, then $U^*U = I$ does not imply $UU^* = I$.[6] Nevertheless, the time evolution operators of QTMs have a special structure, and in fact these two conditions are equivalent for the time evolution operator of a QTM.

THEOREM A.5. *A QTM is well formed iff its time evolution operator is unitary.*

Proof. Let U be the norm preserving time evolution operator of a well-formed QTM $M = (\Sigma, Q, \delta)$. Consider the standard orthonormal basis for the superpositions

[6] Consider, for example, the space of finite complex linear combinations of positive integers and the linear operator which maps $|i\rangle$ to $|i + 1\rangle$.

of M given by the set of vectors $|c\rangle$, where c ranges over all configurations of M (as always, $|c\rangle$ is the superposition with amplitude 1 for configuration c and 0 elsewhere). We may express the action of U with respect to this standard basis by a countable dimensional matrix whose c, c'th entry $u_{c,c'} = \langle c'|U|c\rangle$. This matrix has some special properties. First, each row and column of the matrix has only a finite number of nonzero entries. Second, there are only finitely many different types of rows, where two rows are of the same type if their entries are just permutations of each other. We shall show that each row of the matrix has norm 1, and therefore by Fact A.4 above U is unitary. To do so we will identify a set of n columns of the matrix (for arbitrarily large n) and restrict attention to the finite matrix consisting of all the chosen rows and all columns with nonzero entries in these rows. Let this matrix be the $m \times n$ matrix B. By construction, B satisfies two properties: (1) it is almost square; $\frac{m}{n} \leq 1 + \epsilon$ for arbitrarily small ϵ. (2) There is a constant a such that each distinct row type of the infinite matrix occurs at least $\frac{m}{a}$ times among the rows of B.

Now the sum of the squared norms of the rows of B is equal to the sum of the squared norms of the columns. The latter quantity is just n (since the columns of the infinite matrix are orthonormal by Fact A.2 above). If we assume that some row of the infinite matrix has norm $1 - \delta$ for $\delta > 0$, then we can choose n sufficiently large and ϵ sufficiently small so that the sum of the squared norms of the rows is at most $m(1 - \frac{1}{a}) + m/a(1 - \delta) \leq m - m\delta/a \leq n + n\epsilon - n\delta/a < n$. This gives the required contradiction, and therefore all rows of the infinite matrix have norm 1 and, therefore, by Fact A.4 U is unitary.

To construct the finite matrix B let $k > 2$ and fix some contiguous set of k cells on the tape. Consider the set of all configurations S such that the tape head is located within these k cells and such that the tape is blank outside of these k cells. It is easy to see that the number of such configurations $n = k \operatorname{card}(\Sigma)^k \operatorname{card}(Q)$. The columns indexed by configurations in S are used to define the finite matrix B referred to above. The nonzero entries in these columns are restricted to rows indexed by configurations in S together with rows indexed by configurations where the tape head is in the cell immediately to the left or right of the k special cells, and such that the tape is blank outside of these $k + 1$ cells. The number of these additional configurations over and above S is at most $2 \operatorname{card}(\Sigma)^{k+1} \operatorname{card}(Q)$. Therefore, $m = n(1 + 2/k\operatorname{card}(\Sigma))$. For any $\epsilon > 0$, we can choose k large enough such that $m \leq n(1 + \epsilon)$.

Recall that a row of the infinite matrix (corresponding to the operator U) is indexed by configurations. We say that a configuration c is of type $q, \sigma_1, \sigma_2, \sigma_3$ if c is in state $q \in Q$ and the three adjacent tape cells centered about the tape head contain the three symbols $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$. The entries of a row indexed by c must be a permutation of the entries of a row indexed by any configuration c' of the same type as c. This is because a transition of the QTM M depends only on the state of M and the symbol under the tape head and since the tape head moves to an adjacent cell during a transition. Moreover, any configuration d that yields configuration c with nonzero amplitude as a result of a single step must have tape contents identical to those of c in all but these three cells. It follows that there are only a finite number of such configurations d, and the row indexed by c can have only a finite number of nonzero entries. A similar argument shows that each column has only a finite number of nonzero entries.

Now for given $q, \sigma_1, \sigma_2, \sigma_3$ consider the rows of the finite matrix B indexed by configurations of type $q, \sigma_1, \sigma_2, \sigma_3$ and such that the tape head is located at one of the $k - 2$ nonborder cells. Then $|T| = (k - 2)\operatorname{card}(\Sigma)^{k-3}$. Each row of B indexed

by a configuration $c \in T$ has the property that if d is any configuration that yields c with nonzero amplitude in a single step, then $d \in S$. Therefore, the row indexed by c in the finite matrix B has the same nonzero entries as the row indexed by c in the infinite matrix. Therefore, it makes sense to say that row c of B is of the same type as row c of the infinite matrix. Finally, the rows of each type constitute a fraction at least $|T|/m$ of all rows of B. Substituting the bounds from above, we get that this fraction is at least $\frac{(k-2)\mathrm{card}(\Sigma)^{k-3}}{k\ \mathrm{card}(\Sigma)^k\ \mathrm{card}(Q)(1+2/k\mathrm{card}(\Sigma))}$. Since $k \geq 4$, this is at least $\frac{1}{a}$ for constant $a = 2\mathrm{card}(\Sigma)^3\mathrm{card}(Q)(1 + 1/2\mathrm{card}(\Sigma))$. This establishes all the properties of the matrix B used in the proof above. □

Appendix B. Reversible TMs are as powerful as deterministic TM. In this appendix, we prove the synchronization theorem of section 4.1.

We begin with a few simple facts about reversible TMs. We give necessary and sufficient conditions for a deterministic TM to be reversible, and we show that, just as for QTMs, a partially defined reversible TM can always be completed to give a well-formed reversible TM. We also give, as an aside, an easy proof that reversible TMs can efficiently simulate reversible, generalized TMs.

THEOREM B.1. *A TM or generalized TM M is reversible iff both of the following two conditions hold.*

1. *Each state of M can be entered while moving in only one direction. In other words, if $\delta(p_1, \sigma_1) = (\tau_1, q, d_1)$ and $\delta(p_2, \sigma_2) = (\tau_2, q, d_2)$ then $d_1 = d_2$.*
2. *The transition function δ is one-to-one when direction is ignored.*

Proof. First we show that these two conditions imply reversibility.

Suppose $M = (\Sigma, Q, \delta)$ is a TM or generalized TM satisfying these two conditions. Then, the following procedure lets us take any configuration of M and compute its predecessor if it has one. First, since each state can be entered while moving in only one direction, the state of the configuration tells us in which cell the tape head must have been in the previous configuration. Looking at this cell, we can see what tape symbol was written in the last step. Then, since δ is one-to-one we know the update rule, if any, that was used on the previous step, allowing us to reconstruct the previous configuration.

Next, we show that the first property is necessary for reversibility. So, for example, consider a TM or generalized TM $M = (\Sigma, Q, \delta)$ such that $\delta(p_1, \sigma_1) = (\tau_1, q, L)$ and $\delta(p_2, \sigma_2) = (\tau_2, q, R)$. Then, we can easily construct two configurations which lead to the same next configuration: let c_1 be any configuration where the machine is in state p_1 reading a σ_1 and where the symbol two cells to the left of the tape head is a τ_2, and let c_2 be identical to c_1 except that the σ_1 and τ_2 are changed to τ_1 and σ_2, the machine is in state p_2, and the tape head is two cells further left. Therefore, M is not reversible. Since similar arguments apply for each pair of distinct directions, the first condition in the theorem must be necessary for reversibility.

Finally, we show that the second condition is also necessary for reversibility. Suppose that $M = (\Sigma, Q, \delta)$ is a TM or generalized TM with $\delta(p_1, \sigma_1) = \delta(p_2, \sigma_2)$. Then, any pair of configurations which differ only in the state and symbol under the tape head, where one has (p_1, σ_1) and the other (p_2, σ_2), lead to the same next configuration, and again M is not reversible. □

COROLLARY B.2. *If M is a reversible TM, then every configuration of M has exactly one predecessor.*

Proof. Let $M = (\Sigma, Q, \delta)$ be a reversible TM. By the definition of reversibility, each configuration of M has at most one predecessor.

So, let c be a configuration of M in state q. Theorem B.1 tells us that M can enter state q while moving its tape head in only one direction d_q. Since Theorem B.1 tells us that, ignoring direction, δ is one-to-one, taking the inverse of δ on the state q and the symbol in direction $\bar{d}_q$ tells us how to transform c into its predecessor. □

COROLLARY B.3. *If δ is a partial function from $Q \times \Sigma$ to $\Sigma \times Q \times \{L, R\}$ satisfying the two conditions of Theorem* B.1*, then δ can be extended to a total function that still satisfies Theorem* B.1*.*

Proof. Suppose δ is a partial function from $Q \times \Sigma$ to $\Sigma \times Q \times \{L, R\}$ that satisfies the properties of Theorem B.1. Then, for each $q \in Q$ let d_q be the one direction, if any, in which q can be entered, and let d_q be (arbitrarily) L otherwise. Then we can fill in undefined values of δ with as yet unused triples of the form (τ, q, d_q) so as to maintain the conditions of Theorem B.1. Since the number of such triples is $\text{card}(\Sigma)\ \text{card}(Q)$, there will be exactly enough to fully define δ. □

THEOREM B.4. *If M is a generalized reversible TM, then there is a reversible TM M' that simulates M with slowdown at most* 2*.*

Proof. The idea is to replace any transition that has the tape head stand still with two transitions. The first updates the tape and moves to the right, remembering which state it should enter. The second steps back to the left and enters the desired state.

So, if $M = (\Sigma, Q, \delta)$ is a generalized reversible TM then we let M' be identical to M except that for each state q with a transition of the form $\delta(p, \sigma) = (\tau, q, N)$ we add a new state q' and we also add a new transition rule $\delta(q', \sigma) = \sigma, q, L$ for each $\sigma \in \Sigma$. Finally, we replace each transition $\delta(p, \sigma) = (\tau, q, N)$ with the transition $\delta(p, \sigma) = (\tau, q', R)$. Clearly M' simulates M with slowdown by a factor of at most 2.

To complete the proof, we need to show that M' is also reversible.

So, consider a configuration c of M'. We need to show that c has at most one predecessor. If c is in state $q \in Q$ and M enters q moving left or right, then the transitions into q in M' are identical to those in M and therefore since M is reversible, c has at most one predecessor in M'. Similarly, if c is in one of the new states q' then the transitions into q' in M' are exactly the same as those into q in M, except that the tape head moves right instead of staying still. So, again the reversibility of M implies that c has at most one predecessor. Finally, suppose c is in state $q \in Q$, where M enters q while standing still. Then, Theorem B.1 tells us that all transitions in M that enter q have direction N. Therefore, all of them have been removed, and the only transitions entering q in M' are the new ones of the form $\delta(q', \sigma) = \sigma, q, L$. Again, this means that c can have only one predecessor. □

We will prove the synchronization theorem in the following way. Using ideas from the constructions of Bennett [7] and Morita, Shirasaki, and Gono [33], we will show that given any deterministic TM M there is a reversible multitrack TM which on input x produces output $x; M(x)$, and whose running time depends only on the sequence of head movements of M on input x. Then, since any deterministic computation can be simulated efficiently by an "oblivious" machine whose head movements depend only on the length of its input, we will be able to construct the desired "synchronized" reversible TM.

The idea of Bennett's simulation is to run the target machine, keeping a history to maintain reversibility. Then the output can be copied and the simulation run backward so that the history is exactly erased while the input is recovered. Since the target tape head moves back and forth, while the history steadily grows, Bennett uses a multitape TM for the simulation.

The idea of Morita, Shirasaki, and Gono's simulation is to use a simulation tape with several tracks, some of which are used to simulate the tape of the desired machine and some of which are used to keep the history. Provided that the machine can move reversibly between the current head position of the target machine and the end of the history, it can carry out Bennett's simulation with a quadratic slowdown. Morita, Shirasaki, and Gono work with TMs with one-way infinite tapes so that the simulating machine can move between the simulation and the history by moving left to the end of the tape and then searching back toward the right. In our simulation, we write down history for every step the target machine takes, rather than just the nonreversible steps. This means that the end of the history will always be further right than the simulation, allowing us to work with two-way infinite tapes. Also, we use reversible TMs that can only move their head in the directions $\{L, R\}$ rather than the generalized reversible TMs used by Bennett, Morita, Shirasaki, and Gono.

DEFINITION B.5. *A deterministic TM is* oblivious *if its running time and the position of its tape head at each time step depend only on the length of its input.*

In carrying out our single tape Bennett constructions we will find it useful to first build simple reversible TMs to copy a string from one track to another and to exchange the strings on two tracks. We will copy strings delimited by blanks in the single tape Bennett construction but will copy strings with other delimiters in a later section.

LEMMA B.6. *For any alphabet Σ, there is a normal form, reversible TM M with alphabet $\Sigma \times \Sigma$ with the following property. When run on input $x; y$ M runs for $2\max(|x|, |y|) + 4$ steps, returns the tape head to the start cell, and outputs $y; x$.*

Proof. We let M have alphabet $\Sigma \times \Sigma$, state set $\{q_0, q_1, q_2, q_3, q_f\}$, and transition function defined by

	$(\#, \#)$	other (σ_1, σ_2)
q_0	$(\#, \#), q_1, L$	$(\sigma_1, \sigma_2), q_1, L$
q_1	$(\#, \#), q_2, R$	
q_2	$(\#, \#), q_3, L$	$(\sigma_2, \sigma_1), q_2, R$
q_3	$(\#, \#), q_f, R$	$(\sigma_1, \sigma_2), q_3, L$
q_f	$(\#, \#), q_0, R$	$(\sigma_1, \sigma_2), q_0, R$

Since each state in M can be entered in only one direction and its transition function is one-to-one, M is reversible. Also, it can be verified that M performs the desired computation in the stated number of steps. □

LEMMA B.7. *For any alphabet Σ, there is a normal form, reversible TM M with alphabet $\Sigma \times \Sigma$ with the following property. When run on input x, M outputs $x; x$, and when run on input $x; x$ it outputs x. In either case, M runs for $2|x| + 4$ steps and leaves the tape head back in the start cell.*

Proof. We let M have alphabet $\Sigma \times \Sigma$, state set $\{q_0, q_1, q_2, q_3, q_f\}$, and transition function defined by the following where each transition is duplicated for each nonblank $\sigma \in \Sigma$:

	$(\#, \#)$	$(\sigma, \#)$	(σ, σ)
q_0	$(\#, \#), q_1, L$	$(\sigma, \#), q_1, L$	$(\sigma, \sigma), q_1, L$
q_1	$(\#, \#), q_2, R$		
q_2	$(\#, \#), q_3, L$	$(\sigma, \sigma), q_2, R$	$(\sigma, \#), q_2, R$
q_3	$(\#, \#), q_f, R$	$(\sigma, \#), q_3, L$	$(\sigma, \sigma), q_3, L$
q_f	$(\#, \#), q_0, R$	$(\sigma, \#), q_0, R$	$(\sigma, \sigma), q_0, R$

Since each state in M can be entered in only one direction and its transition function is one-to-one, M can be extended to a reversible TM. Also, it can be verified that M performs the desired computation in the stated number of steps. □

THEOREM B.8. *Let M be an oblivious deterministic TM which on any input x produces output $M(x)$, with no embedded blanks, with its tape head back in the start cell, and with $M(x)$ beginning in the start cell. Then there is a reversible TM M' which on input x produces output $x; M(x)$ and on input $x; M(x)$ produces output x. In both cases, M' halts with its tape head back in the start cell and takes time which depends only on the lengths of x and $M(x)$ and which is bounded by a quadratic polynomial in the running time of M on input x.*

Proof. Let $M = (\Sigma, Q, \delta)$ be an oblivious deterministic TM as stated in the theorem and let q_0, q_f be the initial and final states of M.

The simulation will run in three stages.

1. M' will simulate M maintaining a history to make the simulation reversible.
2. M' will copy its first track, as shown in Lemma B.7.
3. M' runs the reverse of the simulation of M erasing the history while restoring the input.

We will construct a normal form reversible TM for each of the three stages and then dovetail them together.

Each of our machines will be a four-track TM with the same alphabet. The first track, with alphabet $\Sigma_1 = \Sigma$, will be used to simulate the tape of M. The second track, with alphabet $\Sigma_2 = \{\#, 1\}$, will be used to store a single 1 locating the tape head of M. The third track, with alphabet $\Sigma_3 = \{\#, \$\} \cup (Q \times \Sigma)$, will be used to write down a list of the transitions taken by M, starting with the marker \$. This \$ will help us find the start cell when we enter and leave the copying phase. The fourth track, with alphabet $\Sigma_4 = \Sigma$, will be used to write the output of M.

In describing the first machine, we will give a partial list of transitions obeying the conditions of Theorem B.1 and then appeal to Corollary B.3. Our machines will usually only be reading and writing one track at a time. So, for convenience we will list a transition with one or more occurrences of the symbols of the form v_i and v_i' to stand for all possible transitions with v_i replaced by a symbol from the appropriate Σ_i other than the special marker \$, and with v_i' replaced by a symbol from Σ_i other than \$ and #.

The first phase of the simulation will be handled by the machine M_1 with state set given by the union of sets Q, $Q \times Q \times \Sigma \times [1,4]$, $Q \times [5,7]$, and $\{q_a, q_b\}$. Its start state will be q_a and the final state q_f.

The transitions of M_1 are defined as follows. First, we mark the position of the tape head of M in the start cell, mark the start of the history in cell 1, and enter state q_0.

$$\begin{array}{llcll} q_a, & (v_1, \#, \#, v_4) & \rightarrow & (v_1, 1, \#, v_4), & q_b, \quad R, \\ q_b, & (v_1, \#, \#, v_4) & \rightarrow & (v_1, \#, \$, v_4), & q_0, \quad R. \end{array}$$

Then, for each pair p, σ with $p \neq q_f$ and with transition $\delta(p, \sigma) = (\tau, q, d)$ in M we include transitions to go from state p to state q updating the simulated tape of M while adding (p, σ) to the end of the history. We first carry out the update of the simulated tape, remembering the transition taken, and then move to the end of the history to deposit the information on the transition. If we are in the middle of the history, we reach the end of the history by walking right until we hit a blank. However, if we are to the left of the history, then we must first walk right over blanks until we reach the start of the history.

$$
\begin{array}{cccccc}
p, & (\sigma,1,v_3,v_4) & \to & (\tau,\#,v_3,v_4), & (q,p,\sigma,1), & d \\
(q,p,\sigma,1), & (v_1,\#,\$,v_4) & \to & (v_1,1,\$,v_4), & (q,p,\sigma,3), & R \\
(q,p,\sigma,1), & (v_1,\#,v_3',v_4) & \to & (v_1,1,v_3',v_4), & (q,p,\sigma,3), & R \\
(q,p,\sigma,1), & (v_1,\#,\#,v_4) & \to & (v_1,1,\#,v_4), & (q,p,\sigma,2), & R \\
(q,p,\sigma,2), & (v_1,\#,\#,v_4) & \to & (v_1,\#,\#,v_4), & (q,p,\sigma,2), & R \\
(q,p,\sigma,2), & (v_1,\#,\$,v_4) & \to & (v_1,\#,\$,v_4), & (q,p,\sigma,3), & R \\
(q,p,\sigma,3), & (v_1,\#,v_3',v_4) & \to & (v_1,\#,v_3',v_4), & (q,p,\sigma,3), & R \\
(q,p,\sigma,3), & (v_1,\#,\#,v_4) & \to & (v_1,\#,(p,\sigma),v_4), & (q,4), & R
\end{array}
$$

When the machine reaches state $(q,4)$ it is standing on the first blank after the end of the history. So, for each state $q \in Q$, we include transitions to move from the end of the history back left to the head position of M. We enter our walk-left state $(q,5)$ while writing a # on the history tape. So, to maintain reversibility, we must enter a second left-walking state $(q,6)$ to look for the tape head marker past the left end of the history. When we reach the tape head position, we step left and right entering state q.

$$
\begin{array}{cccccc}
(q,4), & (v_1,\#,\#,v_4) & \to & (v_1,\#,\#,v_4), & (q,5), & L \\
(q,5), & (v_1,\#,v_3',v_4) & \to & (v_1,\#,v_3',v_4), & (q,5), & L \\
(q,5), & (v_1,\#,\$,v_4) & \to & (v_1,\#,\$,v_4), & (q,6), & L \\
(q,5), & (v_1,1,v_3',v_4) & \to & (v_1,1,v_3',v_4), & (q,7), & L \\
(q,5), & (v_1,1,\$,v_4) & \to & (v_1,1,\$,v_4), & (q,7), & L \\
(q,6), & (v_1,\#,\#,v_4) & \to & (v_1,\#,\#,v_4), & (q,6), & L \\
(q,6), & (v_1,1,\#,v_4) & \to & (v_1,1,\#,v_4), & (q,7), & L \\
(q,7), & (v_1,v_2,v_3,v_4) & \to & (v_1,v_2,v_3,v_4), & q, & R
\end{array}
$$

Finally, we put M_1 in normal form by including the transition

$$q_f, \quad \sigma \quad \to \quad \sigma, \quad q_a, \quad R$$

for each σ in the simulation alphabet.

It can be verified that each state can be entered in only one direction using the above transitions, and relying on the fact that M is in normal form, and we don't simulate its transitions from q_f back to q_0, it can also be verified that the partial transition function described is one-to-one. Therefore, Corollary B.3 says that we can extend these transitions to give a reversible TM M_1. Notice also that the operation of M_1 is independent of the contents of the fourth track and that it leaves these contents unaltered.

For the second and third machines, we simply use the copying machine constructed in Lemma B.7 above, and the reverse of M_1 constructed using Lemma 4.12 on page 1431. Since M_1 operated independently of the fourth track, so will its reversal. Therefore, dovetailing these three TMs gives a reversible TM M', which on input x produces output $x;\epsilon;\epsilon;M(x)$ with its tape head back in the start cell, and on input $x;\epsilon;\epsilon;M(x)$ produces output x.

Notice that the time required by M_1 to simulate a step of M is bounded by a polynomial in the running time of M and depends only on the current head position of M, the direction M moves in the step, and how many steps have already been carried out. Therefore, the running time of the first phase of M' depends only on the series of head movements of M. In fact, since M is oblivious, this running time

depends only the length of x. The same is true of the third phase of M', since the reversal of M_1 takes exactly two extra time steps. Finally, the running time of the copying machine depends only on the length of $M(x)$. Therefore, the running time of M' on input x depends only on the lengths of x and $M(x)$ and is bounded by a quadratic polynomial in the running time of M on x. □

THEOREM B.9. *Let M_1 be an oblivious deterministic TM which on any input x produces output $M_1(x)$, with no embedded blanks, with its tape head back in the start cell, with $M_1(x)$ beginning in the start cell, and such that the length of $M_1(x)$ depends only on the length of x. Let M_2 be an oblivious deterministic TM with the same alphabet as M_1 which on any input $M_1(x)$ produces output x, with its tape head back in the start cell, and with x beginning in the start cell. Then there is a reversible TM M' which on input x produces output $M_1(x)$. Moreover, M' on input x halts with its tape head back in the start cell, and takes time which depends only on the length of x and which is bounded by a polynomial in the running time of M_1 on input x and the running time of M_2 on $M_1(x)$.*

Proof. Let M_1 and M_2 be as stated in the theorem with the common alphabet Σ.

Then the idea to construct the desired M' is first to run M_1 to compute $x; M(x)$, then to run an exchange routine to produce $M(x); x$ and finally to run M_2 to erase the string x, where each of the three phases starts and ends with the tape head in the start cell. Using the construction in Theorem B.8, we build normal form, reversible TMs to accomplish the first and third phases in times which depend only on the lengths of x and $M_1(x)$ and are bounded by a polynomial in the running times of M_1 on input x and M_2 on input $M_1(x)$. In Lemma B.6 on page 1467 we have already constructed a reversible TM that performs the exchange in time depending only on the lengths of the two strings. Dovetailing these three machines gives the desired M. □

Since any function computable in deterministic polynomial time can be computed in polynomial time by an oblivious generalized deterministic TM, Theorems B.8 and B.9 together with Theorem 4.2 give us the following.

THEOREM 4.3 (synchronization theorem). *If f is a function mapping strings to strings which can be computed in deterministic polynomial time and such that the length of $f(x)$ depends only on the length of x, then there is a polynomial time, stationary, normal form QTM which given input x, produces output $x; f(x)$, and whose running time depends only on the length of x.*

If f is a function from strings to strings such that both f and f^{-1} can be computed in deterministic polynomial time and such that the length of $f(x)$ depends only on the length of x, then there is a polynomial time, stationary, normal form QTM, which, given input x, produces output $f(x)$, and whose running time depends only on the length of x.

Appendix C. A reversible looping TM. We prove here the looping lemma from section 4.2.

LEMMA 4.13 (looping lemma). *There is a stationary, normal form, reversible TM M and a constant c with the following properties. On input any positive integer k written in binary, M runs for time $O(k \log^c k)$ and halts with its tape unchanged. Moreover, M has a special state q^* such that on input k, M visits state q^* exactly k times, each time with its tape head back in the start cell.*

Proof. As mentioned above in section 4.2, the difficulty is to construct a loop with a reversible entrance and exit. We accomplish this as follows. Using the synchronization theorem, we can build three-track stationary, normal form, reversible TM $M_1 = (\Sigma, Q, \delta)$ running in time polynomial in $\log k$ that on input $b; x; k$ where

$b \in \{0,1\}$ outputs $b'; x+1; k$ where b' is the opposite of b if $x = 0$ or $k-1$ (but not both) and $b' = b$ otherwise. Calling the initial and final states of this machine q_0, q_f, we construct a reversible M_2 that loops on machine M_1 as follows. We will give M_2 new initial and final states q_a, q_z and ensure that it has the following three properties.

1. Started in state q_a with a 0 on the first track, M_2 steps left and back right, changing the 0 to a 1, and entering state q_0.
2. When in state q_f with a 0 on the first track, M_2 steps left and back right into state q_0.
3. When in state q_f with a 1 on the first track, M_2 steps left and back right, changing the 1 to a 0, and halts.

So, on input $0;0;k$ M_2 will step left and right into state q_0, changing the tape contents to $1;0;k$. Then machine M_1 will run for the first time changing $1;0;k$ to $0;1;k$ and halting in state q_f. Whenever M_2 is in state q_f with a 0 on the first track, it reenters q_0 to run M_2 again. So, machine M_2 will run $k-1$ more times until it finally produces $1;k;k$. At that point, M_2 changes the tape contents to $0;k;k$ and halts. So on input $0;0;k$, M_2 visits state q_f exactly k times, each time with its tape head back in the start cell. This means we can construct the desired M by identifying q_f as q^* and dovetailing M_2 before and after with reversible TMs, constructed using the synchronization theorem, to transform k to $0;0;k$ and $0;k;k$ back to k.

We complete the proof by constructing a normal form, reversible M_2 that satisfies the three properties above. We give M_2 the same alphabet as M_1 and additional states q_a, q_b, q_y, q_z. The transition function for M_2 is the same as that of M_1 for states in $Q - q_f$ and otherwise depends only on the first track (leaving the others unchanged) and is given by the following table

	#	0	1
q_a		$(1, q_b, L)$	
q_b	$(\#, q_0, R)$		
q_f		$(0, q_b, L)$	$(0, q_y, L)$
q_y	$(\#, q_z, R)$		
q_z	$(\#, q_a, R)$	$(0, q_a, R)$	$(1, q_a, R)$

It is easy to see that M_2 is normal form and satisfies the three properties stated above. Moreover, since M_1 is reversible and obeys the two conditions of Theorem B.1, it can be verified that the transition function of M_2 also obeys the two conditions of Theorem B.1. Therefore, according to Theorem B.3, the transition function of M_2 can be completed giving a reversible TM. □

Acknowledgements. This paper has greatly benefited from the careful reading and many useful comments of Richard Jozsa and Bob Solovay. We wish to thank them as well as Gilles Brassard, Charles Bennett, Noam Nisan, and Dan Simon.

REFERENCES

[1] L. Adleman, J. DeMarrais, and M. Huang, *Quantum computability*, SIAM J. Comput., 26 (1997), pp. 1524–1540.

[2] S. Arora, R. Impagliazzo, and U. Vazirani, *On the Role of the Cook-Levin Theorem in Complexity Theory*, manuscript, 1993.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and intractability of approximation problems*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1992, pp. 14–23.

[4] L. Babai and S. Moran, *Arthur–Merlin games: A randomized proof system, and a hierarchy of complexity classes*, J. Comput. System Sci., 36 (1988), pp. 254–276.

[5] A. Barenco, C. Bennett, R. Cleve, D. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter, *Elementary gates for quantum computation*, Phys. Rev. A, 52 (1995), pp. 3457–3467.

[6] P. Benioff, *Quantum Hamiltonian models of Turing machines*, J. Statist. Phys., 29 (1982), pp. 515–546.

[7] C. H. Bennett, *Logical reversibility of computation*, IBM J. Res. Develop., 17 (1973), pp. 525–532.

[8] C. H. Bennett, *Time/space tradeoffs for reversible computation*, SIAM J. Comput., 18 (1989), pp. 766–776.

[9] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani, *Strengths and weaknesses of quantum computing*, SIAM J. Comput., 26 (1997), pp. 1510–1523.

[10] E. Bernstein, *Quantum Complexity Theory*, Ph.D. dissertation, Univ. of California, Berkeley, May, 1997.

[11] E. Bernstein and U. Vazirani, *Quantum complexity theory*, in Proc. 25th Annual ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 11–20.

[12] A. Berthiaume and G. Brassard, *The quantum challenge to structural complexity theory*, in Proc. 7th IEEE Conference on Structure in Complexity Theory, 1992, pp. 132–137.

[13] A. Berthiaume and G. Brassard, *Oracle quantum computing*, J. Modern Optics, 41 (1994), pp. 2521–2535.

[14] A. Berthiaume, D. Deutsch, and R. Jozsa, *The stabilisation of quantum computation*, in Proc. Workshop on Physics and Computation, Dallas, TX, IEEE Computer Society Press, Los Alamitos, CA, 1994, p. 60.

[15] N. Bshouty and J. Jackson, *Learning DNF over uniform distribution using a quantum example oracle*, in Proc. 8th Annual ACM Conference on Computational Learning Theory, ACM, New York, 1995, pp. 118–127.

[16] A. R. Calderbank and P. Shor, *Good quantum error correcting codes exist*, Phys. Rev. A, 54 (1996), pp. 1098–1106.

[17] Cirac and Zoller, *Quantum computation using trapped cold ions*, Phys. Rev. Lett., 74 (1995), pp. 4091–4094.

[18] I. Chuang, R. LaFlamme, P. Shor, and W. Zurek, *Quantum computers, factoring and decoherence*, Science, Dec. 8, 1995, pp. 1633–1635.

[19] C. Cohen-Tannoudji, B. Diu, and F. LaLoe, *Quantum Mechanics*, Longman Scientific & Technical, Essex, 1977, pp. 108–181.

[20] D. Deutsch, *Quantum theory, the Church–Turing principle and the universal quantum computer*, in Proc. Roy. Soc. London Ser. A, 400 (1985), pp. 97–117.

[21] D. Deutsch, *Quantum computational networks*, in Proc. Roy. Soc. London Ser. A, 425 (1989), pp. 73–90.

[22] D. Deutsch and R. Jozsa, *Rapid solution of problems by quantum computation*, in Proc. Roy. Soc. London Ser. A, 439 (1992), pp. 553–558.

[23] D. DiVincenzo, *Two-bit gates are universal for quantum computation*, Phys. Rev. A, 51 (1995), pp. 1015–1022.

[24] C. Dürr, M. Santha, and Thanh, *A decision procedure for unitary linear quantum cellular automata*, in Proc. 37th IEEE Symposium on the Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1996, pp. 38–45.

[25] R. Feynman, *Simulating physics with computers*, Internat. J. Theoret. Phys., 21 (1982), pp. 467–488.

[26] R. Feynman, *Quantum mechanical computers*, Found. Phys., 16 (1986), pp. 507–531; Optics News, February 1985.

[27] E. Fredkin and T. Toffoli, *Conservative Logic*, Internat. J. Theoret. Phys., 21 (1982), p. 219.

[28] L. Grover, *A fast quantum mechanical algorithm for database search*, in Proc. 28th Annual ACM Symposium on Theory of Computing, ACM, New York, 1996, pp. 212–219.

[29] R. Landauer, *Is quantum mechanics useful?*, Phil. Trans. R. Soc. A, 353 (1995), pp. 367–376.
[30] R. Lipton, Personal communication, 1994.
[31] S. Lloyd, *A potentially realizable quantum computer*, Science, 261 (1993), pp. 1569–1571.
[32] J. Machta, *Phase Information in Quantum Oracle Computing*, Physics Department, University of Massachusetts, Amherst, MA, manuscript, 1996.
[33] K. Morita, A. Shirasaki, and Y. Gono, *A 1-tape 2-symbol reversible Turing machine*, IEEE Trans. IEICE, E72 (1989), pp. 223–228.
[34] J. von Neumann, *Various Techniques Used in Connection with Random Digits*, Notes by G. E. Forsythe, National Bureau of Standards, Applied Math Series, 12 (1951), pp. 36–38. Reprinted in von Neumann's Collected Works, Vol. 5, A. H. Taub, ed., Pergamon Press, Elmsford, NY, 1963, pp. 768–770.
[35] G. Palma, K. Suominen, and A. Ekert, *Quantum Computers and Dissipation*, Proc. Roy. Soc. London Ser. A, 452 (1996), pp. 567–584.
[36] A. Shamir, *IP=PSPACE*, in Proc. 22nd ACM Symposium on the Theory of Computing, ACM, New York, 1990, pp. 11–15.
[37] P. Shor, *Algorithms for quantum computation: Discrete log and factoring*, in Proc. 35th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1994, pp. 124–134.
[38] P. Shor, *Fault-tolerant quantum computation*, in Proc. 37th Annual IEEE Symposium on the Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1996, pp. 56–65.
[39] D. Simon, *On the power of quantum computation*, in Proc. 35th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1994, pp. 116–123; SIAM J. Comput., 26 (1997), pp. 1474–1483.
[40] R. Solovay and A. Yao, manuscript, 1996.
[41] T. Toffoli, *Bicontinuous extensions of invertible combinatorial functions*, Math. Systems Theory, 14 (1981), pp. 13–23.
[42] W. Unruh, *Maintaining coherence in quantum computers*, Phys. Rev. A, 51 (1995), p. 992.
[43] L. Valiant, Personal communication, 1992.
[44] U. Vazirani and V. Vazirani, *Random polynomial time is equal to semi-random polynomial time*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 417–428.
[45] J. Watrous, *On one dimensional quantum cellular automata*, in Proc. 36th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1995, pp. 528–537.
[46] A. Yao, *Quantum circuit complexity*, in Proc. 34th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1993, pp. 352–361.
[47] D. Zuckerman, *Weak Random Sources*, Ph.D. dissertation, Univ. of California, Berkeley, 1990.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1474–1483, October 1997

008

ON THE POWER OF QUANTUM COMPUTATION*

DANIEL R. SIMON†

Abstract. The quantum model of computation is a model, analogous to the probabilistic Turing machine (PTM), in which the normal laws of chance are replaced by those obeyed by particles on a quantum mechanical scale, rather than the rules familiar to us from the macroscopic world. We present here a problem of distinguishing between two fairly natural classes of functions, which can provably be solved exponentially faster in the quantum model than in the classical probabilistic one, when the function is given as an oracle drawn equiprobably from the uniform distribution on either class. We thus offer compelling evidence that the quantum model may have significantly more complexity theoretic power than the PTM. In fact, drawing on this work, Shor has recently developed remarkable new quantum polynomial-time algorithms for the discrete logarithm and integer factoring problems.

Key words. quantum computation, complexity theory, oracles

AMS subject classifications. 03D15, 68Q10, 81P10

PII. S0097539796298637

1. Introduction.

You have nothing to do but mention the quantum theory, and people will take your voice for the voice of science, and believe anything.

—Bernard Shaw, *Geneva* (1938)

The suggestion that the computational power of quantum mechanical processes might be beyond that of traditional computation models was first raised by Feynman [Fey82]. Benioff [Beni82] had already determined that such processes were at least as powerful as Turing machines (TMs); Feynman asked in turn whether such quantum processes could in general be efficiently simulated on a traditional computer. He also identified some reasons why the task appears difficult and pointed out that a "quantum computer" might be imagined that could perform such simulations efficiently. His ideas were elaborated on by Deutsch [Deu85], who proposed that such machines, using quantum mechanical processes, might be able to perform computations that "classical" computing devices (those that do not exploit quantum mechanical effects) can only perform very inefficiently. To that end, he developed a (theoretically) physically realizable model for the "quantum computer" that he conjectured might be more efficient than a classical TM for certain types of computations.

Since the construction of such a computer is beyond the realm of present technology, and would require overcoming a number of daunting practical barriers, it is worth asking first whether the proposed model even theoretically offers any substantial computational benefits over the classical TM model. The first hint of such a possibility was given by Deutsch and Jozsa [DJ92], who presented a simple "promise problem" that can be solved efficiently without error on Deutsch's quantum computer but that requires exhaustive search to solve deterministically without error in a classical setting. Berthiaume and Brassard [BB92] recast this problem in complexity theoretic

* Received by the editors February 7, 1996; accepted for publication (in revised form) December 2, 1996. A preliminary version of this paper appeared in *Proceedings of the 35th IEEE Symposium on the Foundations of Computer Science (FOCS)*, Santa Fe, NM, Shafi Goldwasser, ed., IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 116–123.

http://www.siam.org/journals/sicomp/26-5/29863.html

† Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399 (dansimon@microsoft.com).

terms, constructing an oracle relative to which the quantum computer is exponentially more efficient than any classical (zero-error) PTM. In [BB93], they exhibited a similar separation for nondeterministic (zero-error) TMs.

Unfortunately, the problems explored in [BB92, BB93] are all efficiently solved by a (classical) PTM with exponentially small error probability. However, Bernstein and Vazirani [BV93] subsequently constructed an oracle which produces a superpolynomial relativized separation between the quantum and (classical) probabilistic models. They also gave the first efficient construction of a universal quantum computer which can simulate any quantum computer (as defined by Deutsch, subject to a slight constraint later removed in [Yao93]) with only polynomial overhead (Deutsch's universal quantum computer was subject to exponential slowdown).

In this paper,[1] we present an expected polynomial-time algorithm for a quantum computer that distinguishes between two reasonably natural classes of polynomial-time computable functions. This task appears computationally difficult in the classical setting; in particular, if the function is supplied as an oracle, then distinguishing (with nonnegligible probability) between a random function from one class and a random member of the other would take exponential time for a classical PTM. (A direct consequence is an oracle which produces an exponential relativized gap between the quantum and classical probabilistic models.) Recently Shor [Sho94], drawing on the general approach presented here and using a number of ingenious new techniques, has constructed quantum polynomial-time algorithms for the discrete logarithm and integer factoring problems.

2. The quantum computation model.

2.1. Classical probability versus the quantum model. We can represent a (classical) probabilistic computation on a TM as a leveled tree, as follows: each node corresponds to a state of the machine (i.e., a configuration), and each level represents a step of the computation. The root corresponds to the machine's starting configuration, and each other node corresponds to a different configuration reachable with nonzero probability, in one computation step, from the configuration represented by its parent node. Each edge, directed from parent to child, is associated with the probability that the computation follows that edge to the child node's configuration once reaching the parent node's configuration. Obviously, configurations may be duplicated across a single level of the tree, as children of different parents, as well as appear on different levels of the tree; nevertheless we represent each such appearance by a separate node. Also, we say that any such computation tree is *well defined*, meaning that the probabilities on the edges emanating from a parent node, and the configurations associated with its children, are strictly a function of the configuration associated with the parent node, regardless of the node's position in the tree.

Of course, this tree must necessarily conform not only to the constraints set by the definition of the TM whose computation it represents but also to the laws of probability. For example, the probability of following a particular path from the root to a node is simply the product of the probabilities along its edges. Hence we can associate a probability with each node, corresponding to the probability that that node is reached in the computation, and equal to the product of the probabilities assigned to the edges in the path leading to it from the root. Moreover, the probability that a particular configuration is reached at a certain step i in the computation is simply the sum of the probabilities of all the nodes corresponding to that configuration at

[1] An earlier version of this paper appears in [Sim94].

level i in the tree. (For example, the probability of a particular final configuration is the sum of the probabilities of all leaf nodes corresponding to that configuration.) Finally, the sum of the probabilities of all the configurations at any level of the tree must always be 1, regardless of the starting configuration. A necessary and sufficient condition for a well-defined computation tree to always satisfy this constraint is that the sum of the probabilities on edges leaving any single node always be 1.

A familiar equivalent representation of our well-defined computation, of course, is the Markov chain, in which a vector of probabilities for each possible configuration at a given step is multiplied by a fixed matrix to obtain the vector of probabilities of each configuration at the next step. For example, a space-$S(n)$-bounded computation can be represented by a Markov process with $2^{O(S(n))}$ states. Such a process can always be translated into a PTM, as long as (a) it never takes one configuration to another with nonzero probability unless the second can be obtained from the first via a single TM operation (i.e., changing the control state, and/or changing the contents of the cell under the tape head, and/or moving the head position by one cell); and (b) it assigns probabilities to new configurations consistently for any set of original configurations in which the control state and the contents of the cell under the tape head are identical. We say that processes with this property are *local*; obviously, the computation of any PTM can be represented as a computation tree which is not only well defined but also local.

A computation on a quantum Turing machine (QTM) (as described in [Deu85]) can be represented by a similar tree, but the laws of quantum mechanics require that we make some adjustments to it. Instead of a probability, each edge is associated with an *amplitude.* (In general, an amplitude is a complex number with magnitude at most 1, but it is shown in [BV93] that it is sufficient for complexity theoretic purposes to consider only real amplitudes in the interval $[-1, 1]$.) As before, the amplitude of a node is simply the product of the amplitudes of the edges on the path from the root to that node. The amplitude of a particular configuration at any step in the computation is simply the sum of the amplitudes of all nodes corresponding to that configuration at the level in the tree corresponding to that step. In the vector–matrix representation corresponding to the classical Markov process, a quantum computation step corresponds to multiplying the vector of amplitudes of all possible configurations at the current step by a fixed matrix to obtain the vector representing the amplitude of each configuration in the next step.

Now, the probability of a configuration at any step is the *square* of its amplitude. For example, the probability of a particular final configuration is the square of the sum (*not* the sum of the squares) of the amplitudes of all leaf nodes corresponding to that configuration. This way of calculating probability has some remarkable consequences; for instance, a particular configuration c could correspond to two leaf nodes with amplitudes α and $-\alpha$, respectively, and the probability of c being the final configuration would therefore be zero. Yet the parent nodes of these two nodes might both have nonzero probability. In fact, the computation would produce c with probability α^2 if only the configuration of *one* of the leaf nodes were in some way different. Similarly, if both leaf nodes had amplitude α, then the probability of c being the final configuration would be, not $2\alpha^2$, but rather $4\alpha^2$—that is, more than twice the probability we would obtain if either of the nodes corresponded to a different configuration. This mutual influence between different branches of the computation is called *interference*, and it is the reason why quantum computation is conjectured to be more powerful, in a complexity theoretic sense, than classical probabilistic computation. (It also means that probability is a rather abstract notion for a nonleaf node, with little intuitive

connection to the ultimate probability of any particular computation result.)

However, even a quantum computation tree must obey the property that the sum of the probabilities of configurations at any level must always equal 1. The choice of amplitudes on the edges leading from a node to its children must therefore be restricted so as to ensure that this condition is always obeyed, regardless of the starting configuration. Now, it turns out that it is *not* sufficient simply to require that for each node the sum of the squares of the amplitudes on edges leading to its children be 1. In fact, even *deterministic* ("classical") computation steps, in which a single outgoing edge to a single child has amplitude 1, can violate this constraint by causing previously different configurations in different branches of the tree to become identical. Such an event might change the pattern of interference, thereby altering the sum of the probabilities of the configurations.

Computation steps which never violate this constraint are called *unitary*, because they are equivalent to multiplying the vector of amplitudes of all possible configurations by a unitary matrix. (Recall that a unitary matrix is one whose inverse is its conjugate transpose; when we restrict ourselves to real amplitudes, such a matrix becomes orthogonal—that is, equal to the inverse of its transpose.) A QTM must always execute unitary steps; for instance, its deterministic steps must be *reversible*, in the sense that the preceding configuration can always be determined given the current one. (This restriction eliminates the aforementioned problem of distinct configurations suddenly becoming identical.) To be unitary, nonclassical steps must also be reversible, in the sense that some unitary (nonclassical) step "undoes" the step. Such "unflipping" of quantum coins is made possible by the counterintuitive effects of interference, which can cause alternative branches to cancel each other out, leaving the remaining ones (possibly all leading to an identical outcome) certain.

The QTM model of computation described here is simply a PTM in which the rules described above replace those of classical probability. (A more formal definition of an essentially equivalent QTM model can be found in [BV93].) Just as the computation tree of a classical probabilistic computation is always well defined and local, with probabilities always summing to 1, the computation tree of a quantum computation is always well defined, local, and unitary. At each step, the amplitudes of possible next configurations are determined by the amplitudes of possible current configurations, according to a fixed, local, unitary transformation representable by a matrix analogous to the stochastic matrix of a Markov process.

It is important to note that the standard equivalent characterization of a classical probabilistic computation tree, in which a deterministic machine simply reads a tape containing prewritten outcomes of independent fair coin tosses, does not appear to have a counterpart in the quantum model. It is true that an efficient universal QTM was shown in [BV93] to require only a fixed, standard set of amplitudes for all its nonclassical steps. However, the reversibility condition guarantees that no new interference will be introduced once those steps have been completed (say, after all the "quantum coins" have been tossed), and any remaining computation will thus be unable to exploit quantum effects. Hence the classical and nonclassical parts of the quantum computation tree cannot be "teased apart," as can the deterministic and probabilistic parts of a classical computation tree, and we must always keep an entire tree in mind when we deal with quantum computation, rather than assuming we can just follow a particular (deterministic) branch after some point. We therefore refer to a quantum computation as resulting, at any one step, in a *superposition* of all the branches of its tree simultaneously.

2.2. Notation and an example. It is useful to have a notation to denote superpositions (that is, entire levels of a computation tree). We say that at any step i, the computation is in a superposition of all the configurations $|c_1\rangle, \ldots, |c_k\rangle$ corresponding to nodes that appear in level i of the tree representing the computation, each $|c_j\rangle$ having amplitude α_j. (Borrowing quantum mechanics notation, we distinguish symbols representing configurations from those representing amplitudes by placing $|\rangle$ brackets around configuration symbols.) An abbreviated notation for this superposition is $\sum_j \alpha_j |c_j\rangle$; as we shall see, the suggestive addition/summation notation for superpositions is quite appropriate. A simple example of a unitary quantum step is the quantum "fair coin flip" performed upon a single bit. It is represented by the following matrix M:

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

M acts on 2-element column vectors whose top and bottom entries represent the amplitudes of the states $|0\rangle$ and $|1\rangle$, respectively. A bit in state $|0\rangle$ is transformed by M into a superposition of $|0\rangle$ and $|1\rangle$, both with amplitude $1/\sqrt{2}$. Similarly, a bit in state $|1\rangle$ is transformed into a superposition of $|0\rangle$ and $|1\rangle$ with amplitude of magnitude $1/\sqrt{2}$ in each case, but with the sign, or *phase* of the amplitude of $|1\rangle$ being negative. In other words, the state $|0\rangle$ is transformed into $(1/\sqrt{2})|0\rangle + (1/\sqrt{2})|1\rangle$, and $|1\rangle$ becomes $(1/\sqrt{2})|0\rangle + (-1/\sqrt{2})|1\rangle$.

It turns out that this transformation is its own inverse. For example, performing it a second time on a bit that was originally in state $|0\rangle$ produces $(1/\sqrt{2})((1/\sqrt{2})|0\rangle + (1/\sqrt{2})|1\rangle) + (1/\sqrt{2})((1/\sqrt{2})|0\rangle + (-1/\sqrt{2})|1\rangle)$. Collecting like terms in this expression (here we see the aptness of the addition/summation notation) allows us to obtain the amplitude of each distinct configuration, which in this case is 1 for $|0\rangle$ and 0 for $|1\rangle$. Similarly, performing this same transformation twice on the initial configuration $|1\rangle$ gives us $|1\rangle$ (with amplitude 1, and hence probability 1) again.

In a system of n bits, with 2^n possible configurations, we can perform such a transformation on each bit independently in sequence. The matrices representing these transformations will be of dimension $2^n \times 2^n$, of course; their rows, each corresponding to a different configuration, will each have two nonzero entries, taken from either the top or bottom row of M. Their columns will similarly have two nonzero entries each, taken from either the left or right column of M. Also, they will all be unitary, since they each represent a local, unitary transformation.

The result of performing these n different transformations in sequence will be a superposition of all possible n-bit strings. The amplitude of each string at the end of the n transformations will have magnitude $2^{-n/2}$. As the transformations are applied in turn, the phase of a resulting configuration is changed when a bit that was previously a 1 remains a 1 after the transformation is performed. Hence, the phase of the amplitude of string x is determined by the parity of the dot product of the original configuration string and x. More precisely, if the string w is the original configuration, then performing the product transformation composed of these n transformations in sequence will result in the superposition

$$2^{-n/2} \sum_x (-1)^{w \cdot x} |x\rangle.$$

This product transformation was introduced in [DJ92] and is referred to in [BV93] as the Fourier transformation F.

3. Using quantum computation.

3.1. Problem: Is a function invariant under some xor-mask? Suppose we are given a function $f : \{0,1\}^n \to \{0,1\}^m$, with $m \geq n$, and we are promised that either f is one-to-one, or there exists a nontrivial n-bit string s such that for any pair of distinct inputs x and x', $f(x)$ and $f(x')$ are equal if and only if the bits of x and x' differ in exactly those positions where the bits of s are 1. We wish to determine which of these conditions holds for f, and, in the second case, to find s.

DEFINITION 3.1. *Given a function $f : \{0,1\}^n \to \{0,1\}^m$, with $m \geq n$, the* xor-mask invariance *of f ($XMI(f)$) is*

- *s, if there exists a nontrivial string s of length n such that $\forall x \neq x'(f(x) = f(x') \Leftrightarrow x' = x \oplus s)$, where $\oplus$ denotes bitwise exclusive-or*;
- *0^n, if f is one-to-one*; *and*
- *undefined otherwise.*

THEOREM 3.2. *There exists an algorithm for a QTM which computes $XMI(f)$ (if it is defined), with zero error probability, in expected time $O(nT_f(n) + G(n))$, where $T_f(n)$ is the time required to compute f on inputs of size n, and $G(n)$ is the time required to solve an $n \times n$ linear system of equations over $\mathcal{Z}_2$.*

Proof. The algorithm is very simple, consisting essentially of (an expected) $O(n)$ repetitions of the following routine.

Routine **Fourier-twice**

1. Perform the transformation F described above on a string of n zeros, producing $2^{-n/2} \sum_x |x\rangle$.
2. Compute $f(x)$, concatenating the answer to x, thus producing $2^{-n/2} \sum_x |(x, f(x))\rangle$.
3. Perform F on x, producing $2^{-n} \sum_y \sum_x (-1)^{x \cdot y} |(y, f(x))\rangle$.

End **Fourier-twice**

Note that the (deterministic) computation of $(x, f(x))$ from x in time $T_f(n)$ in step 2 can always be made reversible (and hence unitary) at the cost of only a constant factor in the number of computation steps. This is due to a result obtained independently by Lecerf [Lec63] and Bennett [Benn73].

Suppose f is one-to-one. Then after each performance of **Fourier-twice**, all the possible configurations $|(y, f(x))\rangle$ in the superposition will be distinct, and their amplitudes will therefore all be 2^{-n}, up to phase. Their probabilities will therefore each be 2^{-2n}, and k independent repetitions of **Fourier-twice** will thus yield k configurations each distributed uniformly and independently over configurations of the form $|(y, f(x))\rangle$.

Now suppose that there is some s such that $\forall x \neq x'(f(x) = f(x') \Leftrightarrow x' = x \oplus s)$. Then for each y and x, the configurations $|(y, f(x))\rangle$ and $|(y, f(x \oplus s))\rangle$ are identical, and the amplitude $\alpha(x, y)$ of this configuration will be $2^{-n}((-1)^{x \cdot y} + (-1)^{(x \oplus s) \cdot y})$. Note that if $y \cdot s \equiv 0 \pmod 2$, then $x \cdot y \equiv (x \oplus s) \cdot y \pmod 2$, and $\alpha(x, y) = 2^{-n+1}$; otherwise $\alpha(x, y) = 0$. Thus k independent repetitions of **Fourier-twice** will yield k configurations distributed uniformly and independently over configurations of the form $|(y, f(x))\rangle$ such that $y \cdot s \equiv 0 \pmod 2$.

In both cases, after an expected $O(n)$ repetitions of **Fourier-twice**, sufficiently many linearly independent values of y will have been collected that the nontrivial string s^* whose dot product with each is even is uniquely determined. s^* can then easily be obtained by solving the linear system of equations defined by these values of y. (Once the solution space is constrained to one d imension in $(\mathcal{Z}_2)^n$, it will yield

exactly two solutions, one of which is nontrivial.) In the second case, this string s^* must be the s we are looking for, since we know that $y \cdot s \equiv 0 \pmod 2$ for each y generated in the second case. On the other hand, in the first case, where f is one-to-one, s^* will simply be a random string. Hence, evaluation of, say, $f(0^n)$ and $f(s^*)$ will reveal whether we have found the true s (in the second case) or simply selected a random string (in the first case). □

If we allow a bounded error probability, we can use essentially the same algorithm to solve slightly less constrained promise problems. For example, in the case where f is one-to-one, the outputs of n/ϵ repetitions of **Fourier-twice** (for constants $\epsilon < 1$) will with probability $1 - 2^{O(n)}$ contain a basis for $(\mathcal{Z}_2)^n$. On the other hand, if there exists an s such that for a fraction at least $1 - \epsilon/n$ of possible choices of x, $f(x) = f(x \oplus s)$, then the outputs of n/ϵ repetitions of **Fourier-twice** will still all satisfy $y \cdot s \equiv 0 \pmod 2$, with constant probability, regardless of any other properties of f. Hence we can efficiently distinguish between these two classes of function (for appropriate ϵ) on a quantum computer with negligible error probability.

3.2. Relativized hardness of our problem. Now, in a relativized setting, suppose that an oracle is equiprobably either an oracle uniformly distributed among permutations on n-bit values or an oracle uniformly distributed among those two-to-one functions f for which there exists a unique nontrivial s such that $f(x)$ always equals $f(x \oplus s)$. Then a classical probabilistic oracle TM would require exponentially many oracle queries to successfully distinguish the two cases with probability nonnegligibly greater than $1/2$.

THEOREM 3.3. *Let O be an oracle constructed as follows: for each n, a random n-bit string $s(n)$ and a random bit $b(n)$ are uniformly chosen from $\{0,1\}^n$ and $\{0,1\}$, respectively. If $b(n) = 0$, then the function $f_n : \{0,1\}^n \to \{0,1\}^n$ chosen for O to compute on n-bit queries is a random function uniformly distributed over permutations on $\{0,1\}^n$; otherwise, it is a random function uniformly distributed over two-to-one functions such that $f_n(x) = f_n(x \oplus s(n))$ for all x, where $\oplus$ denotes bitwise exclusive-or. Then any PTM that queries O no more than $2^{n/4}$ times cannot correctly guess $b(n)$ with probability greater than $(1/2) + 2^{-n/2}$, over choices made in the construction of O.*

Proof. Consider any such PTM M. We say that M's choice of the first k queries is *good* for n if M queries O at two n-bit input values whose exclusive or is $s(n)$. If M makes a good choice of $2^{n/4}$ queries for n, then the distribution on answers given by O differs depending on $b(n)$; otherwise, the distributions are identical (i.e., random, uniformly distributed distinct values for each distinct query). Since the probability that M guesses $b(n)$ is only greater than $1/2$ when its choices are good for n, this probability is also bounded above by $1/2 + \delta$, where δ is the probability that M's queries are good for n. Hence, we need only calculate a bound on δ to obtain a bound on M's probability of guessing $b(n)$.

Note that the probability that M's first k queries are good for n is equal to the sum of the conditional probabilities, for each of the queries, that M's queries up to and including that query are good for n, given that the previous ones were not. Note also that given a particular fixed sequence of j queries (and their answers) which is not good for n, the conditional distribution on $s(n)$ (over choices made in constructing O) is uniform over the elements of $\{0,1\}^n$ for which those j queries are still not good for n. (This is because all such possible sequences are equally likely for any $s(n)$, and there are equally many such sequences regardless of $s(n)$.) For example, if the j queries are such that their pairwise bitwise exclusive-ors are all distinct, then $s(n)$ is

conditionally distributed uniformly over the $2^n - j(j-1)/2$ possible values for which the sequence of queries is still not good for n.

Now, consider M's kth oracle query to O, assuming that M's first $k-1$ queries were not good for n. This kth query is completely determined by O's answers to the first $k-1$ queries and by M's probabilistic choices; we will call it q. The probability (over choices made in constructing O) that O's answer to q is the same as its answer to (a distinct) one of the $k-1$ previous queries (and hence that M's first k queries are good for n) is at most $k/(2^n - (k-2)(k-1)/2)$, since there are at most k choices of $s(n)$ (which was uniformly chosen from $\{0,1\}^n$) for which such a "collision" occurs, and $s(n)$ is conditionally distributed uniformly over all but the (at most) $(k-1)(k-2)/2$ values for which M's first $k-1$ queries is not good. Hence, for any sequence of $j = 2^{n/4}$ queries, the probability that it is good for n is at most $\sum_{k=1}^{j}(k/(2^n-(k-2)(k-1)/2)) \leq \sum_{k=1}^{j}(k/(j^4-j^2)) \leq (j^2+j)/(2(j^4-j^2)) \leq 2^{-n/2}$ (for $n \geq 1$). It follows that M cannot estimate $b(n)$ with probability better than $(1/2) + 2^{-n/2}$. □

We can also use Theorem 3.3 to prove the existence of a specific oracle relative to which there is an exponential gap (in terms of classical computing time) between BPP and its quantum analogue, BQP (defined in the natural way; see [BV93]). Let E be the (countable) set of classical oracle PTMs making at most $2^{n/4}$ queries on input 1^n. We say that $M \in E$ *solves* an oracle O generated as in the above theorem if for infinitely many n, M computes $b(n)$, with error bounded away from $1/2$ by some constant, on input 1^n. Theorem 3.3 tells us that for any M, the probability that M solves an O so chosen is 0. Since E is countable, an oracle O so generated will therefore with probability 1 be solved by no $M \in E$. Hence with probability 1 the language $\{1^n | b(n) = 1\}$, for $b(n)$ chosen as in Theorem 3.3, cannot be accepted with error bounded away from $1/2$ by any $M \in E$.

THEOREM 3.4. *There exist an oracle O and constant ϵ relative to which $BQP \not\subseteq PTIME(2^{\epsilon n})$ (with two-sided error).*

4. Conclusion. Since any quantum computer running in polynomial time can be fairly easily simulated in $PSPACE$, as was pointed out in [BV93], we are unlikely to be able to prove anytime soon that BQP is larger than P. However, Shor [Sho94] has recently made a huge advance toward establishing the complexity-theoretic advantage of the quantum model compared to the classical one, by giving quantum polynomial-time algorithms for two well-known presumed-hard problems: computing discrete logarithms modulo an arbitrary prime and factoring integers. His algorithms follow the very rough outline of the ones presented here, but with many additional sophistications that allow them to work over the field $\mathcal{Z}_p^*$ (for primes p such that $p-1$ is smooth) rather than $(\mathcal{Z}_2)^n$, and to extract much more than a single bit of information per iteration. A logical next step might be to try to separate BPP and BQP based on a more general complexity-theoretic assumption such as $P \neq NP$ or the existence of one-way functions. Alternatively, it may be possible to prove limits to the advantages of quantum computation through simulation results of some kind. (In [BBBV94], for example, oracle methods are used to give evidence that $NP \not\subseteq BQP$. On the other hand, Grover [Gro96] has recently shown that for NP-complete decision problems, the associated search problem with solutions of size n can be solved probabilistically, with bounded error, in time $2^{n/2}$ on a quantum computer—i.e., more efficiently than any known classical probabilistic algorithm.)

Another natural question regarding the model is whether the "fair quantum coin flip" suffices as a universal nonclassical step, the way its classical counterpart, the

fair coin flip, suffices as a universal (classical) probabilistic step. Recent work in this direction (see, for instance, [DiV95], [BBCD95]) has shown that there are many choices of a single nonclassical operation that will in fact suffice in simulating quantum computations which use arbitrary feasible quantum operations; however, it is not known whether the "fair quantum coin flip" is one such choice.

Another issue is that of alternative models of quantum computation. Yao [Yao93] has presented a quantum circuit model (following [Deu89]) and proven it equivalent to the QTM. In contrast, it is not yet known whether a quantum cellular automaton is equivalent or more powerful (see [DST96]). Still other distinct quantum-based computational models may exist, as well. For example, any unitary "evolution" matrix describing a quantum computation (in any model) is related (by Schrodinger's equation) to a corresponding Hermitian "Hamiltonian" matrix which describes the same process. There is also a natural notion of locality for Hamiltonians—but evolution matrices and their associated Hamiltonians are not necessarily both local or both nonlocal. It is therefore unclear whether even the definition of BQP (for QTMs or for any other model) is the same for operator-based and Hamiltonian-based encodings. (Feynman has shown, in [Fey86], that the Hamiltonian-based model is at least as powerful as the unitary operator-based one; whether the reverse is true is not known.)

Beyond the question of models is the matter of their implementation. For example, any physical realization of a quantum computer would necessarily be subject to some error; exact superpositions would end up being represented by approximations just as deterministic discrete computations and random coin flips are approximated in modern computers using analog quantities such as voltages. Considerable work has been done on the feasibility of resiliently simulating true randomness with "approximate randomness" (see, for example, [VV85], [CG88]); similar work is necessary to determine if computation using approximations of quantum superpositions can be made comparably resilient. Recent work by Shor [Sho96] on quantum error-correcting codes has made progress toward this goal, showing that errors conforming to a certain restrictive model can in fact be corrected. However, it is not known how well that model covers the types of error likely to be encountered in a practical quantum computer. Resolution of these and other theoretical issues would be a crucial step toward understanding both the utility and the ultimate feasibility of implementing a quantum computer.

Acknowledgments. Many thanks to Charles Bennett, Ethan Bernstein, Gilles Brassard, Jeroen van de Graaf, Richard Jozsa, and Dominic Mayers for valuable insights and helpful discussion.

REFERENCES

[Beni82] P. Benioff, *Quantum mechanical Hamiltonian models of Turing machines*, J. Statist. Phys., 29 (1982), pp. 515–546.

[Benn73] C. H. Bennett, *Logical reversibility of computation*, IBM J. Res. Develop., 17 (1973), pp. 525–532.

[BBBV94] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani, *Strengths and weaknesses of quantum computing*, SIAM J. Comput., 26 (1997), pp. 1510–1523.

[BB92] A. Berthiaume and G. Brassard, *The quantum challenge to structural complexity theory*, in Proc. 7th IEEE Conference on Structure in Complexity Theory, Boston, MA, 1992, pp. 132–137.

[BB93] A. Berthiaume and G. Brassard, *Oracle quantum computing*, J. Modern Optics, 41 (1994), pp. 2521–2535.

[BBCD95] A. Barenco, C. Bennett, R. Cleve, D. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter, *Elementary gates for quantum computation*, Phys. Rev. A, 52 (1995), pp. 3457–3467.

[BV93] E. Bernstein and U. Vazirani, *Quantum complexity theory*, in Proc. 25th ACM Symp. on Theory of Computation, San Diego, CA, 1993, pp. 11–20; SIAM J. Comput., 26 (1997), pp. 1411–1473.

[CG88] B. Chor and O. Goldreich, *Unbiased bits from sources of weak randomness and probabilistic communication complexity*, SIAM J. Comput., 17 (1988), pp. 230–261.

[Deu85] D. Deutsch, *Quantum theory, the Church–Turing principle and the universal quantum computer*, in Proc. Roy. Soc. London Ser. A, 400 (1985), pp. 73–90.

[Deu89] D. Deutsch, *Quantum computational networks*, in Proc. Roy. Soc. London Ser. A, 425 (1989), pp. 73–90.

[DiV95] D. DiVincenzo, *Two-bit gates are universal for quantum computation*, Phys. Rev. A, 51 (1995), pp. 1015–1022.

[DJ92] D. Deutsch and R. Jozsa, *Rapid solution of problems by quantum computation*, in Proc. Roy. Soc. London Ser. A, 439 (1992), pp. 553–558.

[DST96] C. Dürr, H. Lê Thanh, and M. Santha, *A decision procedure for well-formed linear quantum cellular automata*, in Proc. 13th Symposium on Theoretical Aspects of Computer Science, Grenoble, France, 1996, pp. 281–292.

[Fey82] R. Feynman, *Simulating physics with computers*, Internat. J. Theoret. Phys., 21 (1982), pp. 467–488.

[Fey86] R. Feynman, *Quantum mechanical computers*, Found. Phys., 16 (1986), pp. 507–531.

[Gro96] L. Grover, *A fast quantum mechanical algorithm for database search*, in Proc. 28th ACM Symp. on Theory of Computation, Philadelphia, PA, 1996, pp. 212–219.

[Lec63] Y. Lecerf, *Machines de Turing reversibles. Récursive insolubilité en $n\epsilon N$ de l'équation $u = \theta^n$ ou θ est un "isomorphism de codes"*, Comptes Rendus de L'Academie Francaise des Sciences, 257 (1963), pp. 2597–2600.

[Sho94] P. Shor, *Algorithms for quantum computation: Discrete log and factoring*, in Proc. 35th IEEE Symp. on Foundations of Computer Science, Santa Fe, NM, 1994, pp. 124–134.

[Sho96] P. Shor, *Fault-tolerant quantum computation*, in Proc. 37th IEEE Symp. on Foundations of Computer Science, Burlington, VT, 1996, pp. 56–65.

[Sim94] D. Simon, *On the power of quantum computation*, in Proc. 35th IEEE Symp. on Foundations of Computer Science, Santa Fe, NM, 1994, pp. 116–123.

[VV85] U. V. Vazirani and V. V. Vazirani, *Random polynomial time is equal to slightly-random polynomial time*, in Proc. 26th IEEE Symp. on Foundations of Computer Science, Portland, OR, 1985, pp. 417–428.

[Yao93] A. Yao, *Quantum circuit complexity*, in Proc. 34th IEEE Symp. on Foundations of Computer Science, Palo Alto, CA, 1993, pp. 352–361.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1484–1509, October 1997

009

POLYNOMIAL-TIME ALGORITHMS FOR PRIME FACTORIZATION AND DISCRETE LOGARITHMS ON A QUANTUM COMPUTER*

PETER W. SHOR†

Abstract. A digital computer is generally believed to be an efficient universal computing device; that is, it is believed able to simulate any physical computing device with an increase in computation time by at most a polynomial factor. This may not be true when quantum mechanics is taken into consideration. This paper considers factoring integers and finding discrete logarithms, two problems which are generally thought to be hard on a classical computer and which have been used as the basis of several proposed cryptosystems. Efficient randomized algorithms are given for these two problems on a hypothetical quantum computer. These algorithms take a number of steps polynomial in the input size, e.g., the number of digits of the integer to be factored.

Key words. algorithmic number theory, prime factorization, discrete logarithms, Church's thesis, quantum computers, foundations of quantum mechanics, spin systems, Fourier transforms

AMS subject classifications. 81P10, 11Y05, 68Q10, 03D10

PII. S0097539795293172

1. Introduction.

One of the first results in the mathematics of computation, which underlies the subsequent development of much of theoretical computer science, was the distinction between computable and noncomputable functions shown in papers of Church [1936], Post [1936], and Turing [1936]. The observation that several apparently different definitions of what it meant for a function to be computable yielded the same set of computable functions led to the proposal of Church's thesis, which says that all computing devices can be simulated by a Turing machine. This thesis greatly simplifies the study of computation, since it reduces the potential field of study from any of an infinite number of potential computing devices to Turing machines. Church's thesis is not a mathematical theorem; to make it one would require a precise mathematical description of a computing device. Such a description, however, would leave open the possibility of some practical computing device which did not satisfy this precise mathematical description and thus would make the resulting theorem weaker than Church's original thesis.

With the development of practical computers, it became apparent that the distinction between computable and noncomputable functions was much too coarse; computer scientists are now interested in the exact efficiency with which specific functions can be computed. This exact efficiency, on the other hand, was found to be too precise a quantity to work with easily. The generally accepted compromise between coarseness and precision distinguishes efficiently from inefficiently computable functions by whether the length of the computation scales polynomially or superpolynomially with the input size. The class of problems which can be solved by algorithms having a number of steps polynomial in the input size is known as P.

For this classification to make sense, it must be machine independent. That is, the question of whether a function is computable in polynomial time must be independent

*Received by the editors October 16, 1995; accepted for publication (in revised form) December 2, 1996. This paper is an expanded version of a paper that appeared in *Proc. 35th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 124–134.

http://www.siam.org/journals/sicomp/26-5/29317.html

†AT&T Labs–Research, Room C237, 180 Park Avenue, Florham Park, NJ 07932 (shor@research.att.com).

of the type of computing device used. This corresponds to the following quantitative version of Church's thesis, which has been called the "strong Church's thesis" by Vergis, Steiglitz, and Dickinson [1986] and which makes up half of the "invariance thesis" of van Emde Boas [1990].

THESIS 1.1 (quantitative Church's thesis). *Any physical computing device can be simulated by a Turing machine in a number of steps polynomial in the resources used by the computing device.*

Readers who are not comfortable with Turing machines may think instead of digital computers having an amount of memory that grows linearly with the length of the computation, as these two classes of computing machines can efficiently simulate each other. In statements of this thesis, the Turing machine is sometimes augmented with a random number generator, as it has not yet been determined whether there are pseudorandom number generators which can efficiently simulate truly random number generators for all purposes.

There are two escape clauses in the above thesis. One of these is the word "physical." Researchers have produced machine models that violate the above quantitative Church's thesis, but most of these have been ruled out by some reason for why they are not physical, that is, why they could not be built and made to work. The other escape clause in the above thesis is the word "resources," the meaning of which is not completely specified above. There are generally two resources which limit the ability of digital computers to solve large problems: time (computational steps) and space (memory). There are more resources pertinent to analog computation; some proposed analog machines that seem able to solve NP-complete problems in polynomial time have required exponentially precise parts or an exponential amount of energy. (See Vergis, Steiglitz, and Dickinson [1986] and Steiglitz [1988]; this issue is also implicit in the papers of Canny and Reif [1987] and Choi, Sellen, and Yap [1995] on three-dimensional shortest paths.)

For quantum computation, in addition to space and time there is also a third potentially important resource: precision. For a quantum computer to work, at least in any currently envisioned implementation, it must be able to make changes in the quantum states of objects (e.g., atoms, photons, or nuclear spins). These changes can clearly not be perfectly accurate but must contain some small amount of inherent imprecision. If this imprecision is constant (i.e., it does not depend on the size of the input), then it is not known how to compute any functions in polynomial time on a quantum computer that cannot also be computed in polynomial time on a classical computer with a random number generator[1]. However, if we let the precision grow polynomially in the input size (so the number of *bits* of precision grows logarithmically in the input size), we appear to obtain a more powerful type of computer. Allowing the same polynomial growth in precision does not appear to confer extra computing power to classical mechanics, although allowing exponential growth in precision may [Hartmanis and Simon 1974; Vergis, Steiglitz, and Dickinson 1986].

As far as we know, what precision is possible in quantum state manipulation is dictated not by fundamental physical laws but by the properties of the materials from which and the architecture with which a quantum computer is built. It is currently not clear which architectures, if any, will give high precision, and what this precision will be. If the precision of a quantum computer is large enough to make it more powerful

[1] Note added in proof. This is no longer true. See Aharonov and Ben-Or [Proc. 29th Annual ACM Symposium on Theory of Computing, ACM, New York, 1997, pp. 176–188]; Gottesman, Evslin, Kakade, and Preskill [manuscript, 1997]; Kitaev [manuscript, 1997], Knill, Laflamme, and Zurek [LANL e-print quant-ph/9702058, Los Alamos National Laboratories, Los Alamos, NM, 1997].

than a classical computer, then in order to understand its potential it is important to think of precision as a resource that can vary. Treating the precision as a large constant (even though it is almost certain to be constant for any given machine) would be comparable to treating a classical digital computer as a finite automaton—since any given computer has a fixed amount of memory, this view is technically correct; however, it is not particularly useful.

Because of the remarkable effectiveness of our mathematical models of computation, computer scientists have tended to forget that computation is dependent on the laws of physics. This can be seen in the statement of the quantitative Church's thesis in van Emde Boas [1990], where the word "physical" in the above phrasing is replaced by the word "reasonable." It is difficult to imagine any definition of "reasonable" in this context which does not mean "physically realizable," i.e., that this computing machine could actually be built and would work.

Computer scientists have become convinced of the truth of the quantitative Church's thesis through the failure of all proposed counterexamples. Most of these proposed counterexamples have been based on the laws of classical mechanics; however, the universe is in reality quantum mechanical. Quantum mechanical objects often behave quite differently from how our intuition, based on classical mechanics, tells us they should. It thus seems plausible that the natural computing power of classical mechanics corresponds to that of Turing machines,[2] while the natural computing power of quantum mechanics might be greater.

The first person to look at the interaction between computation and quantum mechanics appears to have been Benioff [1980, 1982a, 1982b]. Although he did not ask whether quantum mechanics conferred extra power to computation, he showed that reversible unitary evolution was sufficient to realize the computational power of a Turing machine, thus showing that quantum mechanics is computationally at least as powerful as classical computers. This work was fundamental in making later investigation of quantum computers possible.

Feynman [1982, 1986] seems to have been the first to suggest that quantum mechanics might be computationally more powerful than Turing machines. He gave arguments as to why quantum mechanics is intrinsically computationally expensive to simulate on a classical computer. He also raised the possibility of using a computer based on quantum mechanical principles to avoid this problem, thus implicitly asking the converse question, "By using quantum mechanics in a computer can you compute more efficiently than on a classical computer?" The first to ask this question explicitly was Deutsch [1985, 1989]. In order to study this question, he defined both quantum Turing machines and quantum circuits and investigated some of their properties.

The question of whether using quantum mechanics in a computer allows one to obtain more computational power was more recently addressed by Deutsch and Jozsa [1992] and Berthiaume and Brassard [1992, 1994]. These papers showed that there are problems which quantum computers can quickly solve exactly, but that classical computers can only solve quickly with high probability and the aid of a random number generator. However, these papers did not show how to solve any problem in quantum polynomial time that was not already known to be solvable in polynomial time with the aid of a random number generator, allowing a small

[2]See Vergis, Steiglitz, and Dickinson [1986], Steiglitz [1988], and Rubel [1989]. I believe that this question has not yet been settled and is worthy of further investigation. In particular, turbulence seems a good candidate for a counterexample to the quantitative Church's thesis because the nontrivial dynamics on many length scales appear to make it difficult to simulate on a classical computer.

probability of error; this is the characterization of the complexity class BPP (bounded error probability probabilistic polynomial time), which is widely viewed as the class of efficiently solvable problems.

Further work on this problem was stimulated by Bernstein and Vazirani [1993]. One of the results contained in their paper was an oracle problem (that is, a problem involving a "black box" subroutine that the computer is allowed to perform but for which no code is accessible) which can be done in polynomial time on a quantum Turing machine but which requires superpolynomial time on a classical computer. This result was improved by Simon [1994], who gave a much simpler construction of an oracle problem which takes polynomial time on a quantum computer but requires *exponential* time on a classical computer. Indeed, while Bernstein and Vaziarni's problem appears contrived, Simon's problem looks quite natural. Simon's algorithm inspired the work presented in this paper.

Two number theory problems which have been studied extensively but for which no polynomial-time algorithms have yet been discovered are finding discrete logarithms and factoring integers [Pomerance 1987, Gordon 1993, Lenstra and Lenstra 1993, Adleman and McCurley 1994]. These problems are so widely believed to be hard that several cryptosystems based on their difficulty have been proposed, including the widely used RSA public key cryptosystem developed by Rivest, Shamir, and Adleman [1978]. We show that these problems can be solved in polynomial time on a quantum computer with a small probability of error.

Currently, nobody knows how to build a quantum computer, although it seems as though it might be possible within the laws of quantum mechanics. Some suggestions have been made as to possible designs for such computers [Teich, Obermayer, and Mahler 1988; Lloyd 1993, 1994; Cirac and Zoller 1995; DiVincenzo 1995; Sleator and Weinfurter 1995; Barenco et al. 1995b; Chuang and Yamomoto 1995], but there will be substantial difficulty in building any of these [Landauer 1995, 1997; Unruh 1995; Chuang et al. 1995; Palma, Suominen, and Ekert 1996]. The most difficult obstacles appear to involve the decoherence of quantum superpositions through the interaction of the computer with the environment, and the implementation of quantum state transformations with enough precision to give accurate results after many computation steps. Both of these obstacles become more difficult as the size of the computer grows, so it may turn out to be possible to build small quantum computers, while scaling up to machines large enough to do interesting computations may present fundamental difficulties.

Even if no useful quantum computer is ever built, this research does illuminate the problem of simulating quantum mechanics on a classical computer. Any method of doing this for an arbitrary Hamiltonian would necessarily be able to simulate a quantum computer. Thus, any general method for simulating quantum mechanics with at most a polynomial slowdown would lead to a polynomial-time algorithm for factoring.

The rest of this paper is organized as follows. In section 2, we introduce the model of quantum computation, the *quantum gate array,* that we use in the rest of the paper. In sections 3 and 4, we explain two subroutines that are used in our algorithms: reversible modular exponentiation in section 3 and quantum Fourier transforms in section 4. In section 5, we give our algorithm for prime factorization, and in section 6, we give our algorithm for extracting discrete logarithms. In section 7, we give a brief discussion of the practicality of quantum computation and suggest possible directions for further work.

2. Quantum computation. In this section we give a brief introduction to quantum computation, emphasizing the properties that we will use. We will describe only *quantum gate arrays,* or *quantum acyclic circuits,* which are analogous to acyclic circuits in classical computer science. This model was originally studied by Yao [1993] and is closely related to the quantum computational networks discussed by Deutsch [1989]. For other models of quantum computers, see references on quantum Turing machines [Deutsch 1985; Bernstein and Vazirani 1993; Yao 1993] and quantum cellular automata [Feynman 1986; Margolus 1986, 1990; Lloyd 1993; Biafore 1994]. If they are allowed a small probability of error, quantum Turing machines and quantum gate arrays can compute the same functions in polynomial time [Yao 1993]. This may also be true for the various models of quantum cellular automata, but it has not yet been proved. This gives evidence that the class of functions computable in quantum polynomial time with a small probability of error is robust in that it does not depend on the exact architecture of a quantum computer. By analogy with the classical class BPP, this class is called BQP (bounded error probability quantum polynomial time).

Consider a system with n components, each of which can have two states. Whereas in classical physics, a complete description of the state of this system requires only n bits, in quantum physics, a complete description of the state of this system requires $2^n - 1$ complex numbers. To be more precise, the state of the quantum system is a point in a 2^n-dimensional vector space. For each of the 2^n possible classical positions of the components, there is a basis state of this vector space which we represent, for example, by $|011\cdots0\rangle$ meaning that the first bit is 0, the second bit is 1, and so on. Here, the *ket* notation $|x\rangle$ means that x is a (pure) quantum state. (Mixed states will not be discussed in this paper, and thus we do not define them; see a quantum theory book such as Peres [1993] for their definition.) The *Hilbert space* associated with this quantum system is the complex vector space with these 2^n states as basis vectors, and the state of the system at any time is represented by a unit-length vector in this Hilbert space. As multiplying this state vector by a unit-length complex phase does not change any behavior of the state, we need only $2^n - 1$ complex numbers to completely describe the state. We represent this superposition of states by

$$\sum_{i=0}^{2^n-1} a_i|S_i\rangle, \tag{2.1}$$

where the amplitudes a_i are complex numbers such that $\sum_i |a_i|^2 = 1$ and each $|S_i\rangle$ is a basis vector of the Hilbert space. If the machine is measured (with respect to this basis) at any particular step, the probability of seeing basis state $|S_i\rangle$ is $|a_i|^2$; however, measuring the state of the machine projects this state to the observed basis vector $|S_i\rangle$. Thus, looking at the machine during the computation will invalidate the rest of the computation. General quantum mechanical measurements, i.e., POVMs (positive operator valued measurement, see [Peres 1993]), can be considerably more complicated than the case of projection onto the canonical basis to which we restrict ourselves in this paper. This does not greatly restrict our model of computation, since measurements in other reasonable bases, as well as other local measurements, can be simulated by first using quantum computation to perform a change of basis and then performing a measurement in the canonical basis.

In order to use a physical system for computation, we must be able to change the state of the system. The laws of quantum mechanics permit only unitary transformations of state vectors. A unitary matrix is one whose conjugate transpose is equal to

its inverse, and requiring state transformations to be represented by unitary matrices ensures that summing the probability over all possible outcomes yields 1. The definition of quantum circuits (and quantum Turing machines) allows only *local* unitary transformations—that is, unitary transformations on a fixed number of bits. This is physically justified because, given a general unitary transformation on n bits, it is not clear how one could efficiently implement it physically, whereas two-bit transformations can at least in theory be implemented by relatively simple physical systems [Cirac and Zoller 1995; DiVincenzo 1995; Sleator and Weinfurter 1995; Chuang and Yamomoto 1995]. While general n-bit transformations can always be built out of two-bit transformations [DiVincenzo 1995; Sleator and Weinfurter 1995; Lloyd 1995; Deutsch, Barenco, and Ekert 1995], the number required will often be exponential in n [Barenco et al. 1995a]. Thus, the set of two-bit transformations form a set of building blocks for quantum circuits in a manner analogous to the way a universal set of classical gates (such as the AND, OR, and NOT gates) form a set of building blocks for classical circuits. In fact, for a universal set of quantum gates, it is sufficient to take all one-bit gates and a single type of two-bit gate, the controlled NOT gate (also called the XOR or parity gate), which negates the second bit if and only if the first bit is 1.

Perhaps an example will be informative at this point. A quantum gate can be expressed as a truth table: for each input basis vector we need to give the output of the gate. One such gate is

$$\begin{aligned} &|00\rangle \to |00\rangle, \\ &|01\rangle \to |01\rangle, \\ &|10\rangle \to \tfrac{1}{\sqrt{2}}(|10\rangle + |11\rangle), \\ &|11\rangle \to \tfrac{1}{\sqrt{2}}(|10\rangle - |11\rangle). \end{aligned} \tag{2.2}$$

Not all truth tables correspond to physically feasible quantum gates, as many truth tables will not give rise to unitary transformations.

The same gate can also be represented as a matrix. The rows correspond to input basis vectors. The columns correspond to output basis vectors. The (i, j) entry gives, when the ith basis vector is input to the gate, the coefficient of the jth basis vector in the corresponding output of the gate. The truth table above would then correspond to the following matrix:

$$\begin{array}{c|cccc} & |00\rangle & |01\rangle & |10\rangle & |11\rangle \\ \hline |00\rangle & 1 & 0 & 0 & 0 \\ |01\rangle & 0 & 1 & 0 & 0 \\ |10\rangle & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ |11\rangle & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{array}. \tag{2.3}$$

A quantum gate is feasible if and only if the corresponding matrix is unitary, i.e., its inverse is its conjugate transpose.

Suppose that our machine is in the superposition of states

$$\tfrac{1}{\sqrt{2}}|10\rangle - \tfrac{1}{\sqrt{2}}|11\rangle \tag{2.4}$$

and we apply the unitary transformation represented by (2.2) and (2.3) to this state. The resulting output will be the result of multiplying the vector (2.4) by the matrix (2.3). The machine will thus go to the superposition of states

$$\tfrac{1}{2}\left(|10\rangle + |11\rangle\right) - \tfrac{1}{2}\left(|10\rangle - |11\rangle\right) \;=\; |11\rangle. \tag{2.5}$$

This example shows the potential effects of interference on quantum computation. Had we started with either the state $|10\rangle$ or the state $|11\rangle$, there would have been a chance of observing the state $|10\rangle$ after the application of the gate (2.3). However, when we start with a superposition of these two states, the probability amplitudes for the state $|10\rangle$ cancel, and we have no possibility of observing $|10\rangle$ after the application of the gate. Notice that the output of the gate would have been $|10\rangle$ instead of $|11\rangle$ had we started with the superposition of states

$$\frac{1}{\sqrt{2}}|10\rangle + \frac{1}{\sqrt{2}}|11\rangle, \tag{2.6}$$

which has the same probabilities of being in any particular configuration if it is observed as does the superposition (2.4).

If we apply a gate to only two bits of a longer vector (now our circuit must have more than two wires), for each basis vector we apply the transformation given by the gate's truth table to the two bits on which the gate is operating, and leave the other bits alone. This corresponds to multiplying the whole state by the tensor product of the gate matrix on those two bits with the identity matrix on the remaining bits. For example, applying the transformation represented by (2.2) and (2.3) to the first two bits of the basis vector $|110\rangle$ yields the vector $\frac{1}{\sqrt{2}}(|100\rangle - |110\rangle)$.

A quantum gate array is a set of quantum gates with logical "wires" connecting their inputs and outputs. The input to the gate array, possibly along with extra work bits that are initially set to 0, is fed through a sequence of quantum gates. The values of the bits are observed after the last quantum gate, and these values are the output. This model is analogous to classical acyclic circuits in theoretical computer science, and was previously studied by Yao [1993]. As in the classical case, in order to compare quantum gate arrays with quantum Turing machines, we need to make the gate arrays a *uniform* complexity class. In other words, because we use a different gate array for each size of input, we need to keep the designer of the gate arrays from hiding noncomputable (or hard to compute) information in the arrangement of the gates. To make quantum gate arrays uniform, we must add two requirements to the definition of gate arrays. The first is the standard uniformity requirement that the design of the gate array be produced by a polynomial-time (classical) computation. The second uniformity requirement should be a standard part of the definition of analog complexity classes; however, since analog complexity classes have not been as widely studied, this requirement is not well known. The requirement is that the entries in the unitary matrices describing the gates must be computable numbers. Specifically, the first $\log n$ bits of each entry should be classically computable in time polynomial in n [Solovay 1995]. This keeps noncomputable (or hard to compute) information from being hidden in the bits of the amplitudes of the quantum gates.

3. Reversible logic and modular exponentiation. The definition of quantum gate arrays gives rise to completely reversible computation. That is, knowing the quantum state on the wires leading out of a gate tells uniquely what the quantum state must have been on the wires leading into that gate. This is a reflection of the fact that, despite the macroscopic arrow of time, the laws of physics appear to be completely reversible. This would seem to imply that anything built with the laws of physics must be completely reversible; however, classical computers get around this by dissipating energy and thus making their computations thermodynamically irreversible. This appears impossible to do for quantum computers because superpositions of quantum states need to be maintained throughout the computation. Thus, quantum computers necessarily have to use reversible computation. This imposes

TABLE 3.1
Truth tables for Toffoli and Fredkin gates.

Toffoli gate

INPUT			OUTPUT		
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Fredkin gate

INPUT			OUTPUT		
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1

extra costs when doing classical computations on a quantum computer, which can be necessary in subroutines of quantum computations.

Because of the reversibility of quantum computation, a deterministic computation is performable on a quantum computer only if it is reversible. Luckily, it has already been shown that any deterministic computation can be made reversible [Lecerf 1963; Bennett 1973]. In fact, reversible classical gate arrays (or reversible acyclic circuits) have been studied. Much like the result that any classical computation can be done using NAND gates, there are also universal gates for reversible computation. Two of these are Toffoli gates [Toffoli 1980] and Fredkin gates [Fredkin and Toffoli 1982]; these are illustrated in Table 3.1.

The Toffoli gate is just a doubly controlled NOT, i.e., the last bit is negated if and only if the first two bits are 1. In a Toffoli gate, if the third input bit is set to 1, then the third output bit is the NAND of the first two input bits. Since NAND is a universal gate for classical gate arrays, this shows that the Toffoli gate is universal. In a Fredkin gate, the last two bits are swapped if the first bit is 0, and left untouched if the first bit is 1. For a Fredkin gate, if the third input bit is set to 0, the second output bit is the AND of the first two input bits; and if the last two input bits are set to 0 and 1 respectively, the second output bit is the NOT of the first input bit. Thus, both AND and NOT gates are realizable using Fredkin gates, showing that the Fredkin gate is universal.

From results on reversible computation [Lecerf 1963, Bennett 1973], we can efficiently compute any polynomial time function $F(x)$ as long as we keep the input x in the computer. We do this by adapting the method for computing the function F nonreversibly. These results can easily be extended to work for gate arrays [Toffoli 1980; Fredkin and Toffoli 1982]. When AND, OR, or NOT gates are changed to Fredkin or Toffoli gates, one obtains both additional input bits, which must be preset to specified values, and additional output bits, which contain the information needed to reverse the computation. While the additional input bits do not present difficulties in designing quantum computers, the additional output bits do, because unless they are all reset to 0, they will affect the interference patterns in quantum computation. Bennett's method for resetting these bits to 0 is shown in the top half of Table 3.2. A nonreversible gate array may thus be turned into a reversible gate array as follows. First, duplicate the input bits as many times as necessary (since each input bit could be used more than once by the gate array). Next, keeping one copy of the input around, use Toffoli and Fredkin gates to simulate nonreversible gates, putting the extra output bits into the RECORD register. These extra output bits preserve enough of a record of the operations to enable the computation of the gate array to be

TABLE 3.2
Bennett's method for making a computation reversible.

INPUT	- - - - - -	- - - - - -	- - - - - -
INPUT	OUTPUT	RECORD(F)	- - - - - -
INPUT	OUTPUT	RECORD(F)	OUTPUT
INPUT	- - - - - -	- - - - - -	OUTPUT
INPUT	INPUT	RECORD(F^{-1})	OUTPUT
- - - - - -	INPUT	RECORD(F^{-1})	OUTPUT
- - - - - -	- - - - - -	- - - - - -	OUTPUT

reversed. Once the output $F(x)$ has been computed, copy it into a register that has been preset to zero, and then undo the computation to erase both the first OUTPUT register and the RECORD register.

To erase x and replace it with $F(x)$, in addition to a polynomial-time algorithm for F, we also need a polynomial-time algorithm for computing x from $F(x)$; i.e., we need that F is one-to-one and that both F and F^{-1} are polynomial-time computable. The method for this computation is given in the whole of Table 3.2. There are two stages to this computation. The first is the same as before, taking x to $(x, F(x))$. For the second stage, shown in the bottom half of Table 3.2, note that if we have a method to compute F^{-1} nonreversibly in polynomial time, we can use the same technique to reversibly map $F(x)$ to $(F(x), F^{-1}(F(x))) = (F(x), x)$. However, since this is a reversible computation, we can reverse it to go from $(x, F(x))$ to $F(x)$. Put together, these two stages take x to $F(x)$.

The above discussion shows that computations can be made reversible for only a constant factor cost in time, but the above method uses as much space as it does time. If the classical computation requires much less space than time, then making it reversible in this manner will result in a large increase in the space required. There are methods that do not use as much space, but use more time, to make computations reversible [Bennett 1989, Levine and Sherman 1990]. While there is no general method that does not cause an increase in either space or time, specific algorithms can sometimes be made reversible without paying a large penalty in either space or time; at the end of this section we will show how to do this for modular exponentiation, which is a subroutine necessary for quantum factoring.

The bottleneck in the quantum factoring algorithm—i.e., the piece of the factoring algorithm that consumes the most time and space—is modular exponentiation. The modular exponentiation problem is, given n, x, and r, find $x^r \pmod n$. The best classical method for doing this is to repeatedly square of $x \pmod n$ to get $x^{2^i} \pmod n$ for $i \leq \log_2 r$, and then multiply a subset of these powers $(\text{mod } n)$ to get $x^r \pmod n$. If we are working with l-bit numbers, this requires $O(l)$ squarings and multiplications of l-bit numbers $(\text{mod } n)$. Asymptotically, the best classical result for gate arrays for multiplication is the Schönhage–Strassen algorithm [Schönhage and Strassen 1971, Knuth 1981, Schönhage 1982]. This gives a gate array for integer multiplication that uses $O(l \log l \log\log l)$ gates to multiply two l-bit numbers. Thus, asymptotically, modular exponentiation requires $O(l^2 \log l \log\log l)$ time. Making this reversible would naïvely cost the same amount in space; however, one can reuse the space used in the repeated squaring part of the algorithm, and thus reduce the amount of space needed to essentially that required for multiplying two l-bit numbers; one simple method for reducing this space (although not the most versatile one) will be given later in this section. Thus, modular exponentiation can be done in $O(l^2 \log l \log\log l)$ time and $O(l \log l \log\log l)$ space.

While the Schönhage–Strassen algorithm is the best multiplication algorithm discovered to date for large l, it does not scale well for small l. For small numbers, the best gate arrays for multiplication essentially use elementary-school longhand multiplication in binary. This method requires $O(l^2)$ time to multiply two l-bit numbers, and thus modular exponentiation requires $O(l^3)$ time with this method. These gate arrays can be made reversible, however, using only $O(l)$ space.

We now give the method for constructing a reversible gate array that takes only $O(l)$ space and $O(l^3)$ time to compute $(a, x^a \pmod n)$ from a, where a, x, and n are l-bit numbers and x and n are relatively prime. This case, where x and n are relatively prime, is sufficient for our factoring and discrete logarithm algorithms. A detailed analysis of essentially this method, giving an exact number of quantum gates sufficient for factoring, was performed by Beckman et al. [1996].

The basic building block used is a gate array that takes b as input and outputs $b + c \pmod n$. Note that here b is the gate array's input but c and n are built into the structure of the gate array. Since addition $(\bmod n)$ is computable in $O(\log n)$ time classically, this reversible gate array can be made with only $O(\log n)$ gates and $O(\log n)$ work bits using the techniques explained earlier in this section.

The technique we use for computing $x^a \pmod n$ is essentially the same as the classical method. First, by repeated squaring we compute $x^{2^i} \pmod n$ for all $i < l$. Then, to obtain $x^a \pmod n$ we multiply the powers $x^{2^i} \pmod n$, where 2^i appears in the binary expansion of a. In our algorithm for factoring n, we need only to compute $x^a \pmod n$, where a is in a superposition of states but x is some fixed integer. This makes things much easier, because we can use a reversible gate array where a is input but where x and n are built into the structure of the gate array. Thus, we can use the algorithm described by the following pseudocode; here a_i represents the ith bit of a in binary, where the bits are indexed from right to left and the rightmost bit of a is a_0.

```
power := 1
for i = 0 to l−1
    if ( a_i == 1 ) then
        power := power * x^(2^i)   (mod n)
    endif
endfor
```

The variable a is left unchanged by the code and $x^a \pmod n$ is output as the variable *power*. Thus, this code takes the pair of values $(a, 1)$ to $(a, x^a \pmod n)$.

This pseudocode can easily be turned into a gate array; the only hard part of this is the fourth line, where we multiply the variable *power* by $x^{2^i} \pmod n$; to do this we need to use a fairly complicated gate array as a subroutine. Recall that $x^{2^i} \pmod n$ can be computed classically and then built into the structure of the gate array. Thus, to implement this line, we need a reversible gate array that takes b as input and gives $bc \pmod n$ as output, where the structure of the gate array can depend on c and n. Of course, this step can only be reversible if $\gcd(c, n) = 1$—i.e., if c and n have no common factors—as otherwise two distinct values of b will be mapped to the same value of $bc \pmod n$. Fortunately, x and n being relatively prime in modular exponentiation implies that c and n are relatively prime in this subroutine.

We will show how to build this gate array in two stages. The first stage is directly analogous to exponentiation by repeated multiplication; we obtain multiplication from repeated addition $(\bmod n)$. Pseudocode for this stage is as follows.

```
result := 0
for i = 0 to l−1
    if ( b_i == 1 ) then
        result := result + 2^i c  (mod n)
    endif
endfor
```

Again, $2^i c \pmod n$ can be precomputed and built into the structure of the gate array.

The above pseudocode takes b as input and gives $(b, bc \pmod n)$ as output. To get the desired result, we now need to erase b. Recall that $\gcd(c, n) = 1$, so there is a $c^{-1} \pmod n$ with $c\,c^{-1} \equiv 1 \pmod n$. Multiplication by this c^{-1} could be used to reversibly take $bc \pmod n$ to $(bc \pmod n, bcc^{-1} \pmod n) = (bc \pmod n, b)$. This is just the reverse of the operation we want, and since we are working with reversible computing, we can turn this operation around to erase b. The pseudocode for this follows.

```
for i = 0 to l−1
    if ( result_i == 1 ) then
        b := b − 2^i c^{-1}  (mod n)
    endif
endfor
```

As before, $result_i$ is the ith bit of *result*.

Note that at this stage of the computation, b should be 0. However, we did not set b directly to zero, as this would not have been a reversible operation and thus impossible on a quantum computer, but instead we did a relatively complicated sequence of operations which ended with $b = 0$ and which in fact depended on multiplication being a group (mod n). At this point, then, we could do something somewhat sneaky: we could measure b to see if it actually is 0. If it is not, we know that there has been an error somewhere in the quantum computation, i.e., that the results are worthless and we should stop the computer and start over again. However, if we do find that b is 0, then we know (because we just observed it) that it is now exactly 0. This measurement thus may bring the quantum computation back on track in that any amplitude that b had for being nonzero has been eliminated. Further, because the probability that we observe a state is proportional to the square of the amplitude of that state, doing the modular exponentiation and measuring b every time that we know it should be 0 may result in a higher probability of overall success than the same computation done without the repeated measurements of b. This is the *quantum watchdog* (or *quantum Zeno*) effect [Peres 1993], and whether it is applicable in this setting depends on the error model for our quantum circuits. The argument above does not actually show that repeated measurement of b is indeed beneficial, because there is a cost (in time, if nothing else) of measuring b. Before this is implemented, then, it should be checked with analysis or experiment that the benefit of such measurements exceeds their cost. In general, I believe that partial measurements such as this one are a promising way of trying to stabilize quantum computations.

Currently, Schönhage–Strassen is the algorithm of choice for multiplying very large numbers, and longhand multiplication is the algorithm of choice for small numbers. There are also multiplication algorithms which have asymptotic efficiencies between these two algorithms and which are superior for intermediate length numbers [Karatsuba and Ofman 1962; Knuth 1981; Schönhage, Grotefeld, and Vetter 1994]. It is not clear which algorithms are best for which size numbers. While this is known to

some extent for classical computation [Schönhage, Grotefeld, and Vetter 1994], using data on which algorithms work better on classical computers could be misleading for two reasons. First, classical computers need not be reversible, and the cost of making an algorithm reversible depends on the algorithm. Second, existing computers generally have multiplication for 32- or 64-bit numbers built into their hardware, and this tends to increase the optimal changeover points. To further confuse matters, some multiplication algorithms can take better advantage of hardwired multiplication than others. In order to program quantum computers most efficiently, work thus needs to be done on the best way of implementing elementary arithmetic operations on quantum computers. One tantalizing fact is that the Schönhage–Strassen fast multiplication algorithm uses the fast Fourier transform, which is also the basis for all the fast algorithms on quantum computers discovered to date. It is thus tempting to speculate that integer multiplication itself might be speeded up by a quantum algorithm; if possible, this would result in a somewhat faster asymptotic bound for factoring on a quantum computer, and indeed could even make breaking RSA on a quantum computer asymptotically faster than encrypting with RSA on a classical computer.

4. Quantum Fourier transforms. Since quantum computation deals with unitary transformations, it is helpful to know how to build certain useful unitary transformations. In this section we give a technique for constructing in polynomial time on quantum computers one particular unitary transformation, which is essentially a discrete Fourier transform. This transformation will be given as a matrix, with both rows and columns indexed by states. These states correspond to binary representations of integers on the computer; in particular, the rows and columns will be indexed beginning with 0 unless otherwise specified.

This transformation is as follows. Consider a number a with $0 \le a < q$ for some q. We will perform the transformation that takes the state $|a\rangle$ to the state

$$\frac{1}{q^{1/2}} \sum_{c=0}^{q-1} |c\rangle \exp(2\pi i ac/q). \tag{4.1}$$

That is, we apply the unitary matrix whose (a, c) entry is $\frac{1}{q^{1/2}} \exp(2\pi i ac/q)$. This Fourier transform is at the heart of our algorithms, and we call this matrix A_q.

In our factoring and discrete logarithm algorithms, we will use A_q for q of exponential size (i.e., the number of bits of q grows polynomially with the length of our input). We must thus show how this transformation can be done in time polynomial in the number of bits of q. In this paper, we give a simple construction for A_q when q is a power of 2 that was discovered independently by Coppersmith [1994] and Deutsch [see Ekert and Jozsa 1996]. This construction is essentially the standard fast Fourier transform (FFT) algorithm [Knuth 1981] adapted for a quantum computer; the following description of it follows that of Ekert and Jozsa [1996]. In the earlier version of this paper [Shor 1994], we gave a construction for A_q when q was in the special class of smooth numbers having only small prime power factors. In fact, Cleve [1994] has shown how to construct A_q for all smooth numbers q whose prime factors are at most $O(\log n)$.

Take $q = 2^l$, and let us represent an integer a in binary as $|a_{l-1}a_{l-2}\dots a_0\rangle$. For the quantum Fourier transform A_q, we need only to use two types of quantum gates.

These gates are R_j, which operates on the jth bit of the quantum computer:

$$R_j = \begin{array}{c|cc} & |0\rangle & |1\rangle \\ |0\rangle & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ |1\rangle & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{array}, \tag{4.2}$$

and $S_{j,k}$, which operates on the bits in positions j and k with $j < k$:

$$S_{j,k} = \begin{array}{c|cccc} & |00\rangle & |01\rangle & |10\rangle & |11\rangle \\ |00\rangle & 1 & 0 & 0 & 0 \\ |01\rangle & 0 & 1 & 0 & 0 \\ |10\rangle & 0 & 0 & 1 & 0 \\ |11\rangle & 0 & 0 & 0 & e^{i\theta_{k-j}} \end{array}, \tag{4.3}$$

where $\theta_{k-j} = \pi/2^{k-j}$. To perform a quantum Fourier transform, we apply the matrices in the order (from left to right)

$$R_{l-1}\, S_{l-2,l-1}\, R_{l-2}\, S_{l-3,l-1}\, S_{l-3,l-2}\, R_{l-3} \ldots R_1\, S_{0,l-1}\, S_{0,l-2} \ldots S_{0,2}\, S_{0,1}\, R_0; \tag{4.4}$$

that is, we apply the gates R_j in reverse order from R_{l-1} to R_0, and between R_{j+1} and R_j we apply all the gates $S_{j,k}$ where $k > j$. For example, on 3 bits, the matrices would be applied in the order $R_2 S_{1,2} R_1 S_{0,2} S_{0,1} R_0$. To take the Fourier transform A_q when $q = 2^l$, we thus need to use $l(l-1)/2$ quantum gates.

Applying this sequence of transformations will result in a quantum state $\frac{1}{q^{1/2}} \sum_b \exp(2\pi iac/q)|b\rangle$, where b is the bit-reversal of c, i.e., the binary number obtained by reading the bits of c from right to left. Thus, to obtain the actual quantum Fourier transform, we need either to do further computation to reverse the bits of $|b\rangle$ to obtain $|c\rangle$, or to leave these bits in place and read them in reverse order; either alternative is easy to implement.

To show that this operation actually performs a quantum Fourier transform, consider the amplitude of going from $|a\rangle = |a_{l-1} \ldots a_0\rangle$ to $|b\rangle = |b_{l-1} \ldots b_0\rangle$. First, the factors of $1/\sqrt{2}$ in the R matrices multiply to produce a factor of $1/q^{1/2}$ overall; thus we need only worry about the $\exp(2\pi iac/q)$ phase factor in the expression (4.1). The matrices $S_{j,k}$ do not change the values of any bits, but merely change their phases. There is thus only one way to switch the jth bit from a_j to b_j, and that is to use the appropriate entry in the matrix R_j. This entry adds π to the phase if the bits a_j and b_j are both 1, and leaves it unchanged otherwise. Further, the matrix $S_{j,k}$ adds $\pi/2^{k-j}$ to the phase if a_j and b_k are both 1 and leaves it unchanged otherwise. Thus, the phase on the path from $|a\rangle$ to $|b\rangle$ is

$$\sum_{0 \le j < l} \pi a_j b_j + \sum_{0 \le j < k < l} \frac{\pi}{2^{k-j}} a_j b_k. \tag{4.5}$$

This expression can be rewritten as

$$\sum_{0 \le j \le k < l} \frac{\pi}{2^{k-j}} a_j b_k. \tag{4.6}$$

Since c is the bit-reversal of b, this expression can be further rewritten as

$$\sum_{0 \le j \le k < l} \frac{\pi}{2^{k-j}} a_j c_{l-1-k}. \tag{4.7}$$

Making the substitution $l-k-1$ for k in this sum, we obtain

$$\sum_{0\le j+k<l} 2\pi \frac{2^j 2^k}{2^l} a_j c_k. \tag{4.8}$$

Now, since adding multiples of 2π do not affect the phase, we obtain the same phase if we sum over all j and k less than l, obtaining

$$\sum_{j,k=0}^{l-1} 2\pi \frac{2^j 2^k}{2^l} a_j c_k = \frac{2\pi}{2^l} \sum_{j=0}^{l-1} 2^j a_j \sum_{k=0}^{l-1} 2^k c_k, \tag{4.9}$$

where the last equality follows from the distributive law of multiplication. Now, $q = 2^l$ and

$$a = \sum_{j=0}^{l-1} 2^j a_j, \qquad c = \sum_{k=0}^{l-1} 2^k c_k, \tag{4.10}$$

so the expression (4.9) is equal to $2\pi ac/q$, which is the phase for the amplitude $|a\rangle \to |c\rangle$ in the transformation (4.1).

When $k-j$ is large in the gate $S_{j,k}$ in (4.3), we are multiplying by a very small phase factor. This would be very difficult to do accurately physically, and thus it would be somewhat disturbing if this were necessary for quantum computation. In fact, Coppersmith [1994] has shown that one can use an approximate Fourier transform that ignores these tiny phase factors but which approximates the Fourier transform closely enough that it can also be used for factoring. In fact, this technique reduces the number of quantum gates needed for the (approximate) Fourier transform considerably, as it leaves out most of the gates $S_{j,k}$.

Recently, Griffiths and Niu [1996] have shown that this Fourier transform can be carried out using only one-bit gates and measurements of single bits. Both of these operations are potentially easier to implement in a physical system than two-bit gates. The use of two-bit gates, however, is still required during the modular exponentiation step of the factoring and discrete logarithm algorithms.

5. Prime factorization. It has been known since before Euclid that every integer n is uniquely decomposable into a product of primes. For nearly as long, mathematicians have been interested in the question of how to factor a number into this product of primes. It was only in the 1970s, however, that researchers applied the paradigms of theoretical computer science to number theory, and looked at the asymptotic running times of factoring algorithms [Adleman 1994]. This has resulted in a great improvement in the efficiency of factoring algorithms. Currently the best factoring algorithm, both asymptotically and in practice, is the number field sieve [Lenstra et al. 1990, Lenstra and Lenstra 1993], which in order to factor an integer n takes asymptotic running time $\exp(c(\log n)^{1/3}(\log\log n)^{2/3})$ for some constant c. Since the input n is only $\log n$ bits in length, this algorithm is an exponential-time algorithm. Our quantum factoring algorithm takes asymptotically $O((\log n)^2(\log\log n)(\log\log\log n))$ steps on a quantum computer, along with a polynomial (in $\log n$) amount of post-processing time on a classical computer that is used to convert the output of the quantum computer to factors of n. While this post-processing could in principle be done on a quantum computer, there is no reason not to use a classical computer for this step.

Instead of giving a quantum computer algorithm for factoring n directly, we give a quantum computer algorithm for finding the order r of an element x in the multiplicative group (mod n); that is, the least integer r such that $x^r \equiv 1 \pmod{n}$. It is known that using randomization, factorization can be reduced to finding the order of an element [Miller 1976]; we now briefly give this reduction.

To find a factor of an odd number n, given a method for computing the order r of x, choose a random $x \pmod{n}$, find its order r, and compute $\gcd(x^{r/2} - 1, n)$. Here, $\gcd(a, b)$ is the greatest common divisor of a and b, i.e., the largest integer that divides both a and b. The Euclidean algorithm [Knuth 1981] can be used to compute $\gcd(a, b)$ in polynomial time. Since $(x^{r/2} - 1)(x^{r/2} + 1) = x^r - 1 \equiv 0 \pmod{n}$, the numbers $\gcd(x^{r/2} + 1, n)$ and $\gcd(x^{r/2} - 1, n)$ will be two factors of n. This procedure fails only if r is odd, in which case $r/2$ is not integral, or if $x^{r/2} \equiv -1 \pmod{n}$, in which case the procedure yields the trivial factors 1 and n. Using this criterion, it can be shown that this procedure, when applied to a random $x \pmod{n}$, yields a nontrivial factor of n with probability at least $1 - 1/2^{k-1}$, where k is the number of distinct odd prime factors of n. A brief sketch of the proof of this result follows.

Suppose that $n = \prod_{i=1}^{k} p_i^{\alpha_i}$ is the prime factorization of n. Let r_i be the order of $x \pmod{p_i^{\alpha_i}}$. Then r is the least common multiple of all the r_i. Consider the largest power of 2 dividing each r_i. The algorithm only fails if all of these powers of 2 agree: if they are all 1, then r is odd and $r/2$ does not exist; if they are all equal and larger than 1, then $x^{r/2} \equiv -1 \pmod{p_i^{\alpha_i}}$ for every i, so $x^{r/2} \equiv -1 \pmod{n}$. By the Chinese remainder theorem [Knuth 1981; Hardy and Wright 1979, Theorem 121], choosing an $x \pmod{n}$ at random is the same as choosing for each i a number $x_i \pmod{p_i^{\alpha_i}}$ at random, where $x \equiv x_i \pmod{p_i^{\alpha_i}}$. The multiplicative group $(\text{mod } p^{\alpha})$ for any odd prime power p^{α} is cyclic [Knuth 1981], so for the odd prime power $p_i^{\alpha_i}$, the probability is at most $1/2$ of choosing an x_i having a particular power of 2 as the largest divisor of its order r_i. Thus each of these powers of 2 has at most a 50% probability of agreeing with the previous ones, so all k of them agree with probability at most $1/2^{k-1}$. There is thus at least a $1 - 1/2^{k-1}$ probability that the x we choose is good. This argument shows the scheme will work as long as n is odd and not a prime power; finding a factor of even numbers and of prime powers can be done efficiently with classical methods.

We now describe the algorithm for finding the order of $x \pmod{n}$ on a quantum computer. This algorithm will use two quantum registers which hold integers represented in binary. There will also be some amount of workspace. This workspace gets reset to 0 after each subroutine of our algorithm, so we will not include it when we write down the state of our machine.

Given x and n, to find the order of x, i.e., the least r such that $x^r \equiv 1 \pmod{n}$, we do the following. First, we find q, the power of 2 with $n^2 \le q < 2n^2$. We will not include n, x, or q when we write down the state of our machine, because we never change these values. In a quantum gate array we need not even keep these values in memory, as they can be built into the structure of the gate array.

Next, we put the first register in the uniform superposition of states representing numbers $a \pmod{q}$. This leaves our machine in state

$$\frac{1}{q^{1/2}} \sum_{a=0}^{q-1} |a\rangle|0\rangle. \tag{5.1}$$

This step is relatively easy, since all it entails is putting each bit in the first register into the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

Next, we compute $x^a \pmod{n}$ in the second register as described in section 3.

Since we keep a in the first register this can be done reversibly. This leaves our machine in the state

$$\frac{1}{q^{1/2}} \sum_{a=0}^{q-1} |a\rangle |x^a \pmod{n}\rangle. \tag{5.2}$$

We then perform our Fourier transform A_q on the first register, as described in section 4, mapping $|a\rangle$ to

$$\frac{1}{q^{1/2}} \sum_{c=0}^{q-1} \exp(2\pi i ac/q)|c\rangle. \tag{5.3}$$

That is, we apply the unitary matrix with the (a, c) entry equal to $\frac{1}{q^{1/2}} \exp(2\pi iac/q)$. This leaves our machine in state

$$\frac{1}{q} \sum_{a=0}^{q-1} \sum_{c=0}^{q-1} \exp(2\pi iac/q)|c\rangle |x^a \pmod{n}\rangle. \tag{5.4}$$

Finally, we observe the machine. It would be sufficient to observe solely the value of $|c\rangle$ in the first register, but for clarity we will assume that we observe both $|c\rangle$ and $|x^a \pmod{n}\rangle$. We now compute the probability that our machine ends in a particular state $|c, x^k \pmod{n}\rangle$, where we may assume $0 \le k < r$. Summing over all possible ways to reach the state $|c, x^k \pmod{n}\rangle$, we find that this probability is

$$\left| \frac{1}{q} \sum_{a:\, x^a \equiv x^k} \exp(2\pi iac/q) \right|^2, \tag{5.5}$$

where the sum is over all a, $0 \le a < q$, such that $x^a \equiv x^k \pmod{n}$. Because the order of x is r, this sum is over all a satisfying $a \equiv k \pmod{r}$. Writing $a = br + k$, we find that the above probability is

$$\left| \frac{1}{q} \sum_{b=0}^{\lfloor (q-k-1)/r \rfloor} \exp(2\pi i(br + k)c/q) \right|^2. \tag{5.6}$$

We can ignore the term of $\exp(2\pi ikc/q)$, as it can be factored out of the sum and has magnitude 1. We can also replace rc with $\{rc\}_q$, where $\{rc\}_q$ is the residue which is congruent to $rc \pmod{q}$ and is in the range $-q/2 < \{rc\}_q \le q/2$. This leaves us with the expression

$$\left| \frac{1}{q} \sum_{b=0}^{\lfloor (q-k-1)/r \rfloor} \exp(2\pi ib\{rc\}_q/q) \right|^2. \tag{5.7}$$

We will now show that if $\{rc\}_q$ is small enough, all the amplitudes in this sum will be in nearly the same direction, i.e., have close to the same phase, and thus make the sum large. Turning the sum into an integral, we obtain

$$\frac{1}{q} \int_0^{\lfloor (q-k-1)/r \rfloor} \exp(2\pi ib\{rc\}_q/q)db + O\left(\frac{\lfloor (q-k-1)/r \rfloor}{q} \left(\exp(2\pi i\{rc\}_q/q) - 1 \right) \right). \tag{5.8}$$

If $|\{rc\}_q| \leq r/2$, the error term in the above expression is easily seen to be bounded by $O(1/q)$. We now show that if $|\{rc\}_q| \leq r/2$, the above integral is large, so the probability of obtaining a state $|c, x^k \pmod n\rangle$ is large. Note that this condition depends only on c and is independent of k. Substituting $u = rb/q$ in the above integral, we get

$$\frac{1}{r}\int_0^{\lfloor (q-k-1)/r \rfloor r/q} \exp\left(2\pi i \frac{\{rc\}_q}{r} u\right) du. \tag{5.9}$$

Since $k < r$, approximating the upper limit of integration by 1 results in only a $O(1/q)$ error in the above expression. If we do this, we obtain the integral

$$\frac{1}{r}\int_0^{1} \exp\left(2\pi i \frac{\{rc\}_q}{r} u\right) du. \tag{5.10}$$

Letting $\{rc\}_q/r$ vary between $-\frac{1}{2}$ and $\frac{1}{2}$, the absolute magnitude of the integral (5.10) is easily seen to be minimized when $\{rc\}_q/r = \pm\frac{1}{2}$, in which case the absolute value of expression (5.10) is $2/(\pi r)$. The square of this quantity is a lower bound on the probability that we see any particular state $|c, x^k \pmod n\rangle$ with $\{rc\}_q \leq r/2$; this probability is thus asymptotically bounded below by $4/(\pi^2 r^2)$, and so is at least $1/3r^2$ for sufficiently large n.

The probability of seeing a given state $|c, x^k \pmod n\rangle$ will thus be at least $1/3r^2$ if

$$\frac{-r}{2} \leq \{rc\}_q \leq \frac{r}{2}, \tag{5.11}$$

i.e., if there is a d such that

$$\frac{-r}{2} \leq rc - dq \leq \frac{r}{2}. \tag{5.12}$$

Dividing by rq and rearranging the terms give

$$\left|\frac{c}{q} - \frac{d}{r}\right| \leq \frac{1}{2q}. \tag{5.13}$$

We know c and q. Because $q > n^2$, there is at most one fraction d/r with $r < n$ that satisfies the above inequality. Thus, we can obtain the fraction d/r in lowest terms by rounding c/q to the nearest fraction having a denominator smaller than n. This fraction can be found in polynomial time by using a continued fraction expansion of c/q, which finds all the best approximations of c/q by fractions [Knuth 1981; Hardy and Wright 1979, Chapter X].

The exact probabilities as given by (5.7) for an example case with $r = 10$ and $q = 256$ are plotted in Fig. 5.1. For example, the value $r = 10$ could occur when factoring 33 if x were chosen to be 5. Here q is taken smaller than 33^2 so as to make the values of c in the plot distinguishable; this does not change the functional structure of $\mathrm{P}(c)$. Note that with high probability the observed value of c is near an integral multiple of $q/r = 256/10$.

If we have the fraction d/r in lowest terms, and if d happens to be relatively prime to r, this will give us r. We will now count the number of states $|c, x^k \pmod n\rangle$ which enable us to compute r in this way. There are $\phi(r)$ possible values of d relatively prime to r, where ϕ is Euler's totient function [Knuth 1981; Hardy and Wright 1979,

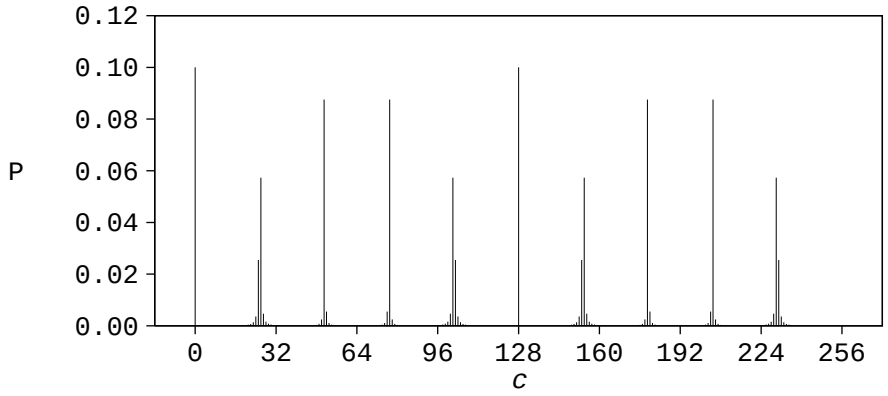


FIG. 5.1. *The probability* P *of observing values of* c *between* 0 *and* 255*, given* $q = 256$ *and* $r = 10$.

section 5.5]. Each of these fractions d/r is close to one fraction c/q with $|c/q - d/r| \leq 1/2q$. There are also r possible values for x^k, since r is the order of x. Thus, there are $r\phi(r)$ states $|c, x^k \pmod n\rangle$ which would enable us to obtain r. Since each of these states occurs with probability at least $1/3r^2$, we obtain r with probability at least $\phi(r)/3r$. Using the theorem that $\phi(r)/r > \delta/\log\log r$ for some constant δ [Hardy and Wright 1979, Theorem 328], this shows that we find r at least a $\delta/\log\log r$ fraction of the time, so by repeating this experiment only $O(\log\log r)$ times, we are assured of a high probability of success.

In practice, assuming that quantum computation is more expensive than classical computation, it would be worthwhile to alter the above algorithm so as to perform less quantum computation and more postprocessing. First, if the observed state is $|c\rangle$, it would be wise to also try numbers close to c such as $c \pm 1, c \pm 2, \ldots$, since these also have a reasonable chance of being close to a fraction qd/r. Second, if $c/q \approx d/r$ where d and r have a common factor, this factor is likely to be small. Thus, if the observed value of c/q is rounded off to d'/r' in lowest terms, for a candidate r one should consider not only r' but also its small multiples $2r', 3r', \ldots$, to see if these are the actual order of x. Although the first technique will only reduce the expected number of trials required to find r by a constant factor, the second technique will reduce the expected number of trials for the hardest n from $O(\log\log n)$ to $O(1)$ if the first $(\log n)^{1+\epsilon}$ multiples of r' are considered [Odylzko 1995]. A third technique, if two candidates for r—say, r_1 and r_2—have been found, is to test the least common multiple of r_1 and r_2 as a candidate r. This third technique is also able to reduce the expected number of trials to a constant [Knill 1995] and will work in some cases where the first two techniques fail.

Note that in this algorithm for determining the order of an element, we did not use many of the properties of multiplication (mod n). In fact, if we have a permutation f mapping the set $\{0, 1, 2, \ldots, n-1\}$ into itself such that its kth iterate, $f^{(k)}(a)$, is computable in time polynomial in $\log n$ and $\log k$, the same algorithm will be able to find the order of an element a under f, i.e., the minimum r such that $f^{(r)}(a) = a$.

6. Discrete logarithms. For every prime p, the multiplicative group (mod p) is cyclic, that is, there are generators g such that $1, g, g^2, \ldots, g^{p-2}$ comprise all the nonzero residues (mod p) [Hardy and Wright 1979, Theorem 111; Knuth 1981]. Suppose that we are given a prime p and such a generator g. The *discrete logarithm* of

a number x with respect to p and g is the integer r with $0 \leq r < p-1$ such that $g^r \equiv x \pmod{p}$. The fastest algorithm known for finding discrete logarithms modulo arbitrary primes p is Gordon's [1993] adaptation of the number field sieve, which runs in time $\exp(O(\log p)^{1/3}(\log\log p)^{2/3})$. We show how to find discrete logarithms on a quantum computer using two modular exponentiations and two quantum Fourier transforms.

This algorithm will use three quantum registers. We first find q a power of 2 such that q is close to p, i.e., with $p < q < 2p$. Next, we put the first two registers in our quantum computer in the uniform superposition of all $|a\rangle$ and $|b\rangle \pmod{p-1}$. One way to do this in quantum polynomial time is to put the register in a uniform superposition of all the numbers from 0 to $q-1$, use quantum computation to test whether the number is less than p, and restart the algorithm if the results of this test are unfavorable. We next compute $g^a x^{-b} \pmod{p}$ in the third register. This leaves our machine in the state

$$\frac{1}{p-1}\sum_{a=0}^{p-2}\sum_{b=0}^{p-2}|a,b,g^a x^{-b} \pmod{p}\rangle. \tag{6.1}$$

As before, we use the Fourier transform A_q to send $|a\rangle \to |c\rangle$ and $|b\rangle \to |d\rangle$ with probability amplitude $\frac{1}{q}\exp(2\pi i(ac+bd)/q)$. That is, we take the state $|a,b\rangle$ to the state

$$\frac{1}{q}\sum_{c=0}^{q-1}\sum_{d=0}^{q-1}\exp\left(\frac{2\pi i}{q}(ac+bd)\right)|c,d\rangle. \tag{6.2}$$

This leaves our quantum computer in the state

$$\frac{1}{(p-1)q}\sum_{a,b=0}^{p-2}\sum_{c,d=0}^{q-1}\exp\left(\frac{2\pi i}{q}(ac+bd)\right)|c,d,g^a x^{-b} \pmod{p}\rangle. \tag{6.3}$$

Finally, we observe the state of the quantum computer.

The probability of observing a state $|c,d,y\rangle$ with $y \equiv g^k \pmod{p}$ is

$$\left|\frac{1}{(p-1)q}\sum_{\substack{a,b\\ a-rb\equiv k}}\exp\left(\frac{2\pi i}{q}(ac+bd)\right)\right|^2, \tag{6.4}$$

where the sum is over all (a,b) such that $a - rb \equiv k \pmod{p-1}$. Note that we now have two moduli to deal with, $p-1$ and q. While this makes keeping track of things more confusing, it does not pose serious problems. We now use the relation

$$a = br + k - (p-1)\left\lfloor\frac{br+k}{p-1}\right\rfloor \tag{6.5}$$

and substitute (6.5) in the expression (6.4) to obtain the amplitude on $|c,d,g^k \pmod{p}\rangle$, which is

$$\frac{1}{(p-1)q}\sum_{b=0}^{p-2}\exp\left(\frac{2\pi i}{q}\left(brc+kc+bd-c(p-1)\left\lfloor\frac{br+k}{p-1}\right\rfloor\right)\right). \tag{6.6}$$

The square of the absolute value of this amplitude is the probability of observing the state $|c, d, g^k \pmod p\rangle$. We will now analyze the expression (6.6). First, a factor of $\exp(2\pi ikc/q)$ can be taken out of all the terms and ignored, because it does not change the probability. Next, we split the exponent into two parts and factor out b to obtain

$$\frac{1}{(p-1)q}\sum_{b=0}^{p-2}\exp\left(\frac{2\pi i}{q}bT\right)\exp\left(\frac{2\pi i}{q}V\right), \tag{6.7}$$

where

$$T = rc + d - \frac{r}{p-1}\{c(p-1)\}_q, \tag{6.8}$$

and

$$V = \left(\frac{br}{p-1} - \left\lfloor\frac{br+k}{p-1}\right\rfloor\right)\{c(p-1)\}_q. \tag{6.9}$$

Here by $\{z\}_q$ we mean the residue of $z \pmod q$ with $-q/2 < \{z\}_q \le q/2$, as in (5.7).

We now classify possible outputs (observed states) of the quantum computer as "good" or "bad." We will show that if we get enough good outputs, then we will likely be able to deduce r, and that furthermore, the chance of getting a good output is constant. The idea is that if

$$\left|\{T\}_q\right| = \left|rc + d - \frac{r}{p-1}\{c(p-1)\}_q - jq\right| \le \frac{1}{2}, \tag{6.10}$$

where j is the closest integer to T/q, then as b varies between 0 and $p-2$, the phase of the first exponential term in (6.7) only varies over at most half of the unit circle. Further, if

$$|\{c(p-1)\}_q| \le q/12, \tag{6.11}$$

then $|V|$ is always at most $q/12$, so the phase of the second exponential term in (6.7) never is farther than $\exp(\pi i/6)$ from 1. If conditions (6.10) and (6.11) both hold, we will say that an output is good. We will show that if both conditions hold, then the contribution to the probability from the corresponding term is significant. Furthermore, both conditions will hold with constant probability, and a reasonable sample of c's for which condition (6.10) holds will allow us to deduce r.

We now give a lower bound on the probability of each good output, i.e., an output that satisfies conditions (6.10) and (6.11). We know that as b ranges from 0 to $p-2$, the phase of $\exp(2\pi ibT/q)$ ranges from 0 to $2\pi iW$ where

$$W = \frac{p-2}{q}\left(rc + d - \frac{r}{p-1}\{c(p-1)\}_q - jq\right). \tag{6.12}$$

It follows from (6.10) that $|W| \le (p-2)/(2q) \le 1/2$. Thus, the component of the amplitude of the first exponential in the summand of (6.7) in the direction

$$\exp(\pi iW) \tag{6.13}$$

is at least $\cos(2\pi(W/2 - Wb/(p-2)))$, where $2\pi(W/2 - Wb/(p-2))$ is between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. By condition (6.11), the phase can vary by at most $\pi i/6$ due to the second

exponential $\exp(2\pi iV/q)$. Applying this variation in the manner that minimizes the component in the direction (6.13), we get that the component in this direction is at least

$$\cos\left(2\pi\,|W/2 - Wb/(p-2)| + \frac{\pi}{6}\right). \tag{6.14}$$

Thus we get that the absolute value of the amplitude (6.7) is at least

$$\frac{1}{(p-1)q}\sum_{b=0}^{p-2}\cos\left(2\pi\,|W/2 - Wb/(p-2)| + \frac{\pi}{6}\right). \tag{6.15}$$

Replacing this sum with an integral, we get that the absolute value of this amplitude is at least

$$\frac{2}{q}\int_0^{1/2}\cos\left(\frac{\pi}{6} + 2\pi|W|u\right)du \;+\; O\left(\frac{W}{pq}\right). \tag{6.16}$$

From condition (6.10), $|W| \le \frac{1}{2}$, so the error term is $O(\frac{1}{pq})$. As W varies between $-\frac{1}{2}$ and $\frac{1}{2}$, the integral (6.16) is minimized when $|W| = \frac{1}{2}$. Thus, the probability of arriving at a state $|c, d, y\rangle$ that satisfies both conditions (6.10) and (6.11) is at least

$$\left(\frac{1}{q}\frac{2}{\pi}\int_{\pi/6}^{2\pi/3}\cos u\;du\right)^2, \tag{6.17}$$

or at least $.054/q^2 > 1/(20q^2)$.

We will now count the number of pairs (c, d) satisfying conditions (6.10) and (6.11). The number of pairs (c, d) such that (6.10) holds is exactly the number of possible c's, since for every c there is exactly one d such that (6.10) holds. Unless $\gcd(p-1, q)$ is large, the number of c's for which (6.11) holds is approximately $q/6$, and even if it is large, this number is at least $q/12$. Thus, there are at least $q/12$ pairs (c, d) satisfying both conditions. Multiplying by $p-1$, which is the number of possible y's, gives approximately $pq/12$ good states $|c, d, y\rangle$. Combining this calculation with the lower bound $1/(20q^2)$ on the probability of observing each good state gives us that the probability of observing some good state is at least $p/(240q)$, or at least $1/480$ (since $q < 2p$). Note that each good c has a probability of at least $(p-1)/(20q^2) \ge 1/(40q)$ of being observed, since there $p-1$ values of y and one value of d with which c can make a good state $|c, d, y\rangle$.

We now want to recover r from a pair c, d such that

$$-\frac{1}{2q} \le \frac{d}{q} + r\left(\frac{c(p-1) - \{c(p-1)\}_q}{(p-1)q}\right) \le \frac{1}{2q} \pmod 1, \tag{6.18}$$

where this equation was obtained from condition (6.10) by dividing by q. The first thing to notice is that the multiplier on r is a fraction with denominator $p-1$, since q evenly divides $c(p-1) - \{c(p-1)\}_q$. Thus, we need only round d/q off to the nearest multiple of $1/(p-1)$ and divide (mod $p-1$) by the integer

$$c' = \frac{c(p-1) - \{c(p-1)\}_q}{q} \tag{6.19}$$

to find a candidate r. To show that the quantum calculation need only be repeated a polynomial number of times to find the correct r requires only a few more details.

The problem is that we cannot divide by a number c' which is not relatively prime to $p-1$.

For the discrete log algorithm, we do not know that all possible values of c' are generated with reasonable likelihood; we only know this about one-twelfth of them. This additional difficulty makes the next step harder than the corresponding step in the algorithm for factoring. If we knew the remainder of r modulo all prime powers dividing $p-1$, we could use the Chinese remainder theorem to recover r in polynomial time. We will only be able to prove that we can find this remainder for primes larger than 18, but with a little extra work we will still be able to recover r.

Recall that each good (c,d) pair is generated with probability at least $1/(20q^2)$, and that at least one-twelfth of the possible c's are in a good (c,d) pair. From (6.19), it follows that these c's are mapped from c/q to $c'/(p-1)$ by rounding to the nearest integral multiple of $1/(p-1)$. Further, the good c's are exactly those in which c/q is close to $c'/(p-1)$. Thus, each good c corresponds with exactly one c'. We would like to show that for any prime power $p_i^{\alpha_i}$ dividing $p-1$, a random good c' is unlikely to contain p_i. If we are willing to accept a large constant for our algorithm, we can just ignore the prime powers under 18; if we know r modulo all prime powers over 18, we can try all possible residues for primes under 18 with only a (large) constant factor increase in running time. Because at least one-twelfth of the c's were in a good (c,d) pair, at least one-twelfth of the c''s are good. Thus, for a prime power $p_i^{\alpha_i}$, a random good c' is divisible by $p_i^{\alpha_i}$ with probability at most $12/p_i^{\alpha_i}$. If we have t good c''s, the probability of having a prime power over 18 that divides all of them is therefore at most

$$\sum_{18<p_i^{\alpha_i}\,|\,(p-1)} \left(\frac{12}{p_i^{\alpha_i}}\right)^t, \tag{6.20}$$

where $a|b$ means that a evenly divides b, so the sum is over all prime powers greater than 18 that divide $p-1$. This sum (over all integers >18) converges for $t=2$, and goes down by at least a factor of $2/3$ for each further increase of t by 1; thus for some constant t it is less than $1/2$.

Recall that each good c' is obtained with probability at least $1/(40q)$ from any experiment. Since there are $q/12$ good c''s, after $480t$ experiments, we are likely to obtain a sample of t good c''s chosen equally likely from all good c''s. Thus, we will be able to find a set of c''s such that all prime powers $p_i^{\alpha_i} > 20$ dividing $p-1$ are relatively prime to at least one of these c''s. To obtain a polynomial time algorithm, all one need do is try all possible sets of c''s of size t; in practice, one would use an algorithm to find sets of c''s with large common factors. This set gives the residue of r for all primes larger than 18. For each prime p_i less than 18, we have at most 18 possibilities for the residue modulo $p_i^{\alpha_i}$, where α_i is the exponent on prime p_i in the prime factorization of $p-1$. We can thus try all possibilities for residues modulo powers of primes less than 18: for each possibility we can calculate the corresponding r using the Chinese remainder theorem and then check to see whether it is the desired discrete logarithm.

If one were to actually program this algorithm there are many ways in which the efficiency could be increased over the efficiency shown in this paper. For example, the estimate for the number of good c''s is likely too low, especially since weaker conditions than (6.10) and (6.11) should suffice. This means that the number of times the experiment need be run could be reduced. It also seems improbable that the distribution of bad values of c' would have any relationship to primes under 18;

if this is true, we need not treat small prime powers separately.

This algorithm does not use very many properties of Z_p, so we can use the same algorithm to find discrete logarithms over other fields such as Z_{p^α}, as long as the field has a cyclic multiplicative group. All we need is that we know the order of the generator, and that we can multiply and take inverses of elements in polynomial time. The order of the generator could in fact be computed using the quantum order-finding algorithm given in section 5 of this paper. Boneh and Lipton [1995] have generalized the algorithm so as to be able to find discrete logarithms when the group is abelian but not cyclic.

7. Comments and open problems. It is currently believed that the most difficult aspect of building an actual quantum computer will be dealing with the problems of imprecision and decoherence. It was shown by Bernstein and Vazirani [1993] that the quantum gates need only have precision $O(1/t)$ in order to have a reasonable probability of completing t steps of quantum computation; that is, there is a c such that if the amplitudes in the unitary matrices representing the quantum gates are all perturbed by at most c/t, the quantum computer will still have a reasonable chance of producing the desired output. Similarly, the decoherence needs to be only polynomially small in t in order to have a reasonable probability of completing t steps of computation successfully. This holds not only for the simple model of decoherence where each bit has a fixed probability of decohering at each time step, but also for more complicated models of decoherence which are derived from fundamental quantum mechanical considerations [Unruh 1995; Palma, Suominen, and Ekert 1996; Chuang et al. 1995]. However, building quantum computers with high enough precision and low enough decoherence to accurately perform long computations may present formidable difficulties to experimental physicists. In classical computers, error probabilities can be reduced not only through hardware but also through software, by the use of redundancy and error-correcting codes. The most obvious method of using redundancy in quantum computers is ruled out by the theorem that quantum bits cannot be cloned [Peres 1993, section 9-4], but this argument does not rule out more complicated ways of reducing inaccuracy or decoherence using software. In fact, some progress in the direction of reducing inaccuracy [Berthiaume, Deutsch, and Jozsa 1994] and decoherence [Shor 1995] has already been made. The result of Bennett et al. [1996] that quantum bits can be faithfully transmitted over a noisy quantum channel gives further hope that quantum computations can similarly be faithfully carried out using noisy quantum bits and noisy quantum gates.

Discrete logarithms and factoring are not in themselves widely useful problems. They have become useful only because they have been found to be crucial for public-key cryptography, and this application is in turn possible only because they have been presumed to be difficult. This is also true of the generalizations of Boneh and Lipton [1995] of these algorithms. If the only uses of quantum computation remain discrete logarithms and factoring, it will likely become a special-purpose technique whose only *raison d'être* is to thwart public key cryptosystems. However, there may be other hard problems which could be solved asymptotically faster with quantum computers. In particular, of interesting problems not known to be NP-complete, the problem of finding a short vector in a lattice [Adleman 1994, Adleman and McCurley 1994] seems as if it might potentially be amenable to solution by a quantum computer.

In the history of computer science, however, most important problems have turned out to be either polynomial-time or NP-complete. Thus quantum computers will likely not become widely useful unless they can solve NP-complete problems. Solving NP-

complete problems efficiently is a Holy Grail of theoretical computer science which very few people expect to be possible on a classical computer. Finding polynomial-time algorithms for solving these problems on a quantum computer would be a momentous discovery. There are some weak indications that quantum computers are not powerful enough to solve NP-complete problems [Bennett et al. 1997], but I do not believe that this potentiality should be ruled out as yet.

Acknowledgments. I would like to thank Jeff Lagarias for finding and fixing a critical error in the first version of the discrete log algorithm. I would also like to thank him, David Applegate, Charlie Bennett, Gilles Brassard, Andrew Odlyzko, Dan Simon, Bob Solovay, Umesh Vazirani, and correspondents too numerous to list, for productive discussions, for corrections to and improvements of early drafts of this paper, and for pointers to the literature.

REFERENCES

L. M. Adleman (1994), *Algorithmic number theory—The complexity contribution,* in Proc. 35th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, pp. 88–113.

L. M. Adleman and K. S. McCurley (1994), *Open problems in number-theoretic complexity* II, in Algorithmic Number Theory, Proc. 1994 Algorithmic Number Theory Symposium, Ithaca, NY, Lecture Notes in Computer Science 877, L. M. Adleman and M.-D. Huang, eds., Springer-Verlag, Berlin, pp. 291–322.

A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter (1995a), *Elementary gates for quantum computation,* Phys. Rev. A, 52, pp. 3457–3467.

A. Barenco, D. Deutsch, A. Ekert, and R. Jozsa (1995b), *Conditional quantum dynamics and logic gates,* Phys. Rev. Lett., 74, pp. 4083–4086.

D. Beckman, A. N. Chari, S. Devabhaktuni, and J. Preskill (1996), *Efficient networks for quantum factoring,* Phys. Rev. A, 54, pp. 1034–1063.

P. Benioff (1980), *The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines,* J. Statist. Phys., 22, pp. 563–591.

P. Benioff (1982a), *Quantum mechanical Hamiltonian models of Turing machines,* J. Statist. Phys., 29, pp. 515–546.

P. Benioff (1982b), *Quantum mechanical Hamiltonian models of Turing machines that dissipate no energy,* Phys. Rev. Lett., 48, pp. 1581–1585.

C. H. Bennett (1973), *Logical reversibility of computation,* IBM J. Res. Develop., 17, pp. 525–532.

C. H. Bennett (1989), *Time/space trade-offs for reversible computation,* SIAM J. Comput., 18, pp. 766–776.

C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani (1997), *Strengths and weaknesses of quantum computing,* SIAM J. Comput., 26, pp. 1510–1523.

C. H. Bennett, G. Brassard, S. Popescu, B. Schumacher, J. A. Smolin, and W. K. Wooters (1996), *Purification of noisy entanglement and faithful teleportation via noisy channels,* Phys. Rev. Lett., 76, pp. 722–725.

E. Bernstein and U. Vazirani (1993), *Quantum complexity theory,* in Proc. 25th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, pp. 11–20; SIAM J. Comput., 26 (1997), pp. 1411–1473.

A. Berthiaume and G. Brassard (1992), *The quantum challenge to structural complexity theory,* in Proc. 7th Annual Structure in Complexity Theory Conference, IEEE Computer Society Press, Los Alamitos, CA, pp. 132–137.

A. Berthiaume and G. Brassard (1994), *Oracle quantum computing,* J. Modern Opt., 41, pp. 2521–2535.

A. Berthiaume, D. Deutsch, and R. Jozsa (1994), *The stabilisation of quantum computations,* in Proc. Workshop on Physics of Computation: PhysComp '94, IEEE Computer Society Press, Los Alamitos, CA, pp. 60–62.

M. Biafore (1994), *Can quantum computers have simple Hamiltonians,* in Proc. Workshop on Physics of Computation: PhysComp '94, IEEE Computer Society Press, Los Alamitos, CA, pp. 63–68.

D. Boneh and R. J. Lipton (1995), *Quantum cryptanalysis of hidden linear functions,* Advances in Cryptology—CRYPTO '95, Proc. 15th Annual International Cryptology Conference, Santa Barbara, CA, D. Coppersmith, ed. Springer-Verlag, Berlin, pp. 424–437.

J. F. Canny and J. Reif (1987), *New lower bound techniques for robot motion planning problems,* in Proc. 28th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, pp. 49–60.

J. Choi, J. Sellen, and C.-K. Yap (1995), *Precision-sensitive Euclidean shortest path in* 3*-space,* in Proc. 11th Annual Symposium on Computational Geometry, Association for Computing Machinery, New York, pp. 350–359.

I. L. Chuang, R. Laflamme, P. W. Shor, and W. H. Zurek (1995), *Quantum computers, factoring and decoherence,* Science, 270, pp. 1633–1635.

I. L. Chuang and Y. Yamamoto (1995), *A simple quantum computer,* Phys. Rev. A, 52, pp. 3489–3496.

A. Church (1936), *An unsolvable problem of elementary number theory,* Amer. J. Math., 58, pp. 345–363.

J. I. Cirac and P. Zoller (1995), *Quantum computations with cold trapped ions,* Phys. Rev. Lett., 74, pp. 4091–4094.

R. Cleve (1994), *A note on computing Fourier transforms by quantum programs,* preprint.

D. Coppersmith (1994), *An Approximate Fourier Transform Useful in Quantum Factoring,* IBM Research Report RC 19642.

D. Deutsch (1985), *Quantum theory, the Church–Turing principle and the universal quantum computer,* Proc. Roy. Soc. London Ser. A, 400, pp. 96–117.

D. Deutsch (1989), *Quantum computational networks,* Proc. Roy. Soc. London Ser. A, 425, pp. 73–90.

D. Deutsch, A. Barenco, and A. Ekert (1995), *Universality of quantum computation,* Proc. Roy. Soc. London Ser. A, 449, pp. 669-677.

D. Deutsch and R. Jozsa (1992), *Rapid solution of problems by quantum computation,* Proc. Roy. Soc. London Ser. A, 439, pp. 553–558.

D. P. DiVincenzo (1995), *Two-bit gates are universal for quantum computation,* Phys. Rev. A, 51, pp. 1015–1022.

A. Ekert and R. Jozsa (1996), *Shor's quantum algorithm for factorising numbers,* Rev. Mod. Phys., 68, pp. 733–753.

R. Feynman (1982), *Simulating physics with computers,* Internat. J. Theoret. Phys., 21, pp. 467–488.

R. Feynman (1986), *Quantum mechanical computers,* Found. Phys., 16, pp. 507–531; originally published in Optics News (February 1985), pp. 11–20.

E. Fredkin and T. Toffoli (1982), *Conservative logic,* Internat. J. Theoret. Phys., 21, pp. 219–253.

D. M. Gordon (1993), *Discrete logarithms in* GF(p) *using the number field sieve,* SIAM J. Discrete Math., 6, pp. 124–139.

R. B. Griffiths and C.-S. Niu (1996), *Semiclassical Fourier tranform for quantum computation,* Phys. Rev. Lett., 76, pp. 3228–3231.

G. H. Hardy and E. M. Wright (1979), *An Introduction to the Theory of Numbers,* 5th ed., Oxford University Press, New York.

J. Hartmanis and J. Simon (1974), *On the power of multiplication in random access machines,* in Proc. 15th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, Long Beach, CA, pp. 13–23.

A. Karatsuba and Yu. Ofman (1962), *Multiplication of multidigit numbers on automata,* Dokl. Akad. Nauk SSSR, 145, pp. 293–294 (in Russian); Sov. Phys. Dokl., 7 (1963), pp. 595–596 (English translation).

E. Knill (1995), personal communication.

D. E. Knuth (1981), *The Art of Computer Programming, Vol.* 2: *Seminumerical Algorithms,* 2nd ed., Addison-Wesley, Reading, MA.

R. Landauer (1995), *Is quantum mechanics useful*? Philos. Trans. Roy. Soc. London Ser. A, 353, pp. 367-376.

R. Landauer (1997), *Is quantum mechanically coherent computation useful*?, in Proc. Drexel-4 Symposium on Quantum Nonintegrability—Quantum Classical Correspondence, D. H. Feng and B-L. Hu, eds., International Press, Cambridge, MA, to appear.

Y. Lecerf (1963), *Machines de Turing réversibles. Récursive insolubilité en* $n \in N$ *de l'équation* $u = \theta^n u$*, où* θ *est un isomorphisme de codes,* C. R. Acad. Française Sci., 257, pp. 2597–2600.

A. K. Lenstra and H. W. Lenstra, Jr., eds. (1993), *The Development of the Number Field Sieve,* Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin.

A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard (1990), *The number field sieve,* in Proc. 22nd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, pp. 564–572; expanded version appears in Lenstra and Lenstra,

Jr. [1993], pp. 11–42.
R. Y. Levine and A. T. Sherman (1990), *A note on Bennett's time-space tradeoff for reversible computation,* SIAM J. Comput., 19, pp. 673–677.
S. Lloyd (1993), *A potentially realizable quantum computer,* Science, 261, pp. 1569–1571.
S. Lloyd (1994), *Envisioning a quantum supercomputer,* Science, 263, p. 695.
S. Lloyd (1995), *Almost any quantum logic gate is universal,* Phys. Rev. Lett., 75, pp. 346–349.
N. Margolus (1986), *Quantum computation,* Ann. New York Acad. Sci., 480, pp. 487–497.
N. Margolus (1990), *Parallel quantum computation,* in Complexity, Entropy and the Physics of Information, Santa Fe Institute Studies in the Sciences of Complexity, Vol. VIII, W. H. Zurek, ed., Addison-Wesley, Reading, MA, pp. 273–287.
G. L. Miller (1976), *Riemann's hypothesis and tests for primality,* J. Comput. System Sci., 13, pp. 300–317.
A. M. Odlyzko (1995), personal communication.
G. M. Palma, K.-A. Suominen, and A. K. Ekert (1996), *Quantum computers and dissipation,* Proc. Roy. Soc. London Ser. A, 452, pp. 567–584.
A. Peres (1993), *Quantum Theory*: *Concepts and Methods,* Kluwer Academic Publishers, Dordrecht, The Netherlands.
C. Pomerance (1987), *Fast, rigorous factorization and discrete logarithm algorithms,* in Discrete Algorithms and Complexity, Proc. Japan–US Joint Seminar, 1986, Kyoto, D. S. Johnson, T. Nishizeki, A. Nozaki, and H. S. Wilf, eds., Academic Press, New York, pp. 119–143.
E. Post (1936), *Finite combinatory processes. Formulation* I, J. Symbolic Logic, 1, pp. 103–105.
R. L. Rivest, A. Shamir, and L. Adleman (1978), *A method of obtaining digital signatures and public-key cryptosystems,* Comm. Assoc. Comput. Mach., 21, pp. 120–126.
L. A. Rubel (1989), *Digital simulation of analog computation and Church's thesis,* J. Symbolic Logic, 54, pp. 1011–1017.
A. Schönhage (1982), *Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients,* in Computer Algebra EUROCAM '82, Lecture Notes in Computer Science 144, J. Calmet, ed., Springer-Verlag, Berlin, pp. 3–15.
A. Schönhage, A. F. W. Grotefeld, and E. Vetter (1994), *Fast Algorithms*: *A Multitape Turing Machine Implementation,* B. I. Wissenschaftsverlag, Mannheim, Germany.
A. Schönhage and V. Strassen (1971), *Schnelle Multiplikation grosser Zahlen,* Computing, 7, pp. 281–292.
P. W. Shor (1994), *Algorithms for quantum computation*: *Discrete logarithms and factoring,* in Proc. 35th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, pp. 124–134.
P. W. Shor (1995), *Scheme for reducing decoherence in quantum computer memory,* Phys. Rev. A, 52, pp. 2493–2496.
D. Simon (1994), *On the power of quantum computation,* in Proc. 35th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, pp. 116–123; SIAM J. Comput., 26 (1997), pp. 1474–1483.
T. Sleator and H. Weinfurter (1995), *Realizable universal quantum logic gates,* Phys. Rev. Lett., 74, pp. 4087–4090.
R. Solovay (1995), personal communication.
K. Steiglitz (1988), *Two non-standard paradigms for computation*: *Analog machines and cellular automata,* in Performance Limits in Communication Theory and Practice, Proc. NATO Advanced Study Institute, Il Ciocco, Castelvecchio Pascoli, Tuscany, Italy, 1986, J. K. Skwirzynski, ed., Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 173–192.
W. G. Teich, K. Obermayer, and G. Mahler (1988), *Structural basis of multistationary quantum systems* II: *Effective few-particle dynamics,* Phys. Rev. B, 37, pp. 8111–8121.
T. Toffoli (1980), *Reversible computing,* in Automata, Languages and Programming, 7th Colloquium, Lecture Notes in Computer Science 84, J. W. de Bakker and J. van Leeuwen, eds., Springer-Verlag, Berlin, pp. 632–644.
A. M. Turing (1936), *On computable numbers, with an application to the Entscheidungsproblem,* in Proc. London Math. Soc. (2), 42, pp. 230–265; corrections in Proc. London Math. Soc. (2), 43 (1937), pp. 544–546.
W. G. Unruh (1995), *Maintaining coherence in quantum computers,* Phys. Rev. A, 51, pp. 992–997.
P. van Emde Boas (1990), *Machine models and simulations,* in Handbook of Theoretical Computer Science, Vol. A, J. van Leeuwen, ed., Elsevier, Amsterdam, pp. 1–66.
A. Vergis, K. Steiglitz, and B. Dickinson (1986), *The complexity of analog computation,* Math. Comput. Simulation, 28, pp. 91–113.
A. Yao (1993), *Quantum circuit complexity,* in Proc. 34th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, pp. 352–361.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1510–1523, October 1997

010

STRENGTHS AND WEAKNESSES OF QUANTUM COMPUTING*

CHARLES H. BENNETT†, ETHAN BERNSTEIN‡, GILLES BRASSARD§, AND UMESH VAZIRANI¶

Abstract. Recently a great deal of attention has been focused on quantum computation following a sequence of results [Bernstein and Vazirani, in *Proc. 25th Annual ACM Symposium Theory Comput.*, 1993, pp. 11–20, *SIAM J. Comput.,* 26 (1997), pp. 1411–1473], [Simon, in *Proc. 35th Annual IEEE Symposium Foundations Comput. Sci.*, 1994, pp. 116–123, *SIAM J. Comput.,* 26 (1997), pp. 1474–1483], [Shor, in *Proc. 35th Annual IEEE Symposium Foundations Comput. Sci.*, 1994, pp. 124–134] suggesting that quantum computers are more powerful than classical probabilistic computers. Following Shor's result that factoring and the extraction of discrete logarithms are both solvable in quantum polynomial time, it is natural to ask whether all of **NP** can be efficiently solved in quantum polynomial time. In this paper, we address this question by proving that relative to an oracle chosen uniformly at random with probability 1 the class **NP** cannot be solved on a quantum Turing machine (QTM) in time $o(2^{n/2})$. We also show that relative to a permutation oracle chosen uniformly at random with probability 1 the class $\mathbf{NP} \cap \mathbf{co\text{-}NP}$ cannot be solved on a QTM in time $o(2^{n/3})$. The former bound is tight since recent work of Grover [in *Proc. 28th Annual ACM Symposium Theory Comput.*, 1996] shows how to accept the class **NP** relative to any oracle on a quantum computer in time $O(2^{n/2})$.

Key words. quantum Turing machines, oracle quantum Turing machines, quantum polynomial time

AMS subject classifications. 68Q05, 68Q15, 03D10, 03D15

PII. S0097539796300933

1. Introduction.

Quantum computational complexity is an exciting new area that touches upon the foundations of both theoretical computer science and quantum physics. In the early eighties, Feynman [12] pointed out that straightforward simulations of quantum mechanics on a classical computer appear to require a simulation overhead that is exponential in the size of the system and the simulated time; he asked whether this is inherent, and whether it is possible to design a universal quantum computer. Deutsch [9] defined a general model of quantum computation—the QTM. Bernstein and Vazirani [4] proved that there is an efficient universal QTM. Yao [17] extended this by proving that quantum circuits (introduced by Deutsch [10]) are polynomially equivalent to QTMs.

The computational power of QTMs has been explored by several researchers. Early work by Deutsch and Jozsa [11] showed how to exploit some inherently quantum mechanical features of QTMs. Their results, in conjunction with subsequent results by Berthiaume and Brassard [5, 6], established the existence of oracles under which there are computational problems that QTMs can solve in polynomial

*Received by the editors March 21, 1996; accepted for publication (in revised form) December 2, 1996.

http://www.siam.org/journals/sicomp/26-5/30093.html

†IBM T. J. Watson Research Laboratory, Yorktown Heights, New York, NY 10598 (bennetc@watson.ibm.com).

‡Microsoft Corporation, One Microsoft Way, Redmond, WA 98052 (ethanb@microsoft.com). The research of this author was supported by NSF grant CCR-9310214.

§Département IRO, Université de Montréal, C. P. 6128, succursale centre-ville, Montréal (Québec), Canada H3C 3J7 (brassard@iro.umontreal.ca). The research of this author was supported in part by Canada's NSERC and Québec's FCAR.

¶Computer Science Division, University of California, Berkeley, CA 94720 (vazirani@cs.berkeley.edu). The research of this author was supported by NSF grant CCR-9310214.

time with certainty; whereas if we require a classical probabilistic Turing machine to produce the correct answer with certainty, then it must take exponential time on some inputs. On the other hand, these computational problems are in **BPP**[1] relative to the same oracle and are therefore efficiently solvable in the classical sense. The quantum analogue of the class **BPP** is the class **BQP**[2] [5]. Bernstein and Vazirani [4] proved that **BPP** $\subseteq$ **BQP** $\subseteq$ **PSPACE**, thus establishing that it will not be possible to conclusively prove that **BQP** $\neq$ **BPP** without resolving the major open problem **P** $\stackrel{?}{=}$ **PSPACE**. They also gave the first evidence that **BQP** $\neq$ **BPP** (polynomial-time QTMs are more powerful than polynomial-time probabilistic Turing machines) by proving the existence of an oracle relative to which there are problems in **BQP** that cannot be solved with small error probability by probabilistic machines restricted to running in $n^{o(\log n)}$ steps. Since **BPP** is regarded as the class of all "efficiently computable" languages (computational problems), this provided evidence that quantum computers are inherently more powerful than classical computers in a model-independent way. Simon [16] strengthened this evidence by proving the existence of an oracle relative to which **BQP** cannot even be simulated by probabilistic machines allowed to run for $2^{n/2}$ steps. In addition, Simon's paper also introduced an important new technique which was one of the ingredients in a remarkable result proved subsequently by Shor [15]. Shor gave polynomial-time quantum algorithms for the factoring and discrete logarithm problems. These two problems have been well studied, and their presumed intractability forms the basis of much of modern cryptography. In view of these results, it is natural to ask whether **NP** $\subseteq$ **BQP**; i.e., can quantum computers solve **NP**-complete problems in polynomial time?

In this paper, we address this question by proving that relative to an oracle chosen uniformly at random [3], with probability 1, the class **NP** cannot be solved on a QTM in time $o(2^{n/2})$. We also show that relative to a permutation oracle chosen uniformly at random, with probability 1, the class **NP** $\cap$ **co-NP** cannot be solved on a QTM in time $o(2^{n/3})$. The former bound is tight since recent work of Grover [13] shows how to accept the class **NP** relative to any oracle on a quantum computer in time $O(2^{n/2})$. See [7] for a detailed analysis of Grover's algorithm.

What is the relevance of these oracle results? We should emphasize that they do not rule out the possibility that **NP** $\subseteq$ **BQP**. What these results do establish is that there is no black-box approach to solving **NP**-complete problems by using some uniquely quantum-mechanical features of QTMs. That this was a real possibility is clear from Grover's [13] result, which gives a black-box approach to solving **NP**-complete problems in square-root as much time as is required classically.

One way to think of an oracle is as a special subroutine call whose invocation only costs unit time. In the context of QTMs, subroutine calls pose a special problem that has no classical counterpart. The problem is that the subroutine must not leave around any bits beyond its computed answer, because, otherwise, computational paths with different residual information do not interfere. This is easily achieved for deterministic subroutines since any classical deterministic computation can be carried out reversibly so that only the input and the answer remain. However, this leaves open the more

[1] **BPP** is the class of decision problems (languages) that can be solved in polynomial time by probabilistic Turing machines with error probability bounded by 1/3 (for all inputs). Using standard boosting techniques, the error probability can then be made exponentially small in k by iterating the algorithm k times and returning the majority answer.

[2] **BQP** is the class of decision problems (languages) that can be solved in polynomial time by QTMs with error probability bounded by 1/3 (for all inputs)—see [4] for a formal definition. We prove in section 4 that, as is the case with **BPP**, the error probability of **BQP** machines can be made exponentially small.

general question of whether a **BQP** machine can be used as a subroutine. Our final result in this paper is to show how any **BQP** machine can be modified into a *tidy* **BQP** machine whose final superposition consists almost entirely of a tape configuration containing just the input and the single bit answer. Since these tidy **BQP** machines can be safely used as subroutines, this allows us to show that $\mathbf{BQP}^{\mathbf{BQP}} = \mathbf{BQP}$. The result also justifies the definition of oracle quantum machines that we now give.

2. Oracle QTMs. In this section and the next, we shall assume without loss of generality that the Turing machine alphabet (for each track or tape) is $\{0, 1, \#\}$, where "#" denotes the blank symbol. Initially all tapes are blank except that the input tape contains the actual input surrounded by blanks. We shall use Σ to denote $\{0, 1\}$.

In the classical setting, an oracle may be described informally as a device for evaluating some Boolean function $A : \Sigma^* \to \Sigma$ on arbitrary arguments at unit cost per evaluation. This allows us to formulate questions such as, "If A were efficiently computable by a Turing machine, which other functions (or languages) could be efficiently computed by Turing machines?" In the quantum setting, an equivalent question can be asked, provided we define oracle QTMs appropriately, which we do in this section, and provided bounded-error QTMs can be composed, which we show in section 4.

An oracle QTM has a special *query tape* (or track), all of whose cells are blank except for a single block of nonblank cells. In a well-formed oracle QTM, the Turing machine rules may allow this region to grow and shrink but prevent it from fragmenting into noncontiguous blocks.[3] Oracle QTMs have two distinguished internal states: a prequery state q_q and a postquery state q_a. A query is executed whenever the machine enters the prequery state. If the query string is empty, a no-op occurs, and the machine passes directly to the postquery state with no change. If the query string is nonempty, it can be written in the form $x \circ b$ where $x \in \Sigma^*$, $b \in \Sigma$, and "$\circ$" denotes concatenation. In that case, the result of a call on oracle A is that internal control passes to the postquery state while the contents of the query tape changes from $|x \circ b\rangle$ to $|x \circ (b \oplus A(x))\rangle$, where "$\oplus$" denotes the exclusive-or (addition modulo 2). Except for the query tape and internal control, other parts of the oracle QTM do not change during the query. If the target bit $|b\rangle$ is supplied in initial state $|0\rangle$, then its final state will be $|A(x)\rangle$, just as in a classical oracle machine. Conversely, if the target bit is already in state $|A(x)\rangle$, calling the oracle will reset it to $|0\rangle$. This ability to "uncompute" will often prove essential to allow proper interference among computation paths to take place. Using this fact, it is also easy to see that the above definition of oracle QTMs yields unitary evolutions if we restrict ourselves to machines that are well formed in other respects, in particular evolving unitarily as they enter the prequery state and leave the postquery state.

The power of quantum computers comes from their ability to follow a coherent superposition of computation paths. Similarly oracle quantum machines derive great power from the ability to perform superpositions of queries. For example, oracle A might be called when the query tape is in state $|\psi \circ 0\rangle = \sum_x \alpha_x |x \circ 0\rangle$, where α_x are complex coefficients, corresponding to an arbitrary superposition of queries with a constant $|0\rangle$ in the target bit. In this case, after the query, the query string will be left in the entangled state $\sum_x \alpha_x |x \circ A(x)\rangle$. It is also useful to be able to put the target bit b into a superposition. For example, the conditional phase inversion used in Grover's algorithm can be achieved by performing queries with the target bit b in the nonclassical superposition $\beta = (|0\rangle - |1\rangle)/\sqrt{2}$. It can readily be verified that an

[3] This restriction can be made without loss of generality and it can be verified syntactically by allowing only machines that make sure they do not break the rule before writing on the query tape.

oracle call with the query tape in state $x \circ \beta$ leaves the entire machine state, including the query tape, unchanged if $A(x) = 0$, and leaves the entire state unchanged while introducing a phase factor -1 if $A(x) = 1$.

It is often convenient to think of a Boolean oracle as defining a length-preserving function on Σ^*. This is easily accomplished by interpreting the oracle answer on the pair (x, i) as the ith bit of the function value. The pair (x, i) is encoded as a binary string using any standard pairing function. A *permutation oracle* is an oracle which, when interpreted as a length-preserving function, acts for each $n \geq 0$ as a permutation on Σ^n. Henceforth, when no confusion may arise, we shall use $A(x)$ to denote the length-preserving function associated with oracle A rather than the Boolean function that gives rise to it.

Let us define $\mathbf{BQTime}(T(n))^A$ as the sets of languages accepted with probability at least $2/3$ by some oracle QTM M^A whose running time is bounded by $T(n)$. This bound on the running time applies to each individual input, not just on the average. Notice that whether or not M^A is a **BQP**-machine might depend upon the oracle A—thus M^A might be a **BQP**-machine while M^B might not be one.

Note: The above definition of a quantum oracle for an arbitrary Boolean function will suffice for the purposes of the present paper, but the ability of quantum computers to perform general unitary transformations suggests a broader definition, which may be useful in other contexts. For example, oracles that perform more general, non-Boolean unitary operations have been considered in computational learning theory [8] and for hiding information against classical queries [14].

Most broadly, a quantum oracle may be defined as a device that, when called, applies a fixed unitary transformation U to the current contents $|z\rangle$ of the query tape, replacing it by $U|z\rangle$. Such an oracle U must be defined on a countably infinite-dimensional Hilbert space, such as that spanned by the binary basis vectors $|\epsilon\rangle, |0\rangle, |1\rangle, |00\rangle, |01\rangle, |10\rangle, |11\rangle, |000\rangle, \ldots$, where ϵ denotes the empty string. Clearly, the use of such general unitary oracles still yields unitary evolution for well-formed oracle QTMs. Naturally, these oracles can map inputs onto superpositions of outputs, and vice versa, and they need not even be length preserving. However, in order to obey the dictum that a single machine cycle ought not to make infinite changes in the tape, one might require that $U|z\rangle$ have amplitude zero on all but finitely many basis vectors. (One could even insist on a uniform and effective version of the above restriction.) Another natural restriction one may wish to impose upon U is that it be an involution, $U^2 = I$, so that the effect of an oracle call can be undone by a further call on the same oracle. Again this may be crucial to allow proper interference to take place. Note that the special case of unitary transformation considered in this paper, which corresponds to evaluating a classical Boolean function, is an involution.

3. On the difficulty of simulating nondeterminism on QTMs. The computational power of QTMs lies in their ability to maintain and compute with exponentially large superpositions. It is tempting to try to use this "exponential parallelism" to simulate nondeterminism. However, there are inherent constraints on the scope of this parallelism, which are imposed by the formalism of quantum mechanics.[4] In this section, we explore some of these constraints.

[4] There is a superficial similarity between this exponential parallelism in quantum computation and the fact that probabilistic computations yield probability distributions over exponentially large domains. The difference is that in the probabilistic case, the computational path is chosen by making a sequence of random choices—one for each step. In the quantum-mechanical case, it is possible for several computational paths to interfere destructively, and therefore it is necessary to keep track of the entire superposition at each step to accurately simulate the system.

To see why quantum interference can speed up **NP** problems quadratically but not exponentially, consider the problem of distinguishing the empty oracle ($\forall_x A(x) = 0$) from an oracle containing a single random unknown string y of known length n (i.e., $A(y) = 1$, but $\forall_{x \neq y} A(x) = 0$). We require that the computer never answer yes on an empty oracle and seek to maximize its "success probability" of answering yes on a nonempty oracle. A classical computer can do no better than to query distinct n-bit strings at random, giving a success probability $1/2^n$ after one query and $k/2^n$ after k queries. How can a quantum computer do better, while respecting the rule that its overall evolution be unitary and, in a computation with a nonempty oracle, all computation paths querying empty locations evolve exactly as they would for an empty oracle? A direct quantum analogue of the classical algorithm would start in an equally weighted superposition of 2^n computation paths, query a different string on each path, and finally collapse the superposition by asking whether the query had found the nonempty location. This yields a success probability $1/2^n$, the same as the classical computer. However, this is not the best way to exploit quantum parallelism. Our goal should be to maximize the separation between the state vector $|\psi_k\rangle$ after k interactions with an empty oracle and the state vector $|\psi_k(y)\rangle$ after k interactions with an oracle nonempty at an unknown location y. Starting with a uniform superposition

$$|\psi_0\rangle = \frac{1}{\sqrt{2^n}} \sum_x |x\rangle,$$

it is easily seen that the separation after one query is maximized by a unitary evolution to

$$|\psi_1(y)\rangle = \frac{1}{\sqrt{2^n}} \sum_x (-1)^{\delta_{x,y}} |x\rangle = |\psi_0\rangle - \frac{2}{\sqrt{2^n}} |y\rangle.$$

This is a phase inversion of the term corresponding to the nonempty location. By testing whether the postquery state agrees with $|\psi_0\rangle$ we obtain a success probability

$$1 - |\langle \psi_0 | \psi_1(y) \rangle|^2 \approx 4/2^n$$

approximately four times better than the classical value. Thus, if we are allowed only one query, quantum parallelism gives a modest improvement but is still overwhelmingly likely to fail because the state vector after interaction with a nonempty oracle is almost the same as after interaction with an empty oracle. The only way of producing a large difference after one query would be to concentrate much of the initial superposition in the y term before the query, which cannot be done because that location is unknown.

Having achieved the maximum separation after one query, how best can that separation be increased by subsequent queries? Various strategies can be imagined, but a good one (called "inversion about the average" by Grover [13]) is to perform an oracle-independent unitary transformation so as to change the phase difference into an amplitude difference, leaving the y term with the same sign as all the other terms but a magnitude approximately threefold larger. Subsequent phase-inverting interactions with the oracle, alternating with oracle-independent phase-to-amplitude conversions, cause the distance between $|\psi_0\rangle$ and $|\psi_k(y)\rangle$ to grow linearly with k, approximately as $2k/\sqrt{2^n}$ when $k \leq \sqrt{N}/2$. This results in a quadratic growth of the success probability, approximately as $4k^2/2^n$ for small k. The proof of Theorem 3.5 shows that this approach is essentially optimal; no quantum algorithm can gain more than this quadratic factor in success probability compared with classical algorithms when attempting to answer **NP**-type questions formulated relative to a random oracle.

3.1. Lower bounds on quantum search. We will sometimes find it convenient to measure the accuracy of a simulation by calculating the Euclidean distance[5] between the target and simulation superpositions. The following theorem from [4] shows that the simulation accuracy is at most four times worse than this Euclidean distance.

THEOREM 3.1. *If two unit-length superpositions are within Euclidean distance ε, then observing the two superpositions gives samples from distributions which are within total variation distance*[6] *at most 4ε.*

DEFINITION 3.2. *Let $|\phi_i\rangle$ be the superposition of M^A on input x at time i. We denote by $q_y(|\phi_i\rangle)$ the sum of squared magnitudes in $|\phi_i\rangle$ of configurations of M which are querying the oracle on string y. We refer to $q_y(|\phi_i\rangle)$ as the* query magnitude of y *in $|\phi_i\rangle$.*

THEOREM 3.3. *Let $|\phi_i\rangle$ be the superposition of M^A on input x at time i. Let $\varepsilon > 0$. Let $F \subseteq [0, T-1] \times \Sigma^*$ be a set of time-string pairs such that $\sum_{(i,y)\in F} q_y(|\phi_i\rangle) \leq \frac{\varepsilon^2}{T}$. Now suppose the answer to each query $(i, y) \in F$ is modified to some arbitrary fixed $a_{i,y}$ (these answers need not be consistent with an oracle). Let $|\phi'_i\rangle$ be the time i superposition of M on input x with oracle A modified as stated above. Then $||\phi_T\rangle - |\phi'_T\rangle| \leq \varepsilon$.*

Proof. Let U be the unitary time evolution operator of M^A. Let A_i denote an oracle such that if $(i, y) \in F$ then $A_i(y) = a_{i,y}$ and if $(i, y) \notin F$ then $A_i(y) = A(y)$. Let U_i be the unitary time evolution operator of M^{A_i}. Let $|\phi_i\rangle$ be the superposition of M^A on input x at time i. We define $|E_i\rangle$ to be the error in the ith step caused by replacing the oracle A with A_i. Then

$$|E_i\rangle = U_i|\phi_i\rangle - U|\phi_i\rangle.$$

So we have

$$|\phi_T\rangle = U|\phi_{T-1}\rangle = U_{T-1}|\phi_{T-1}\rangle + |E_{T-1}\rangle = \cdots = U_{T-1}\cdots U_0|\phi_0\rangle + \sum_{i=0}^{T-1} U_{T-1}\cdots U_{i+1}|E_i\rangle.$$

Since all of the U_i are unitary, $|U_{T-1}\cdots U_i|E_i\rangle| = ||E_i\rangle|$.

The sum of squared magnitudes of all of the E_i is equal to $2\sum_{(i,y)\in F} q_y(|\phi_i\rangle)$ and therefore at most $\frac{\varepsilon^2}{T}$. In the worst case, the $U_{T-1}\cdots U_i|E_i\rangle$'s could interfere constructively; however, the squared magnitude of their sum is at most T times the sum of their squared magnitudes, i.e., ε^2. Therefore $||\phi_T\rangle - |\phi'_T\rangle| \leq \varepsilon$. □

COROLLARY 3.4. *Let A be an oracle over alphabet Σ. For $y \in \Sigma^*$, let A_y be any oracle such that $\forall x \neq y\ A_y(x) = A(x)$. Let $|\phi_i\rangle$ be the time i superposition of M^A on input x and $|\phi_i\rangle^{(y)}$ be the time i superposition of M^{A_y} on input x. Then for every $\varepsilon > 0$ there is a set S of cardinality at most $\frac{2T^2}{\varepsilon^2}$ such that $\forall y \notin S \left||\phi_T\rangle - |\phi_T\rangle^{(y)}\right| \leq \varepsilon$.*

Proof. Since each $|\phi_t\rangle$ has unit length, $\sum_{i=0}^{T-1}\sum_y q_y(|\phi_i\rangle) \leq T$. Let S be the set of strings y such that $\sum_{i=0}^{T-1} q_y(|\phi_i\rangle) \geq \frac{\varepsilon^2}{2T}$. Clearly $\text{card}(S) \leq \frac{2T^2}{\varepsilon^2}$.

If $y \notin S$ then $\sum_{i=0}^{T-1} q_y(|\phi_i\rangle) < \frac{\varepsilon^2}{2T}$. Therefore, by Theorem 3.3 $\forall y \notin S \left||\phi_i\rangle - |\phi_i\rangle^{(y)}\right| \leq \varepsilon$. □

THEOREM 3.5. *For any $T(n)$ which is $o(2^{n/2})$ relative to a random oracle with probability* 1, **BQTime**$(T(n))$ *does not contain* **NP**.

[5] The Euclidean distance between $|\phi\rangle = \sum_x \alpha_x|x\rangle$ and $|\psi\rangle = \sum_x \beta|x\rangle$ is defined as $\left(\sum_x |\alpha_x - \beta_x|^2\right)^{1/2}$.

[6] The total variation distance between two distributions $\mathcal{D}$ and $\mathcal{D}'$ is $\sum_x |\mathcal{D}(x) - \mathcal{D}'(x)|$.

Proof. Recall from section 2 that an oracle can be thought of as a length-preserving function; this is what we mean below by $A(x)$. Let $\mathcal{L}_A = \{y : \exists x\, A(x) = y\}$. Clearly, this language is contained in $\mathbf{NP}^A$. Let $T(n) = o(2^{n/2})$. We show that for any bounded-error oracle QTM M^A running in time at most $T(n)$, with probability 1, M^A does not accept the language $\mathcal{L}_A$. The probability is taken over the choice of a random length-preserving oracle A. Then, since there are a countable number of QTMs and the intersection of a countable number of probability 1 events still has probability 1, we conclude that with probability 1 no bounded error oracle QTM accepts $\mathcal{L}_A$ in time bounded by $T(n)$.

Since $T(n) = o(2^{n/2})$, we can pick n large enough so that $T(n) \leq \frac{2^{n/2}}{20}$. We will show that the probability that M gives the wrong answer on input 1^n is at least $1/8$ for every way of fixing the oracle answers on inputs of length not equal to n. The probability is taken over the random choices of the oracle for inputs of length n.

Let us fix an arbitrary length-preserving function from strings of lengths other than n over alphabet Σ. Let $\mathcal{C}$ denote the set of oracles consistent with this arbitrary function. Let $\mathcal{A}$ be the set of oracles in $\mathcal{C}$ such that 1^n has no inverse (does not belong to $\mathcal{L}_A$). If the oracle answers to length n strings are chosen uniformly at random, then the probability that the oracle is in $\mathcal{A}$ is at least $1/4$. This is because the probability that 1^n has no inverse is $(\frac{2^n-1}{2^n})^{2^n}$ which is at least $1/4$ (for n sufficiently large). Let $\mathcal{B}$ be the set of oracles in $\mathcal{C}$ such that 1^n has a unique inverse. As above, the probability that a randomly chosen oracle is in $\mathcal{B}$ is $(\frac{2^n-1}{2^n})^{2^n-1}$ which is at least $1/e$.

Given an oracle A in $\mathcal{A}$, we can modify its answer on any single input, say y, to 1^n and therefore get an oracle A_y in $\mathcal{B}$. We will show that for most choices of y, the acceptance probability of M^A on input 1^n is almost equal to the acceptance probability of M^{A_y} on input 1^n. On the other hand, M^A must reject 1^n and M^{A_y} must accept 1^n. Therefore M cannot accept both $\mathcal{L}_A$ and $\mathcal{L}_{A_y}$. By working through the details more carefully, it is easy to show that M fails on input 1^n with probability at least $1/8$ when the oracle is a uniformly random function on strings of length n and is an arbitrary function on all other strings.

Let A_y be the oracle such that $A_y(y) = 1^n$ and $\forall z \neq y\ A_y(z) = A(z)$. By Corollary 3.4 there is a set S of at most $338T^2(n)$ strings such that the difference between the $T(n)$th superposition of M^{A_y} on input 1^n and M^A on input 1^n has norm at most $1/13$. Using Theorem 3.1 we can conclude that the difference between the acceptance probabilities of M^{A_y} on input 1^n and M^A on input 1^n is at most $1/13 \times 4 < 1/3$. Since M^{A_y} should accept 1^n with probability at least $2/3$ and M^A should reject 1^n with probability at least $2/3$, we can conclude that M fails to accept either $\mathcal{L}_A$ or $\mathcal{L}_{A_y}$.

So, each oracle $A \in \mathcal{A}$ for which M correctly decides whether $1^n \in \mathcal{L}_A$ can, by changing a single answer of A to 1^n, be mapped to at least $(2^n - \mathrm{card}(S)) \geq 2^{n-1}$ different oracles $A_f \in \mathcal{B}$ for which M fails to correctly decide whether $1^n \in \mathcal{L}_{A_f}$. Moreover, any particular $A_f \in \mathcal{B}$ is the image under this mapping of at most $2^n - 1$ oracles $A \in \mathcal{A}$, since where it now answers 1^n, it must have given one of the $2^n - 1$ other possible answers. Therefore, the number of oracles in $\mathcal{B}$ for which M fails must be at least $1/2$ the number of oracles in $\mathcal{A}$ for which M succeeds. So, calling a the number of oracles in $\mathcal{A}$ for which M fails, M must fail for at least $a + 1/2(\mathrm{card}(\mathcal{A}) - a)$ oracles. Therefore M fails to correctly decide whether $1^n \in \mathcal{L}_A$ with probability at least $(1/2)P[\mathcal{A}] \geq 1/8$.

It is easy to conclude that M decides membership in $\mathcal{L}_A$ with probability 0 for a uniformly chosen oracle A. □

Note: Theorem 3.3 and its Corollary 3.4 isolate the constraints on "quantum parallelism" imposed by unitary evolution. The rest of the proof of the above theorem is similar in spirit to standard techniques used to separate **BPP** from **NP** relative to a random oracle [3]. For example, these techniques can be used to show that, relative to a random oracle A, no classical probabilistic machine can recognize $\mathcal{L}_A$ in time $o(2^n)$. However, quantum machines can recognize this language quadratically faster, in time $O(\sqrt{2^n})$, using Grover's algorithm [13]. This explains why a substantial modification of the standard technique was required to prove the above theorem.

The next result about $\mathbf{NP} \cap \mathbf{co\text{-}NP}$ relative to a random permutation oracle requires a more subtle argument; ideally we would like to apply Theorem 3.3 after asserting that the total query magnitude with which $A^{-1}(1^n)$ is probed is small. However, this is precisely what we are trying to prove in the first place.

THEOREM 3.6. *For any $T(n)$ which is $o(2^{n/3})$, relative to a random permutation oracle, with probability* 1, $\mathbf{BQTime}(T(n))$ *does not contain* $\mathbf{NP} \cap \mathbf{co\text{-}NP}$.

Proof. For any permutation oracle A, let $\mathcal{L}_A = \{y :$ first bit of $A^{-1}(y)$ is $1\}$. Clearly, this language is contained in $(\mathbf{NP} \cap \mathbf{co\text{-}NP})^A$. Let $T(n) = o(2^{n/3})$. We show that for any bounded-error oracle QTM M^A running in time at most $T(n)$, with probability 1, M^A does not accept the language $\mathcal{L}_A$. The probability is taken over the choice of a random permutation oracle A. Then, since there are a countable number of QTMs and the intersection of a countable number of probability 1 events still has probability 1, we conclude that with probability 1, no bounded error oracle QTM accepts $\mathcal{L}_A$ in time bounded by $T(n)$.

Since $T(n) = o(2^{n/3})$, we can pick n large enough so that $T(n) \leq \frac{2^{n/3}}{100}$. We will show that the probability that M gives the wrong answer on input 1^n is at least $1/8$ for every way of fixing the oracle answers on inputs of length not equal to n. The probability is taken over the random choices of the permutation oracle for inputs of length n.

Consider the following method of defining random permutations on $\{0,1\}^n$: let $x_0, x_1, \ldots, x_{T+1}$ be a sequence of strings chosen uniformly at random in $\{0,1\}^n$. Pick π_0 uniformly at random among permutations such that $\pi(x_0) = 1^n$. Let $\pi_i = \pi_{i-1} \cdot \tau$, where τ is the transposition (x_{i-1}, x_i), i.e., $\pi_i(x_i) = \pi_{i-1}(x_{i-1})$ and $\pi_i(x_{i-1}) = \pi_{i-1}(x_i)$. Clearly each π_i is a random permutation on $\{0,1\}^n$.

Consider a sequence of permutation oracles A_i such that $A_i(y) = A_j(y)$ if $y \notin \{0,1\}^n$ and $A_i(y) = \pi_i(y)$ if $y \in \{0,1\}^n$. Denote by $|\phi_i\rangle$ the time i superposition of $M^{A_{T(n)}}$ on input 1^n, and by $|\phi_i'\rangle$ the time i superposition of $M^{A_{T(n)-1}}$ on input 1^n. By construction, with probability exactly $1/2$, the string 1^n is a member of exactly one of the two languages $L_{A_{T(n)}}$ and $L_{A_{T(n)-1}}$. We will show that $E[\left||\phi_{T(n)}\rangle - |\phi'_{T(n)}\rangle\right|] \leq 1/50$. Here the expectation is taken over the random choice of the oracles. By Markov's bound, $P[\left||\phi_{T(n)}\rangle - |\phi'_{T(n)}\rangle\right| \leq 2/25] \geq 3/4$. Applying Theorem 3.1 we conclude that if $\left||\phi_{T(n)}\rangle - |\phi'_{T(n)}\rangle\right| \leq 2/25$, then the acceptance probability of $M^{A_{T(n)}}$ and $M^{A_{T(n)-1}}$ differ by at most $8/25 < 1/3$, and hence either both machines accept input 1^n or both reject that input. Therefore $M^{A_{T(n)}}$ and $M^{A_{T(n)-1}}$ give the same answers on input 1^n with probability at least $3/4$. By construction, the probability that the string 1^n belongs to exactly one of the two languages $L_{A_{T(n)}}$ and $L_{A_{T(n)-1}}$ is equal to $P[$first bit of $x_{T(n)-1} \neq$ first bit of $x_{T(n)}] = 1/2$. Therefore, we can conclude that with probability at least $1/4$, either $M^{A_{T(n)}}$ or $M^{A_{T(n)-1}}$ gives the wrong answer on input 1^n. Since each of $A_{T(n)}$ and $A_{T(n)-1}$ are chosen from the same distribution,

we can conclude that $M^{A_{T(n)}}$ gives the wrong answer on input 1^n with probability at least 1/8.

To bound $E[\left\||\phi_{T(n)}\rangle - |\phi'_{T(n)}\rangle\right\|]$, we show that $|\phi_{T(n)}\rangle$ and $|\phi'_{T(n)}\rangle$ are each close to a certain superposition $|\psi_{T(n)}\rangle$. To define this superposition, run M on input 1^n with a different oracle on each step; on step i, use A_i to answer the oracle queries. Denote by $|\psi_i\rangle$, the time i superposition that results. Consider the set of time-string pairs $S = \{(i, x_j) : \ j \geq i, \ 0 \leq i \leq T\}$. It is easily checked that the oracle queries in the computation described above and those of $M^{A_{T(n)}}$ and $M^{A_{T(n)+1}}$ differ only on the set S. We claim that the expected query magnitude of any pair in the set is at most $1/2^n$, since for $j \geq i$ we may think of x_j as having been randomly chosen during step j, *after* the superposition of oracle queries to be performed has already been written on the oracle tape. Let α be the sum of the query magnitudes for time-string pairs in S. Then

$$E[\alpha] \leq \text{card}(S)/2^n = \binom{T(n)+1}{2}/2^n \leq \frac{T(n)^2}{2^n}$$

for $T(n) \geq 4$. Let ε be a random variable such that $\alpha = \varepsilon^2/2T(n)$. Then by Theorem 3.3, $\left\||\phi\rangle - |\phi_{T(n)}\rangle\right\| \leq \varepsilon$ and $\left\||\phi\rangle - |\phi'_{T(n)}\rangle\right\| \leq \varepsilon$. We showed above that

$$E[\varepsilon^2/T(n)] = E[\alpha] \leq \frac{T(n)^2}{2^n}.$$

But $E[\varepsilon/\sqrt{2T(n)}]^2 \leq E[\varepsilon^2/2T(n)]$. Therefore,

$$\begin{aligned} E[\varepsilon] &= \sqrt{2T(n)}E[\varepsilon/\sqrt{2T(n)}] \leq \sqrt{2T(n)E[\varepsilon^2/2T(n)]} \leq \sqrt{2T(n)\frac{T(n)^2}{2^n}} \\ &\leq \sqrt{\frac{2}{100^3}} < 1/100. \end{aligned}$$

Therefore $E[\left\||\phi\rangle - |\phi_{T(n)}\rangle\right\|] \leq E[\varepsilon] < 1/100$ and $E[\left\||\phi\rangle - |\phi'_{T(n)}\rangle\right\|] \leq E[\varepsilon] < 1/100$. It follows that $E[\left\||\phi_{T(n)}\rangle - |\phi'_{T(n)}\rangle\right\|] < 1/50$.

Finally, it is easy to conclude that M decides membership in $\mathcal{L}_A$ with probability 0 for a uniformly random permutation oracle A. □

Note: In view of Grover's algorithm [13], we know that the constant "1/2" in the statement of Theorem 3.5 cannot be improved. On the other hand, there is no evidence that the constant "1/3" in the statement of Theorem 3.6 is fundamental. It may well be that Theorem 3.6 would still hold (albeit not its current proof) with 1/2 substituted for 1/3.

COROLLARY 3.7. *Relative to a random permutation oracle, with probability* 1, *there exists a* quantum one-way permutation. *Given the oracle, this permutation can be computed efficiently even with a classical deterministic machine, yet it requires exponential time to invert even on a quantum machine.*

Proof. Given an arbitrary permutation oracle A for which A^{-1} can be computed in time $o(2^{n/3})$ on a QTM, it is just as easy to decide $\mathcal{L}_A$ as defined in the proof of Theorem 3.6. It follows from that proof that this happens with probability 0 when A is a uniformly random permutation oracle. □

4. Using a bounded-error QTM as a subroutine. The notion of a subroutine call or an oracle invocation provides a simple and useful abstraction in the context of classical computation. Before making this abstraction in the context of quantum computation, there are some subtle considerations that must be thought through. For example, if the subroutine computes the function f, we would like to think of an invocation of the subroutine on the string x as magically writing $f(x)$ in some designated spot (actually xoring it to ensure unitarity). In the context of quantum algorithms, this abstraction is only valid if the subroutine cleans up all traces of its intermediate calculations and leaves just the final answer on the tape. This is because if the subroutine is invoked on a superposition of x's, then different values of x would result in different scratch work on the tape and would prevent these different computational paths from interfering. Since erasing is not a unitary operation, the scratch work cannot, in general, be erased postfacto. In the special case where f can be efficiently computed deterministically, it is easy to design the subroutine so that it reversibly erases the scratch work—simply compute $f(x)$, copy $f(x)$ into safe storage, and then uncompute $f(x)$ to get rid of the scratch work [2]. However, in the case that f is computed by a **BQP** machine, the situation is more complicated. This is because only some of the computational paths of the machine lead to the correct answer $f(x)$, and therefore if we copy $f(x)$ into safe storage and then uncompute $f(x)$, computational paths with different values of $f(x)$ will no longer interfere with each other, and we will not reverse the first phase of the computation. We show, nonetheless, that if we boost the success probability of the **BQP** machine before copying $f(x)$ into safe storage and uncomputing $f(x)$, then most of the weight of the final superposition has a clean tape with only the input x and the answer $f(x)$. Since such tidy **BQP** machines can be safely used as subroutines, this allows us to show that $\mathbf{BQP}^{\mathbf{BQP}} = \mathbf{BQP}$. The result also justifies our definition of oracle quantum machines.

The correctness of the boosting procedure is proved in Theorems 4.13 and 4.14. The proof follows the same outline as in the classical case, except that we have to be much more careful in simple programming constructs such as looping, etc. We therefore borrow the machinery developed in [4] for this purpose and present the statements of the relevant lemmas and theorems in the first part of this section. The main new contribution in this section is in the proofs of Theorems 4.13 and 4.14. The reader may therefore wish to skip directly ahead to these proofs.

4.1. Some programming primitives for QTMs. In this subsection, we present several definitions, lemmas and theorems from [4].

Recall that a QTM M is defined by a triplet (Σ, Q, δ) where Σ is a finite alphabet with an identified blank symbol $\#$, Q is a finite set of states with an identified initial state q_0 and final state $q_f \neq q_0$, and δ, the *quantum transition function*, is a function

$$\delta \;:\; Q \times \Sigma \rightarrow \tilde{\mathbf{C}}^{\Sigma \times Q \times \{L,R\}}$$

where $\tilde{\mathbf{C}}$ is the set of complex numbers whose real and imaginary parts can be approximated to within 2^{-n} in time polynomial in n.

DEFINITION 4.1. *A* final configuration *of a QTM is any configuration in state q_f. If when QTM M is run with input x, at time T the superposition contains only final configurations and at any time less than T the superposition contains no final configuration, then M* halts *with* running time T *on input x. The superposition of M at time T is called the* final superposition *of M run on input x. A polynomial-time QTM is a well-formed QTM which on every input x halts in time polynomial in the length of x.*

DEFINITION 4.2. *A QTM M is called* well behaved *if it halts on all input strings in a final superposition where each configuration has the tape head in the same cell. If this cell is always the start cell, we call the QTM* stationary.

We will say that a QTM M is in *normal form* if all transitions from the distinguished state q_f lead to the distinguished state q_0, the symbol in the scanned cell is left unchanged, and the head moves right, say. We formally state the following definition.

DEFINITION 4.3. *A QTM $M = (\Sigma, Q, \delta)$ is in* normal form *if*

$$\forall \sigma \in \Sigma \quad \delta(q_f, \sigma) = |\sigma\rangle|q_0\rangle|R\rangle.$$

THEOREM 4.4. *If f is a function mapping strings to strings which can be computed in deterministic polynomial time and such that the length of $f(x)$ depends only on the length of x, then there is a polynomial-time, stationary, normal form QTM which, given input x, produces output $x; f(x)$ and whose running time depends only on the length of x.*

If f is a one-to-one function from strings to strings such that both f and f^{-1} can be computed in deterministic polynomial time, and such that the length of $f(x)$ depends only on the length of x, then there is a polynomial-time, stationary, normal form QTM which, given input x, produces output $f(x)$ and whose running time depends only on the length of x.

DEFINITION 4.5. *A* multitrack *Turing machine with k tracks is a Turing machine whose alphabet Σ is of the form $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_k$ with a special blank symbol $\#$ in each Σ_i so that the blank in Σ is $(\#, \ldots, \#)$. We specify the input by specifying the string on each "track" (separated by ";"), and optionally by specifying the alignment of the contents of the tracks.*

LEMMA 4.6. *Given any QTM $M = (\Sigma, Q, \delta)$ and any set Σ', there is a QTM $M' = (\Sigma \times \Sigma', Q, \delta')$ such that M' behaves exactly like M while leaving its second track unchanged.*

LEMMA 4.7. *Given any QTM $M = (\Sigma_1 \times \cdots \times \Sigma_k, Q, \delta)$ and permutation $\pi : [1, k] \to [1, k]$, there is a QTM $M' = (\Sigma_{\pi(1)} \times \cdots \times \Sigma_{\pi(k)}, Q, \delta')$ such that the M' behaves exactly as M except that its tracks are permuted according to π.*

LEMMA 4.8. *If M_1 and M_2 are well-behaved, normal form QTMs with the same alphabet, then there is a normal form QTM M which carries out the computation of M_1 followed by the computation of M_2.*

LEMMA 4.9. *Suppose that M is a well-behaved, normal form QTM. Then there is a normal form QTM M' such that on input $x; k$ with $k > 0$, the machine M' runs M for k iterations on its first track.*

DEFINITION 4.10. *If QTMs M_1 and M_2 have the same alphabet, then we say that M_2* reverses *the computation of M_1 if the following holds: for any input x on which M_1 halts, let c_x and ϕ_x be the initial configuration and final superposition of M_1 on input x. Then M_2 on input the superposition ϕ_x, halts with final superposition consisting entirely of configuration c_x. Note that for M_2 to reverse M_1, the final state of M_2 must be equal to the initial state of M_1 and vice versa.*

LEMMA 4.11. *If M is a normal form QTM which halts on all inputs, then there is a normal form QTM M' that reverses the computation of M with slowdown by a factor of* 5.

Finally, recall the definition of the class **BQP**.

DEFINITION 4.12. *Let M be a stationary, normal form, multitrack QTM M whose last track has alphabet $\{\#, 0, 1\}$. We say that M accepts x if it halts with a* 1 *in the last track of the start cell. Otherwise, we say that M rejects x.*

A QTM accepts *the language* $\mathcal{L} \subseteq (\Sigma - \#)^*$ with probability p *if* M *accepts with probability at least* p *every string* $x \in \mathcal{L}$ *and rejects with probability at least* p *every string* $x \in (\Sigma - \#)^* - \mathcal{L}$. *We define the class* **BQP** (bounded-error quantum polynomial time) *as the set of languages which are accepted with probability* $2/3$ *by some polynomial-time QTM. More generally, we define the class* $\mathbf{BQTime}(T(n))$ *as the set of languages which are accepted with probability* $2/3$ *by some QTM whose running time on any input of length* n *is bounded by* $T(n)$.

4.2. Boosting and subroutine calls.

THEOREM 4.13. *If QTM* M *accepts language* $\mathcal{L}$ *with probability* $2/3$ *in time* $T(n) > n$, *with* $T(n)$ *time constructible, then for any* $\varepsilon > 0$ *there is a QTM* M' *which accepts* $\mathcal{L}$ *with probability* $1 - \varepsilon$ *in time* $cT(n)$ *where* c *is polynomial in* $\log 1/\varepsilon$ *but independent of* n.

Proof. Let M be a stationary QTM which accepts the language $\mathcal{L}$ in time $T(n)$.

We will build a machine that runs k independent copies of M and then takes the majority vote of the k answers. On any input x, M will have some final superposition of strings $\sum_i \alpha_i |x_i\rangle$. If we call A the set of i for which x_i has the correct answer $M(x)$ then $\sum_{i \in A} |\alpha_i|^2 \geq 2/3$. Now running M on separate copies of its input k times will produce $\sum_{i_1,\ldots,i_k} \alpha_{i_1} \cdots \alpha_{i_k} |x_{i_1}\rangle \cdots |x_{i_k}\rangle$. Then the probability of seeing $|x_{i_1}\rangle \cdots |x_{i_k}\rangle$ such that the majority have the correct answer $M(x)$ is the sum of $|\alpha_{i_1}|^2 \cdots |\alpha_{i_k}|^2$ such that the majority of $i_1, \ldots, i_k$ lie in A. But this is just like taking the majority of k independent coin flips each with probability at least $2/3$ of heads. Therefore, there is some constant b such that when $k = b \log 1/\varepsilon$, the probability of seeing the correct answer will be at least $1 - \varepsilon$.

So, we will build a machine to carry out the following steps.

1. Compute $n = T(|x|)$.
2. Write out k copies of the input x spaced out with $2n$ blank cells in between, and write down k and n on other tracks.
3. Loop k times on a machine that runs M and then steps n times to the right.
4. Calculate the majority of the k answers and write it back in the start cell.

We construct the desired QTM by building a QTM for each of these four steps and then dovetailing them together.

Since steps 1, 2, and 4 require easily computable functions whose output length depends only on k and the length of x, we can carry them out using well-behaved, normal form QTMs, constructed using Theorem 4.4, whose running times also depend only on k and the length of x.

So, we complete the proof by constructing a QTM to run the given machine k times. First, using Theorem 4.4 we can construct a stationary, normal form QTM which drags the integers k and n one square to the right on its work track. If we add a single step right to the end of this QTM and apply Lemma 4.9, we can build a well-behaved, normal form QTM moves which n squares to the right, dragging k and n along with it. Dovetailing this machine after M and then applying Lemma 4.9 gives a normal form QTM that runs M on each of the k copies of the input. Finally, we can dovetail with a machine to return with k and n to the start cell by using Lemma 4.9 two more times around a QTM which carries k and n one step to the left. □

The extra information on the output tape of a QTM can be erased by copying the desired output to another track and then running the reverse of the QTM. If the output is the same in every configuration in the final superposition, then this reversal will exactly recover the input. Unfortunately, if the output differs in different con-

figurations, then saving the output will prevent these configurations from interfering when the machine is reversed, and the input will not be recovered. We show that the output is the same in most of the final superpositions; then the reversal must lead us close to the input.

THEOREM 4.14. *If the language $\mathcal{L}$ is contained in the class* $\mathbf{BQTime}(T(n))$, *with $T(n) > n$ and $T(n)$ time constructible, then for any $\varepsilon > 0$ there is a QTM M' which accepts $\mathcal{L}$ with probability $1 - \varepsilon$ and has the following property. When run on input x of length n, M' runs for time bounded by $cT(n)$, where c is a polynomial in $\log 1/\varepsilon$, and produces a final superposition in which $|x\rangle|\mathcal{L}(x)\rangle$ with $\mathcal{L}(x) = 1$ if $x \in \mathcal{L}$ and 0 otherwise has squared magnitude at least $1 - \varepsilon$.*

Proof. Let $M = (\Sigma, Q, \delta)$ be a stationary, normal form QTM which accepts language $\mathcal{L}$ in time bounded by $T(n)$.

According to Theorem 4.13, at the expense of a slowdown by factor which is polynomial in $\log 1/\varepsilon$ but independent of n, we can assume that M accepts $\mathcal{L}$ with probability $1 - \varepsilon/2$ on every input.

Then we can construct the desired M' by running M, copying the answer to another track, and then running the reverse of M. The copy is easily accomplished with a simple two-step machine that steps left and back right while writing the answer on a clean track. Using Lemma 4.11, we can construct a normal form QTM M^R which reverses M. Finally, with appropriate use of Lemmas 4.6 and 4.7, we can construct the desired stationary QTM M' by dovetailing machines M and M^R around the copying machine.

To see that this M' has the desired properties, consider running M' on input x of length n. M' will first run M on x producing some final superposition of configurations $\sum_y \alpha_y|y\rangle$ of M on input x. Then it will write a 0 or 1 in the extra track of the start cell of each configuration, and run M^R on this superposition $|\phi\rangle = \sum_y \alpha_y|y\rangle|b_y\rangle$. If we were to instead run M^R on the superposition $|\phi'\rangle = \sum_y \alpha_y|y\rangle|M(x)\rangle$ we would after $T(n)$ steps have the superposition consisting entirely of the final configuration with output $x; M(x)$. Clearly, $\langle\phi|\phi'\rangle$ is real, and since M has success probability at least $1 - \varepsilon/2$, $\langle\phi|\phi'\rangle \geq \sqrt{1-\varepsilon}$. Therefore, since the time evolution of M^R is unitary and hence preserves the inner product, the final superposition of M' must have an inner product with $|x\rangle|M(x)\rangle$ which is real and at least $1 - \varepsilon/2$. Therefore, the squared magnitude in the final superposition of M' of the final configuration with output $x; M(x)$ must be at least $(1 - \varepsilon/2)^2 \geq 1 - \varepsilon$. ∎

COROLLARY 4.15. $\mathbf{BQP}^{\mathbf{BQP}} = \mathbf{BQP}$.

Acknowledgment. We wish to thank Bob Solovay for several useful discussions.

REFERENCES

[1] L. BABAI AND S. MORAN, *Arthur–Merlin games: A randomized proof system, and a hierarchy of complexity classes*, J. Comput. System Sci., 36 (1988), pp. 254–276.

[2] C. H. BENNETT, *Logical reversibility of computation*, IBM J. Res. Develop., 17 (1973), pp. 525–532.

[3] C. H. BENNETT AND J. GILL, *Relative to a random oracle A, $P^A \neq NP^A \neq co--NP^A$ with probability* 1, SIAM J. Comput., 10 (1981), pp. 96–113.

[4] E. BERNSTEIN AND U. VAZIRANI, *Quantum complexity theory*, in Proc. 25th Annual ACM Symposium Theory Comput., San Diego, CA, 1993, pp. 11–20; SIAM J. Comput., 26 (1997), pp. 1411–1473.

[5] A. BERTHIAUME AND G. BRASSARD, *The quantum challenge to structural complexity theory*, in Proc. 7th IEEE Conference on Structure in Complexity Theory, Boston, MA, 1992, pp. 132–137.

[6] A. Berthiaume and G. Brassard, *Oracle quantum computing*, J. Modern Opt., 41 (1994), pp. 2521–2535.

[7] M. Boyer, G. Brassard, P. Høyer, and A. Tapp, *Tight bounds on quantum searching*, in Proc. 4th Workshop on Phys. Comput., New England Complex Systems Institute, Boston, 1996, pp. 36–43. Available online in the *InterJournal* at `http://interjournal.org`.

[8] N. Bshouty and J. Jackson, *Learning DNF over uniform distribution using a quantum example oracle*, in Proc. 8th Annual ACM Conference on Comput. Learning Theory, Santa Cruz, CA, 1995, pp. 118–127.

[9] D. Deutsch, *Quantum theory, the Church-Turing principle and the universal quantum computer*, Proc. Roy. Soc. London A, 400 (1985), pp. 97–117.

[10] D. Deutsch, *Quantum computational networks*, Proc. Roy. Soc. London A, 425 (1989), pp. 73–90.

[11] D. Deutsch and R. Jozsa, *Rapid solution of problems by quantum computation*, Proc. Roy. Soc. London A, 439 (1992), pp. 553–558.

[12] R. Feynman, *Simulating physics with computers*, Internat. J. Theoret. Phys., 21 (1982), pp. 467–488.

[13] L. Grover, *A fast quantum mechanical algorithm for database search*, in Proc. 28th Annual ACM Symposium on Theory of Comput., Philadelphia, PA, 1996, pp. 212–219.

[14] J. Machta, *Phase Information in Quantum Oracle Computing*, manuscript, Physics Dept., University of Massachusetts at Amherst, Amherst, MA, 1996.

[15] P. W. Shor, *Algorithms for quantum computation: Discrete logarithms and factoring*, in Proc. 35th Annual IEEE Symposium on Foundations of Comput. Sci., Santa Fe, NM, 1994, pp. 124–134.

[16] D. Simon, *On the power of quantum computation*, in Proc. 35th Annual IEEE Symposium on Foundations of Comput. Sci., Santa Fe, NM, 1994, pp. 116–123; SIAM J. Comput., 26 (1997), pp. 1474–1483.

[17] A. Yao, *Quantum circuit complexity*, in Proc. 34th Annual IEEE Symposium on Foundations of Comput. Sci., Palo Alto, CA, 1993, pp. 352–361.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1524–1540, October 1997

011

QUANTUM COMPUTABILITY*

LEONARD M. ADLEMAN†, JONATHAN DEMARRAIS†, AND MING-DEH A. HUANG†

Abstract. In this paper some theoretical and (potentially) practical aspects of quantum computing are considered. Using the tools of transcendental number theory it is demonstrated that quantum Turing machines (QTM) with rational amplitudes are sufficient to define the class of bounded error quantum polynomial time (BQP) introduced by Bernstein and Vazirani [*Proc. 25th ACM Symposium on Theory of Computation*, 1993, pp. 11–20, *SIAM J. Comput.,* 26 (1997), pp. 1411–1473]. On the other hand, if quantum Turing machines are allowed unrestricted amplitudes (i.e., arbitrary complex amplitudes), then the corresponding BQP class has uncountable cardinality and contains sets of all Turing degrees. In contrast, allowing unrestricted amplitudes does not increase the power of computation for error-free quantum polynomial time (EQP). Moreover, with unrestricted amplitudes, BQP is not equal to EQP. The relationship between quantum complexity classes and classical complexity classes is also investigated. It is shown that when quantum Turing machines are restricted to have transition amplitudes which are algebraic numbers, BQP, EQP, and nondeterministic quantum polynomial time (NQP) are all contained in PP, hence in $P^{\#P}$ and PSPACE. A potentially practical issue of designing "machine independent" quantum programs is also addressed. A single ("almost universal") quantum algorithm based on Shor's method for factoring integers is developed which would run correctly on almost all quantum computers, even if the underlying unitary transformations are unknown to the programmer and the device builder.

Key words. quantum Turing machines, quantum complexity classes

AMS subject classifications. 68Q05, 68Q10, 68Q15

PII. S0097539795293639

1. Introduction.

In 1982, Feynman [F] considered computers based on quantum mechanical principles and speculated about the existence of a universal quantum simulator analogous to a universal Turing machine. That work was followed by a sequence of important papers by Deutsch [D1, D2], Deutch and Jouzsa [DJ], Bernstein and Vazirani [BV], and others which brought the topic to a state of development suitable for rigorous investigation [Y, Si]. Recently, the topic garnered great attention when Shor [Sh] argued that integer factoring (and the discrete logarithm problem) could be solved in polynomial time on a quantum machine. More formally, Shor asserted that a problem polynomial time equivalent to integer factoring was in the class BQP defined by Bernstein and Vazirani [BV]. Since much of public key cryptography is dependent on the difficulty of factoring and discrete logarithms, the existence of these machines could have a profound effect on cryptography. It is not yet known whether these machines can be built in practice.

In this paper we study some of the theoretical and (potentially) practical aspects of quantum computing. In addition to the class BQP, the classes EQP and NQP, the analogues of the classes P and NP, are also investigated.

The results in [BV] demonstrated that when considering BQP[1] one could restrict attention to QTMs which use rotations by the angle $R = 2\pi \sum_{i=1}^{\infty} 2^{-2^i}$. In this paper, the tools of transcendental number theory are used to demonstrate that, rather than R, the angle θ such that $\cos(\theta) = 3/5$ and $\sin(\theta) = 4/5$ is sufficient (the same result

*Received by the editors October 20, 1995; accepted for publication (in revised form) December 2, 1996. The research of the first and second authors was supported by NSF grant CCR-9403662. The research of the third author was supported by NSF grant CCR-9412383.

http://www.siam.org/journals/sicomp/26-5/29363.html

†Department of Computer Science, University of Southern California, Los Angeles, CA 90089-0781 (adleman@pollux.usc.edu, jed@pollux.usc.edu, huang@pollux.usc.edu).

[1] $BQP_{poly(1/\epsilon)}$ to be precise; see section 2.

has been announced by Solovay [So]). As a result, when considering BQP, one can restrict one's attention to QTMs with rational amplitudes.

We also address an issue concerning implementation of quantum computation. Building a physical device on which to run quantum algorithms apparently requires selecting from the physical universe a set of unitary transformations (e.g., rotations) which will be used as "primitive" operations. It is unclear to what extent the builder of such a device can choose or even know with arbitrary accuracy which unitary transformations have been selected. We show that a single ("almost universal") quantum algorithm based on Shor's result would run correctly on almost all devices (i.e., the set of unacceptable rotations has Lebesgue measure 0)—even if the underlying unitary transformations are unknown to the programmer and the device builder.

Whereas QTMs with rational amplitudes are sufficient for investigating BQP, it is of theoretical interest to understand the power of QTMs with no restrictions on the amplitudes allowed. It is shown that when QTMs are allowed "unrestricted" amplitudes (i.e., arbitrary complex amplitudes), the class of sets which are decidable with bounded error in polynomial time has uncountable cardinality and contains sets of all Turing degrees. In contrast, allowing unrestricted amplitudes does not increase the power of computation for the EQP class. In fact, it is shown that if a set is accepted in EQP by a QTM with unrestricted amplitudes, it is also accepted in EQP by a QTM with amplitudes that are (real) algebraic numbers. It is also shown that, with unrestricted amplitudes, BQP is not equal to EQP.

The relationship between quantum complexity classes and classical complexity classes is also investigated. It is shown that when QTMs are restricted to have transition amplitudes which are algebraic numbers, BQP, EQP, and NQP are all contained in PP, hence in $\mathrm{P}^{\#\mathrm{P}}$ and PSPACE. Finally, let EQP_θ consist of the sets in EQP accepted by QTMs equipped with a single primitive rotation by angle θ. It is demonstrated that for θ such that $\cos(\theta)$ is transcendental, $\mathrm{EQP}_\theta = \mathrm{P}$, in particular assuming $\cos(R)$ is transcendental, $\mathrm{EQP}_R = \mathrm{P}$.

2. Definitions and results. As defined in [BV], a QTM M is a Turing machine where each tuple specifying a transition is assigned an *amplitude* which is a complex number. As in [BV], we assume that M has no stationary transition. The transition function δ of M maps each transition tuple to its amplitude. It induces a linear map, called the *time evolution operator* of M, on the infinite-dimensional linear space $\mathcal{H}$ which has the set of all configurations as an orthonormal basis. A vector in $\mathcal{H}$ is a *superposition of configuration*. Thus, if, for example, $C_1, \ldots, C_m$ are the configurations M can reach in one step from a configuration C under δ, with amplitudes $a_1, \ldots, a_m$, then the time evolution operator maps C to $a_1C_1 + \cdots + a_mC_m$. A QTM M is *well formed* if its time evolution operator preserves the L_2-norm.[2]

Later we will have the need to refer to subsets of QTMs with restricted amplitudes. We introduce notation to facilitate this.

Given any field K, we define QTM_K to be the subset of QTMs whose amplitudes (i.e., the range of δ) are all in K. Examples include $\mathrm{QTM}_{\mathbf{C}}$, $\mathrm{QTM}_{\mathbf{R}}$, $\mathrm{QTM}_{\bar{\mathbf{Q}}}$, and $\mathrm{QTM}_{\mathbf{Q}}$. We will write QTM to refer to $\mathrm{QTM}_{\mathbf{C}}$.

According to [BV], one can assume without loss of generality that the amplitudes of δ are all real and that M enters any particular state from one direction. One can therefore define the local matrix $Ł_\delta$ whose columns are indexed by pairs of current state and symbol and rows by pairs of new state and symbol; and the entry

[2] A probabilistic Turing machine is in fact a Turing machine with a transition function δ whose amplitudes are restricted to 0,1 and 1/2 and whose time evolution operator preserves the L_1-norm.

corresponding to a column and a row is the amplitude for the associated transition.

If $\theta \in \mathbf{R}_{\geq 0}^{<2\pi}$, then we will define QTM_θ, to be the subset of QTM M whose local matrix is block diagonal (up to permutations of rows and columns) with each block either 1, -1, or 2 by 2 of the form

$$\begin{matrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta). \end{matrix}$$

Note that the 2×2 block represents rotation by angle θ. The universal QTM constructed by Bernstein and Vazirani [BV], for example, belongs to QTM_R, where $R = 2\pi \sum_{i=1}^{\infty} 2^{-2^i}$. We will also call this class QTM_{BV}. Another example is QTM_π, which consists of deterministic Turing machines with all entries either $0, \pm 1$.

On occasion we will use an ad hoc notation. For example, M is in $\mathrm{QTM}_{poly(1/\epsilon)}$ iff there exist an $f \in \mathbf{Z}[x]$ and a deterministic algorithm which, on input $1/\epsilon$, where $\epsilon \in \mathbf{Q}_{>0}^{<1}$, approximates all transition amplitudes of M within ϵ in $f(1/\epsilon)$ time.[3]

If Q_1 and Q_2 are subsets of QTM, then we will write $Q_1 \preceq Q_2$ iff for all $\epsilon \in \mathbf{Q}_{>0}^{<1}$, for all machines $M_1 \in Q_1$, there exists a machine $M_2 \in Q_2$ such that M_2 simulates M_1 to within ϵ with at most a polynomial slowdown, in the sense of [BV].[4] We use $\prec$, $\succ$, and $\asymp$ in the expected ways. For example, $\mathrm{QTM}_{\mathbf{C}} \asymp \mathrm{QTM}_{\mathbf{R}}$ as was shown by Bernstein and Vazirani [BV].

The classes BQP and EQP are due to Bernstein and Vazirani [BV]. The next definitions define restricted notions of BQP and EQP.

In this paper, $||$ denotes length in binary, except in section 3 where it denotes absolute value.

DEFINITION 2.1. *For all $T \subseteq$ QTM, for all $S \subseteq \mathbf{N}$, $S \in \mathrm{BQP}_T$ iff there exists an $f \in \mathbf{Z}[x]$ and an $M \in T$ such that, for all $x \in \mathbf{N}$,*

$x \in S \Rightarrow$ for input x, M accepts with probability greater than $2/3$ after $f(|x|)$ steps;

$x \in \overline{S} \Rightarrow$ for input x, M rejects with probability greater than $2/3$ after $f(|x|)$ steps.

DEFINITION 2.2. *For all $T \subseteq$ QTM, for all $S \subseteq \mathbf{N}$, $S \in \mathrm{EQP}_T$ iff there exists an $f \in \mathbf{Z}[x]$ and an $M \in T$ such that, for all $x \in \mathbf{N}$,*

$x \in S \Rightarrow$ for input x, M accepts with probability 1 after $f(|x|)$ steps;

$x \in \overline{S} \Rightarrow$ for input x, M rejects with probability 1 after $f(|x|)$ steps.

DEFINITION 2.3. *For all $T \subseteq$ QTM, for all $S \subseteq \mathbf{N}$, $S \in \mathrm{NQP}_T$ iff there exists an $f \in \mathbf{Z}[x]$ and an $M \in T$ such that, for all $x \in \mathbf{N}$,*

$x \in S \Rightarrow$ for input x, M accepts with positive probability after $f(|x|)$ steps;

$x \in \overline{S} \Rightarrow$ for input x, M accepts with probability 0 after $f(|x|)$ steps.

For convenience, for all fields K, when T=QTM_K, we will write BQP_K or EQP_K. For all $\theta \in \mathbf{R}_{\geq 0}^{<2\pi}$, when $T = \mathrm{QTM}_\theta$, we will write BQP_θ or EQP_θ. Similarly, when T=$\mathrm{QTM}_{poly(1/\epsilon)}$, we will write $\mathrm{BQP}_{poly(1/\epsilon)}$.

2.1. Results. Let $T_1, T_2 \subseteq$QTM . Then $T_1 \preceq T_2$ implies that $\mathrm{BQP}_{T_1} \subseteq \mathrm{BQP}_{T_2}$. (This is not merely an observation but requires a short proof which will not be given here.) The results in [BV] imply that $\mathrm{QTM}_{poly(1/\epsilon)} \preceq \mathrm{QTM}_R$, and since R is approximable in polynomial time within ϵ, it follows that $\mathrm{BQP}_{poly(1/\epsilon)} = \mathrm{BQP}_R$. Consequently, rotation by angle R serves as a universal primitive for $\mathrm{BQP}_{poly(1/\epsilon)}$. It is

[3] One can define $\mathrm{QTM}_{rpoly(1/\epsilon)}$ as above, but where instead of a deterministic algorithm a "probabilistic" one is used.

[4] That is, suppose in t steps M_1 produces, on input x, a superposition of configuration ϕ_1. Then in time polynomial in $1/\epsilon$ and t, M_2 produces, on the same input x, a superposition of configuration ϕ_2 such that the L_2-norm of $\phi_1 - \phi_2$ is less than ϵ.

natural to ask if R can be replaced by other angles, particularly an angle θ with rational $\cos(\theta)$ and $\sin(\theta)$. We show that for all angles θ in a set S of Lebesgue measure 1, $\mathrm{QTM}_R \preceq \mathrm{QTM}_\theta$, consequently any such θ can replace R as a universal angle. In particular the angle θ with $\cos(\theta) = 3/5$ is in S; hence, $\mathrm{BQP}_{poly(1/\epsilon)} = \mathrm{BQP}_{\mathbf{Q}}$. These results are presented in section 3.

The proof technique for the fact that $\mathrm{QTM}_R \preceq \mathrm{QTM}_\theta$ for all $\theta \in S$ can be applied to construct a program based on Shor's factoring algorithm which works universally for all QTM_θ with $\theta \in S$. This makes it possible to write a single program for a quantum computer without knowing the primitive rotation used by the machine. This is discussed in section 4.

The next set of results addresses the following question: would unrestricted amplitudes for transition functions increase the power of quantum computation? It is shown in section 5 that $\mathrm{BQP}_{\mathbf{C}}$ contains sets of arbitrary Turing degrees, hence undecidable sets in particular. In contrast, it is shown in section 6 that $\mathrm{EQP}_{\mathbf{C}} = \mathrm{EQP}_{\bar{\mathbf{Q}}}$. Moreover, $\mathrm{BQP}_{poly(1/\epsilon)}$, $\mathrm{EQP}_{\bar{\mathbf{Q}}}$, and $\mathrm{NQP}_{\bar{\mathbf{Q}}}$ are all contained in PP, hence in $\mathrm{P}^{\#\mathrm{P}}$ and PSPACE. As a result, $\mathrm{BQP}_{\mathbf{C}} \neq \mathrm{EQP}_{\mathbf{C}}$. The proofs for these equalities explore algebraic geometric structures underlying the EQP and NQP classes. The techniques also yield the following result: for angles θ with $\cos\theta$ transcendental, $\mathrm{EQP}_\theta = \mathrm{P}$. In particular, assuming $\cos(R)$ is transcendental, $\mathrm{EQP}_R = \mathrm{P}$.

3. $\mathbf{QTM_Q \asymp QTM_{BV}}$.

THEOREM 3.1. $\mathrm{QTM}_{BV} \asymp \mathrm{QTM}_{\mathbf{Q}} \asymp \mathrm{QTM}_{\bar{\mathbf{Q}}} \asymp \mathrm{QTM}_{poly(1/\epsilon)}$.

COROLLARY 3.2. $\mathrm{BQP}_{BV} = \mathrm{BQP}_{\mathbf{Q}} = \mathrm{BQP}_{\bar{\mathbf{Q}}} = \mathrm{BQP}_{poly(1/\epsilon)}$.

Essentially the same result has been announced by Solovay [So].

It can be easily demonstrated that $\mathrm{QTM}_{\mathbf{Q}} \preceq \mathrm{QTM}_{\bar{\mathbf{Q}}} \preceq \mathrm{QTM}_{poly(1/\epsilon)}$, and it follows from results on approximations [BV, BBBV] that $\mathrm{QTM}_{poly(1/\epsilon)} \preceq \mathrm{QTM}_{BV}$. Hence, Theorem 3.1 will follow from establishing that $\mathrm{QTM}_{BV} \preceq \mathrm{QTM}_\theta$ for some θ with rational $\cos\theta$ and $\sin\theta$. More generally, one would like to understand for what angles θ, $\mathrm{QTM}_{BV} \preceq \mathrm{QTM}_\theta$. The following theorem provides an answer.

THEOREM 3.3. *For all* $\theta \in \mathbf{R}_{\geq 0}^{<2\pi}$ *if either*

(a) $\theta/2\pi$ *is not rational and not Liouville*;

(b) $\theta/2\pi \in \bar{\mathbf{Q}} - \mathbf{Q}$;

(c) $e^{i\theta} \in \bar{\mathbf{Q}}$ *and* $e^{i\theta}$ *not a root of unity*;

(d) $\cos(\theta), \sin(\theta) \in \mathbf{Q} - \{0\}$;

(e) $\cos(\theta) = 3/5, \sin(\theta) = 4/5$;

then $\mathrm{QTM}_{BV} \preceq \mathrm{QTM}_\theta$.

Thus Theorem 3.1 follows from (e) of Theorem 3.3. It will be shown that (b)–(e) of Theorem 3.3 are actually subcases of (a). We also recall for Theorem 3.3 that (see, e.g., [Ni, B]) a real number ξ is a *Liouville number* if for every positive integer m there is a distinct rational number h_m/k_m with $k_m > 1$ such that $|\xi - h_m/k_m| < (k_m)^{-m}$.

COROLLARY 3.4. *For almost all* $\theta \in \mathbf{R}_{\geq 0}^{<2\pi}$, $\mathrm{QTM}_{BV} \preceq \mathrm{QTM}_\theta$, *where "almost all" means* $S = \{\theta | \theta \in \mathbf{R}_{\geq 0}^{<2\pi}$ *and* $\mathrm{QTM}_{BV} \preceq \mathrm{QTM}_\theta\}$ *has Lebesgue measure* 1.

Corollary 3.4 follows from Theorem 3.3 and the fact that the set of Liouville numbers has Lebesgue measure 0 (see, e.g., [B, p. 86]).

To prove Theorem 3.3 we will need the following lemma.

LEMMA 3.5. *For all* $\theta \in \mathbf{R}$ *with* $\theta/(2\pi)$ *not rational and not Liouville, there exists an* $f \in \mathbf{Z}[X]$ *such that for all* $\gamma \in \mathbf{R}$ *and all* $\epsilon \in \mathbf{R}_{>0}^{<1}$ *there exist* $x \in \mathbf{Z}_{\geq 0}$ *and* $w \in \mathbf{Z}$ *such that* $|x\theta - 2\pi w - \gamma| < \epsilon$ *and* $x < f(1/\epsilon)$.

Proof of Lemma 3.5. Given any irrational α and any positive integer n, there exist integers h and k with $0 < k \leq n$ such that $|k\alpha - h| < 1/n$ (see Theorem 4.2 of

[Ni, p. 44]). Since $\theta/2\pi$ is irrational, if $n = \lceil 2\pi/\epsilon \rceil$, then there exist $h, k \in \mathbf{Z}$ with $0 < k \leq n$ such that $|(k\theta/(2\pi)) - h| < 1/n$ or, equivalently, $|k\theta - 2\pi h| < 2\pi/n \leq \epsilon$.

Since $\theta/2\pi$ is not Liouville and not rational, there exists an $m \in \mathbf{Z}_{>0}$ such that for all $h', k' \in \mathbf{Z}$ with $k' > 1$, $|(\theta/(2\pi)) - h'/k'| \geq 1/k'^m$, and therefore $|k'\theta - 2\pi h'| \geq 2\pi/k'^{m-1}$. Hence $2\pi/k^{m-1} \leq |k\theta - 2\pi h| < \epsilon$.

Let $\beta = \gamma + 2\pi y'$ such that $y' \in \mathbf{Z}$, $-2\pi < \beta < 2\pi$, and $\beta/(k\theta - 2\pi h) \geq 0$. Let $x' = \lceil \beta/(k\theta - 2\pi h) \rceil$. Let $x = x'k$, and $y = x'h$, then $|x\theta - 2\pi y - \beta| = |(k\theta - 2\pi h)x' - \beta| < |(k\theta - 2\pi h)((\beta/(k\theta - 2\pi h)) + 1) - \beta| = |k\theta - 2\pi h| < \epsilon$. Also $|x'| < |\beta/(k\theta - 2\pi h)| + 1 \leq (|\beta|/|k\theta - 2\pi h|) + 1 \leq (|\beta|/(2\pi/k^{m-1})) + 1 < k^{m-1} + 1$ since $|\beta| < 2\pi$. Hence $|x| = x'k < k^m + k$, and since $k \leq n = \lceil 2\pi/\epsilon \rceil$, if $f = 8^m X^m + 8X$, then $x < f(1/\epsilon)$. Let $w = y + y'$, then $|x\theta - 2\pi w - \gamma| = |x\theta - 2\pi y - 2\pi y' - \gamma| = |x\theta - 2\pi y - \beta| < \epsilon$ as required. □

Remark. When $\theta, \gamma \in \mathbf{R}_{>0}^{<2\pi}$, then it follows easily from the lemma that $|w| \leq x + 2$.

The proof which follows demonstrates the capacity of a quantum computer to obtain an approximation to its intrinsic angle through experimentation. This capacity will also play an important role in the subsequent section on almost universal quantum programs.

Proof of Theorem 3.3(a). Given an $M \in \mathrm{QTM}_{BV}$ and $\epsilon \in \mathbf{Q}_{>0}^{<1}$, we will construct an $M' \in \mathrm{QTM}_\theta$ such that M' simulates M with accuracy ϵ and slowdown polynomial in $1/\epsilon$ and time t. The QTM M' will determine a natural number a such that $a\theta$ approximates the angle $R = 2\pi \sum_{i=1}^{\infty} 2^{-2^i}$ used by machines in QTM_{BV}. Once a is determined, M' simulates M by replacing each R-transition with a sequence of a θ-transitions.

In order to determine the number a, M' needs to compute an accurate enough rational approximation $\hat{\theta}$ of its underlying angle θ. Unlike R, the angle θ will not in general be easy to approximate deterministically. However, it is possible to have the QTM compute its own internal angle up to reflection about the x-axis and y-axis with arbitrary accuracy. That is, it is possible to compute approximations to $|\sin(\theta)|$ and $|\cos(\theta)|$. From these approximations an angle $\hat{\theta}$ which approximates θ can be determined. The correct $\hat{\theta}$ (of the four which are consistent with the approximated $|\sin(\theta)|$ and $|\cos(\theta)|$) depends on the quadrant that θ is in; however, in this existence proof, we may assume that this is known. For convenience, we will proceed under the assumption that θ is in the first quadrant. The other cases are handled in a similar manner.

Consider the following procedure that can be implemented on a machine in QTM_θ. On input $n, m \in \mathbf{Z}_{>0}$,

(i) begin with n, m on the tape;

(ii) place m^2n^2 0's on the tape, and for each 0 do a quantum step such that the symbol stays the same with probability $\cos^2(\theta)$ and becomes a 1 with probability $\sin^2(\theta)$;

(iii) observe the bits after the quantum flips and record the ratio r of the number of 0's to m^2n^2;

(iv) output $\hat{\theta} \in \mathbf{Q}$ such that $|\hat{\theta} - \theta'| < 1/n$, where $\theta' = \arccos(\sqrt{r})$.

In the above procedure, the number of 0's follows the binomial distribution with mean $\mu = m^2n^2\cos^2(\theta)$ and standard deviation $\sigma = mn\cos(\theta)\sin(\theta)$. From Chebyshev's inequality it follows that with probability at least $1 - 1/m^2$,

$$\left|r - \cos^2(\theta)\right| = |\cos^2(\theta') - \cos^2(\theta)| < m\sigma/m^2n^2 = \sin(2\theta)/2n < 1/2n.$$

Hence the ratio r approximates $\cos^2(\theta)$ better and better with increasing n. When n is large enough so that the recorded $r = \cos^2(\theta')$ is greater than $2/n$, we have $\cos^2(\theta) >$

$\cos^2(\theta') - 1/2n > 1/n$, and it also follows that $\cos^2(\theta') > \cos^2(\theta) - 1/2n > \cos^2(\theta)/2$. Since θ is in the first quadrant, it follows that $\cos(\theta') > \cos(\theta)/\sqrt{2}$. A similar argument shows that $\sin(\theta') > \sin(\theta)/\sqrt{2}$. Moreover if ψ is an angle between θ and θ', then $\cos(\psi) > \cos(\theta)/\sqrt{2}$ and $\sin(\psi) > \sin(\theta)/\sqrt{2}$; hence $2\sin(\psi)\cos(\psi) > \sin(\theta)\cos(\theta)$. By the mean value theorem,

$$|\theta - \theta'| = |\cos^2(\theta) - \cos^2(\theta')|/2\sin(\psi)\cos(\psi)$$

for some ψ between θ and θ'. It follows that

$$|\theta - \theta'| < |\cos^2(\theta) - \cos^2(\theta')|/\sin(\theta)\cos(\theta) < 1/n.$$

Hence

$$|\theta - \hat{\theta}| \leq |\theta - \theta'| + |\hat{\theta} - \theta'| < 2/n.$$

Consequently the probability that $|\theta - \hat{\theta}| < 2/n$ is at least $1 - 1/m^2$.

We will choose m so that $m > 1/\epsilon$. Let $\delta = \frac{\epsilon}{6t}$. We will run the above procedure on input m and increasing value of n, until the following conditions are met:

(a) the recorded number $r > 2/n$;

(b) there are integers a, b with $0 \leq a, |b| < (n\delta)/6$ such that $|a\hat{\theta} - b2\hat{\pi} - \hat{R}| < 2\delta$, where $\hat{R}, \hat{\pi} \in \mathbf{Q}$ such that $|R - \hat{R}| < \delta/3$ and $|2\pi - 2\hat{\pi}| < 1/n$.

Let f be the polynomial in Lemma 3.5 for θ. We argue that the condition (b) will be met before n exceeds $\frac{6(f(1/\delta)+2)}{\delta}$, which is polynomial in t and $1/\epsilon$.

Indeed let $n = \frac{6(f(1/\delta)+2)}{\delta}$. Then Lemma 3.5 implies the existence of a, b with $0 \leq a, |b| < f(1/\delta) + 2 = (n\delta)/6$ such that $|a\theta - b2\pi - R| < \delta$. With such a and b, $|a||\theta - \hat{\theta}|$, $|b||2\pi - 2\hat{\pi}|$, and $|R - \hat{R}|$ are all bounded by $\delta/3$. Since $|a\hat{\theta} - b2\hat{\pi} - \hat{R}| \leq |a\theta - b2\pi - R| + |a||\theta - \hat{\theta}| + |b||2\pi - 2\hat{\pi}| + |R - \hat{R}|$, it follows that $|a\hat{\theta} - b2\hat{\pi} - \hat{R}| < 2\delta$ as required.

With the computed $\hat{\theta}$ and a, b, we have

$$|a\theta - b2\pi - R| \leq |a\hat{\theta} - b2\hat{\pi} - \hat{R}| + |a||\theta - \hat{\theta}| + |b||2\pi - 2\hat{\pi}| + |R - \hat{R}| < 3\delta.$$

Hence rotation by the angle θ a times approximates rotation by angle R to within $3\delta = \epsilon/(2t)$.

Now on input x, M' can simulate M on input x by replacing each R-transition with a sequence of a θ-transitions appropriately.

Finally, the following argument adapted from [BBBV] (see also [BV]) shows that the superposition of configurations produced by M in t steps is approximated by M' within ϵ after the t steps of M are all simulated.

Let $\{C_1, C_2, ...\}$ be the set of configurations and $\mathcal{H}$ be the space of superpositions of configurations of M. Let U be the time evolution operator of M. Let $\hat{U}$ be the linear map on $\mathcal{H}$ determined by the local matrix which is obtained from the local matrix of M by replacing each 2×2 block representing rotation by angle R with a 2×2 block representing rotation by angle $a\theta$. Suppose at a certain time that ϕ is the superposition of configurations that M has. Then $U^t\phi$ is the superposition of configurations M arrives at after t steps, and $\hat{U}^t\phi$ is the superposition of configurations M' arrives at after simulating these t steps of M. Hence it suffices to show that $\|\hat{U}^t\phi - U^t\phi\| < \epsilon$, for all $\phi \in \mathcal{H}$ with $\|\phi\| = 1$, where $\| \|$ denotes the L_2-norm.

First we show that $\|\hat{U}\phi - U\phi\|^2 < \epsilon^2/t^2$. To see this, let $I(i)$ be the set of C_j such that there is a transition from C_j to C_i with nonzero amplitude; let $E(i)$ be the set

of C_j such that there is a transition from C_i to C_j with nonzero amplitude. Since the local matrix is block diagonal with each block of size at most 2×2, the cardinality of $I(i)$ and $E(i)$ are bounded by 2. Let $UC_i = \sum_j \alpha_{ij} C_j$ and $\hat{U}C_i = \sum_j \hat{\alpha}_{ij} C_j$. Then $|\alpha_{ij} - \hat{\alpha}_{ij}| < \Delta$ where $\Delta = 3\delta = \epsilon/(2t)$. Let $\phi = \sum_j \beta_j C_j$ be of L_2-norm equal to 1. Then

$$U\phi = \sum_i \left(\sum_{j \in I(i)} \alpha_{ji} \beta_j \right) C_i;$$

hence

$$\begin{aligned} \|\hat{U}\phi - U\phi\|^2 = \sum_i \left| \sum_{j \in I(i)} (\hat{\alpha}_{ji} - \alpha_{ji}) \beta_j \right|^2 &\leq \sum_i |I(i)| \sum_{j \in I(i)} |(\hat{\alpha}_{ji} - \alpha_{ji}) \beta_j|^2 \\ &< 2\Delta^2 \sum_i \sum_{j \in I(i)} \beta_j^2 = 2\Delta^2 \sum_j |E(j)| \beta_j^2 \leq 4\Delta^2 \sum_j \beta_j^2 = 4\Delta^2 = \epsilon^2/t^2. \end{aligned}$$

It is easy to see by induction that

$$\hat{U}^t \phi - U^t \phi = \sum_{i=1}^{t} \hat{U}^{t-i} E_i,$$

where $E_i = \hat{U}U^{i-1}\phi - U^i\phi$. Since $U^{i-1}\phi$ is of L_2-norm equal to 1,

$$\|E_i\|^2 = \|\hat{U}(U^{i-1}\phi) - U(U^{i-1}\phi)\|^2 \leq \epsilon^2/t^2.$$

So

$$\|\hat{U}^t \phi - U^t \phi\|^2 = \left\| \sum_{i=1}^{t} \hat{U}^{t-i} E_i \right\|^2 \leq t \sum_{i=1}^{t} \|\hat{U}^{t-i} E_i\|^2 = t \sum_{i=1}^{t} \|E_i\|^2 < \epsilon^2. \qquad \square$$

For the proof of Theorem 3.3(b)–(e), we need a lemma and proof which were generously provided to us by Harold Stark.

LEMMA 3.6. *For all $\theta \in \mathbf{R}_{>0}^{<2\pi}$, if $e^{i\theta} \in \bar{\mathbf{Q}}$ and $e^{i\theta}$ is not a root of unity, then $\theta/(2\pi)$ is not Liouville.*

Note that if θ satisfies the conditions in Lemma 3.6, then $\theta/(2\pi)$ is also not rational. Lemma 3.6 is a consequence of the next theorem which follows from results of Feldman [Fe].

THEOREM 3.7. *For all $a = \cos(b) + i\sin(b) = \exp(ib) \in \bar{\mathbf{Q}}$, with a not a root of unity, there exists a $C, N \in \mathbf{Z}_{>0}$ such that for all $p \in \mathbf{Z}$, $q \in \mathbf{Z}_{>1}$, $|p2\log(-1) - q\log(a)| > Cq^{-N}$, where C and N depend only on a and on the choice of the branch of logarithms used.*

Since $e^{i\theta}$ is algebraic and not a root of unity, by choosing the branch of logarithm such that $2\log(-1) = 2\pi i$, and $\log(e^{i\theta}) = i\theta$, we have that for all $p \in \mathbf{Z}$, $q \in \mathbf{Z}_{>1}$, $|p2\pi i - qi\theta| = |p2\pi - q\theta| > Cq^{-N}$. Hence $|\theta/2\pi - p/q| > C/(2\pi q^{N+1})$ from which it follows that $\theta/2\pi$ is not Liouville. This proves Lemma 3.6. $\square$

Proof of Theorem 3.3(b)–(e). (b) follows from (a) since Liouville numbers are transcendental (see, e.g., [Ni, p. 92]). (c) follows from (a) and Lemma 3.6. (d) follows from (c) and the fact that the only roots of unity with rational real and imaginary parts are 1 and i. Finally, (e) follows from (d). $\square$

4. Almost universal quantum programs. Shor's quantum factoring method is of interest for purely mathematical reasons. However, it is unclear whether devices can be built which will actually implement Shor's method. It follows from the previous section that apparently it is enough to build a device capable of a rotation θ with $\cos(\theta) = 3/5$. But can such a device be built? To what extent can a device builder choose the angles of rotations which will be provided? To what accuracy can the device builder or the programmer know the angles of rotation inherent in the device? It turns out that these questions may be irrelevant and that one can write a single "program" which will factor with high probability on virtually any device which can be built.

This is done essentially as indicated in the proof of Theorem 3.3. One starts with a program for Shor's method which will factor with high probability on a device with rotation by the angle θ with $\cos(\theta) = 3/5$ as primitive. One then writes a new "almost universal program" which will, in the manner described in the proof of Theorem 3.3, calculate "on line" a sufficiently accurate estimation of whatever angle of rotation γ the underlying physical device provides and then use that estimate to simulate the original program with sufficient accuracy to insure factoring. Since the estimate of γ is only unique up to the sign of $\cos(\gamma)$ and $\sin(\gamma)$, we will simultaneously run all four possible approximations; however, this will neither increase the running time significantly nor decrease the probability of a successful factorization substantially. It follows from Corollary 3.4 that the "almost universal program" will factor with high probability on all but those devices with an angle of rotation in a set of Lebesgue measure 0.

5. Unrestricted quantum computation. We have demonstrated that restricting attention to quantum machines with rational amplitudes is sufficient to define the class BQP. However, it is of theoretical interest to understand the power of "unrestricted" quantum computation. In this section, it is shown that $\mathrm{BQP}_{\mathbf{C}}$ has sets of all possible Turing degrees.

THEOREM 5.1. *For all $S \subseteq \mathbf{N}$, there exists an $S' \subseteq \mathbf{N}$ such that*

1. *$S \equiv_T S'$ (Turing equivalent);*
2. *$S' \in \mathrm{BQP}_{\mathbf{C}}$.*

Proof. Throughout this proof, for all $x \in \mathbf{N}$, $|x|$ will denote the length of x. Let $S \subseteq \mathbf{N}$ and let $H : \mathbf{N} \to \{1, -1\}$ be such that for all $x \in \mathbf{N}$, $H(x) = 1$ if $x \in S$, $H(x) = -1$ if $x \in \overline{S}$. Let $S' \subseteq \mathbf{N}$ such that for all $x \in \mathbf{N}$, $x \in S'$ if and only if $H(i+1) = 1$ where i is the greatest integer such that $8^i \leq |x|$. It is clear from the definition that S and S' are Turing equivalent.

Let $\theta = 2\pi(\sum_{x=1}^{\infty} H(x)/8^x)$. Then $\theta \in \mathbf{R}$ (i.e., the series converges).

Let M be a QTM which on input $x \in \mathbf{N}$.

1. Calculate the greatest l such that $l \leq |x|$ and $l = 8^i$ for some $i \in \mathbf{N}$.
2. Place a 0 on the tape.
3. Perform l rotations by the angle θ on this tape location followed by a single rotation by the angle $\pi/4$, so that the amplitude of the configuration with 0 on the tape is $\cos(l\theta + \pi/4)$, and the amplitude with 1 on the tape is $\sin(l\theta + \pi/4)$. This is done in a manner similar to the method used by Bernstein and Vazirani [BV].
4. Output 0 or 1 based on the value of the given tape location. If the output is 1, the machine accepts, and if the output is 0, the machine rejects.

It is clear from the algorithm that for all $x \in \mathbf{N}$, the number of steps taken by M on input x is polynomially bounded in $|x|$.

After the third step of the algorithm, the configuration that corresponds to an output of 0 has amplitude $\cos(l\theta + \pi/4)$, and the configuration that corresponds to

an output of 1 has amplitude $\sin(l\theta + \pi/4)$ for the greatest l such that $l \leq |x|$ and l is power of 8. Let $l = 8^i$. Then

$$l\theta = 2\pi(H(1)8^{i-1} + H(2)8^{i-2} + \cdots + H(i) + H(i+1)/8 + H(i+2)/8^2 + \cdots).$$

Let $k = \pi/4 + l\theta \bmod 2\pi$. If $H(i+1) = 1$, then $\pi/2 - \pi/28 \leq k \leq \pi/2 + \pi/28$, in this case $\sin^2(\pi/4 + l\theta) > 0.98$; hence the input x is accepted with probability greater than 0.98. On the other hand if $H(i+1) = -1$, then $-\pi/28 \leq k \leq \pi/28$, in this case $\cos^2(\pi/4 + l\theta) > 0.98$; hence x is rejected with probability greater than 0.98. Therefore M accepts S' in $\mathrm{BQP}_{\mathbf{C}}$ and the theorem follows. □

Since there are uncountably many subsets of N and each Turing class can contain at most countably many subsets of N (since there are only this many TM), it follows that $\mathrm{BQP}_{\mathbf{C}}$ has uncountable cardinality. From this it follows that $\mathrm{BQP}_{\mathbf{Q}}$ (which clearly has countable cardinality) is a proper subset of $\mathrm{BQP}_{\mathbf{C}}$. From this in turn (see section 2) it follows that $\mathrm{QTM}_{\mathbf{Q}} \prec \mathrm{QTM}_{\mathbf{C}}$ and not $\mathrm{QTM}_{\mathbf{Q}} \succeq \mathrm{QTM}_{\mathbf{C}}$.

6. EQP and NQP. Let M be a QTM with transition function δ. Following [BV] we can assume without loss of generality that δ is a real-valued function.

Suppose there are N tuples $T_1, \ldots, T_N$ in the domain of δ. Let $v = (v_1, \ldots, v_N)$ where $v_i = \delta(T_i)$, the amplitude of δ on transition T_i, for $i = 1, \ldots, N$. We shall call v the *amplitude vector* of δ.

Replacing v by $x = (x_1, \ldots, x_N)$, where x_i are distinct complex variables, we obtain a "symbolic" QTM $M(x)$. By assigning a vector $u \in \mathbf{C}^N$ to x we obtain a QTM denoted by $M(u)$, in particular, $M = M(v)$. The QTMs obtained in this way have transition functions with the same domain, the same set of configurations, hence the same space of superpositions of configurations. They are distinguished by having different amplitude vectors associated with the transition function.

For $u \in \mathbf{R}^N$, $M(u)$ is well formed iff u satisfies a finite set of polynomial equations which can be obtained as follows.

Let $A(x)$ denote the infinite-dimensional matrix representing the (symbolic) time evolution operator of $M(x)$, the linear map induced by δ, on the infinite-dimensional space of superpositions of configurations [BV] with respect to the basis consisting of the whole set of configurations of $M(x)$. The rows and columns of $A(x)$ are labelled by the set of configurations. If upon applying the ith transition, a configuration C yields C', then x_i will be the entry corresponding to column labelled by C and the row labelled by C'. If no such transition exists, the corresponding entry will be 0. Thus each entry of $A(x)$ is one of the variables $x_1, \ldots, x_N$, and for all $u = (u_1, \ldots, u_N) \in \mathbf{R}^N$, by substituting x_i with u_i for $i = 1, \ldots, N$, we obtain the time evolution matrix $A(u)$ for $M(u)$. For all $i, j \in \mathbf{Z}_{>0}$, let $P_{ij}(x)$ denote the dot product of the ith and jth columns of $A(x)$. Then since each column of $A(x)$ has at most N nonzero entries, P_{ij} is the sum of at most N monomials in $x_1, \ldots, x_N$ of degree 2. In particular the set of P_{ij} is finite.

By [BV], $M(u)$ is well formed iff $A(u)^*A(u) = A(u)A(u)^* = I$, where $A(u)^*$ denotes the transpose conjugate of $A(u)$. Since u is real, $A(u)^*$ is just the transpose of $A(u)$. Hence $M(u)$ is well formed iff for all i, j, if $i \neq j$, then $P_{ij}(u) = 0$; if $i = j$, then $P_{ij}(u) = 1$. Hence let J_W be the set of polynomials P_{ij} for $i \neq j$ and $1 - P_{ii}$ for all i. Then $M(u)$ is well formed iff u is a zero to all polynomials in J_W. Note that J_W is finite since the set of P_{ij} is finite.

As will be demonstrated below, whether $M(u)$ accepts or rejects an input with probability 1 can also be characterized algebraically.

For all $i \in \mathbf{Z}_{>0}$, let $A^i(x)$ denote the product of $A(x)$ by itself i times. We note that each entry in $A^i(x)$ is a polynomial (possibly 0) in $\mathbf{Z}[X]$. Let α be an input string.

Let C_α be the initial configuration of $M(x)$ (hence of M and for all $u \in \mathbf{R}^N$, of $M(u)$) on input α. Then for all configurations C, for all $t \in \mathbf{Z}_{>0}$, we denote by $amp_{\alpha,t,C}$ the polynomial which is in the entry of $A^t(x)$ corresponding to the column labelled by C_α and the row labelled by C. For all $u \in \mathbf{C}^N$, $amp_{\alpha,t,C}(u)$ is the amplitude for C at time t on input α to $M(u)$.

Let $\mathcal{C}_A$ denote the set of accepting configurations and $\mathcal{C}_R = \mathcal{C} - \mathcal{C}_A$ the set of rejecting configurations. Suppose $u \in \mathbf{R}^N$, and assuming $M(u)$ is well formed, then an input α is accepted at time t with probability 1 iff

$$\sum_{C \in \mathcal{C}_R} |amp_{\alpha,t,C}(u)|^2 = \sum_{C \in \mathcal{C}_R} amp^2_{\alpha,t,C}(u) = 0,$$

iff $amp_{\alpha,t,C}(u) = 0$ for all $C \in \mathcal{C}_R$. Hence let

$$J_A(t,\alpha) = \{amp_{\alpha,t,C} : C \in \mathcal{C}_R\}.$$

Then $M(u)$ accepts α at time t with probability 1 iff u is a zero of all polynomials in $J_A(t,\alpha)$. Similarly, let

$$J_R(t,\alpha) = \{amp_{\alpha,t,C} : C \in \mathcal{C}_A\}.$$

Then $M(u)$ rejects α with probability 1 iff u is a zero of all polynomials in $J_R(t,\alpha)$.

Let L be a language over the input alphabet of M. Let p be a polynomial function where $p(n)$ is a positive integer for all $n \in \mathbf{Z}_{>0}$. For all inputs α, let $t_\alpha = p(n)$ where n is the length of α. Let

$$J(p,L) = J_W \cup_{\alpha \in L} J_A(t_\alpha,\alpha) \cup_{\alpha \notin L} J_R(t_\alpha,\alpha).$$

From the discussion above we see that for all $u \in \mathbf{R}^N$, $M(u)$ is well formed and accepts L in EQP at time $p(n)$ iff u is a zero of all polynomials in $J(p,L)$.

We recall the following proposition.

PROPOSITION 6.1. *Let I be an ideal in $\mathbf{Q}[x_1,\ldots,x_N]$. If the polynomials in I have a common zero in $\mathbf{R}^N$, then they have a common zero in $(\bar{\mathbf{Q}} \cap \mathbf{R})^N$.*

The proposition follows immediately from Tarski's theorem (see, e.g., [J, p. 323]), noting the fact that I is finitely generated and that both $\mathbf{R}$ and $\bar{\mathbf{Q}} \cap \mathbf{R}$ are real closed fields. It is a direct consequence of the mathematical principle implied by Tarski's theorem that any elementary sentence of algebra which is true in one real closed field is true in every real closed field.

Suppose L is accepted by $M = M(v)$ in EQP at time $p(n)$. Then v is a real zero of polynomials in the ideal I of $\mathbf{Q}[x_1,\ldots,x_N]$ generated by $J(p,L)$. From Proposition 6.1 it follows that the polynomials in I also have a common zero $u \in (\bar{\mathbf{Q}} \cap \mathbf{R})^N$. Since u is a zero of all polynomials in $J(p,L)$, it follows that $M(u)$ is well formed and accepts L in EQP at time $p(n)$.

Hence we have the following.

THEOREM 6.2. $\mathrm{EQP}_{\mathbf{C}} = \mathrm{EQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}}$.

Finally, we give two results which relate quantum complexity classes to classical complexity classes.

THEOREM 6.3. *For all $\theta \in \mathbf{R}^{<2\pi}_{\geq 0}$ such that $\cos\theta$ is transcendental,* $\mathrm{EQP}_\theta = \mathrm{P}$. *In particular assuming $\cos(R)$ is transcendental,* $\mathrm{EQP}_R = \mathrm{P}$.

Proof. Let $M \in \mathrm{EQP}_\theta$ with amplitude vector $v = (v_1,\ldots,v_N) \in \mathbf{R}^N$. Then v_i is 0, ± 1, $\pm\cos\theta$, or $\pm\sin\theta$. Suppose $v_i \in \{0,\pm 1\}$ for all i. Since the time evolution

operator of M preserves the L_2-norm, there is at most one transition tuple with amplitude ± 1 for every pair of current state and symbol. Hence M is deterministic.

Suppose on the other hand that v contains some transcendental coordinates. Let L be accepted by M in EQP at time $p(n)$ for some polynomial p. Let $x = (x_1, \ldots, x_N)$. Form the symbolic QTM $M(x)$ and the set of polynomials $J(p, L)$ as before. Then for $u \in \mathbf{R}^N$, $M(u)$ is well formed and accepts L in EQP at time $p(n)$ iff u is a common zero of all polynomials in $J(p, L)$. Let s and t be variables and $u(s, t)$ be the N-vector obtained from x as follows: for all i, replace x_i by v_i if v_i is 0 or ± 1, replace x_i by $\pm s$ if $v_i = \pm \cos\theta$, and replace x_i by $\pm t$ if $v_i = \pm \sin\theta$. Let $J'(p, L) \subset \mathbf{Q}[s, t]$ be obtained from $J(p, L)$ by specializing each polynomial $F(x)$ in $J(p, L)$ to $F(u(s, t))$. Then for all $a, b \in \mathbf{R}$, $M(u(a, b))$ accepts L in EQP at time $p(n)$ iff a, b is a common zero of all polynomials in $J'(p, L)$. In particular we note that $v = u(\cos\theta, \sin\theta)$ and that $(\cos\theta, \sin\theta)$ is a common zero of all polynomials in $J'(p, L)$. Since $(\cos\theta, \sin\theta)$ is generic for the curve $s^2 + t^2 = 1$, it follows that every polynomial in $J'(p, L)$ is in the ideal generated by $s^2 + t^2 - 1$. But since $(1, 0)$ is a zero of $s^2 + t^2 - 1$, it follows that $(1, 0)$ is also a zero of all polynomials in $J'(p, L)$. This implies that $M(u(1, 0))$ accepts L in EQP. As $u(1, 0) \in \{0, \pm 1\}^N$, $M(u(1, 0))$ is deterministic as observed before, and the theorem follows. □

THEOREM 6.4. $\mathrm{BQP}_{poly(1/\epsilon)}, \mathrm{EQP}_{\mathbf{C}}, \mathrm{NQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}} \subseteq \mathrm{PP} \subseteq \mathrm{P}^{\#\mathrm{P}}$.

From Theorems 5.1 and 6.4 we have the following.

COROLLARY 6.5. $\mathrm{BQP}_{\mathbf{C}} \neq \mathrm{EQP}_{\mathbf{C}}$.

We remark that the result $\mathrm{BQP}_{poly(1/\epsilon)} \subseteq \mathrm{P}^{\#\mathrm{P}}$ was first announced by Valiant (see [BV]). The rest of this section is devoted to the proof of Theorem 6.4. We begin by deriving some additional facts about QTMs with real algebraic amplitudes.

Let M be a QTM with real algebraic amplitude; that is, $M \in \mathrm{QTM}_{\bar{\mathbf{Q}} \cap \mathbf{R}}$. Let the local matrix L_δ of M be an $n \times m$ matrix with $\lambda_{i,j}$ as the entry corresponding to the ith row and jth column for $1 \leq i \leq n$ and $1 \leq j \leq m$. Let K be the field generated by all $\lambda_{i,j}$ over $\mathbf{Q}$. Let D be the degree of K over $\mathbf{Q}$. Then $K = \mathbf{Q}[\beta]$ for some $\beta \in \bar{\mathbf{Q}} \cap \mathbf{R}$, and the irreducible polynomial for β over $\mathbf{Q}$ is of degree D. We will call K the *field of amplitudes* for M.

For all $i, j, k \in \mathbf{Z},\ 1 \leq i \leq n,\ 1 \leq j \leq m,\ 0 \leq k \leq D-1$, let $r_{i,j,k} \in \mathbf{Q}$ be the values such that $\lambda_{i,j} = \sum_{k=0}^{D-1} r_{i,j,k}\beta^k$. For all $i, j, k \in \mathbf{Z},\ 0 \leq i \leq D-1,\ 0 \leq j \leq D-1,\ 0 \leq k \leq D-1$, let $s_{i,j,k} \in \mathbf{Q}$ be the values such that $\beta^i\beta^j = \sum_{k=0}^{D-1} s_{i,j,k}\beta^k$. Let $d_1 \in \mathbf{Z}$ be a common denominator of the $r_{i,j,k}$ and $d_2 \in \mathbf{Z}$ be a common denominator of the $s_{i,j,k}$. These values will be used below.

Let $\mathcal{C}$ be the set of all possible configurations of the machine M. Consider running machine M for t steps on input α. We can define a path of length t for input α leading to configuration C as a sequence of t configurations $\langle C_0, C_1, C_2, \ldots, C_t\rangle$ such that C_0 corresponds to the initial configuration for input α, $C_t = C$, and C_{i+1} can be reached from C_i with one step of the machine (including 0 amplitude moves) for $0 \leq i \leq t-1$. Let $P_{\alpha,t,C}$ be the set of paths of length t for input α leading to configuration C. For all paths $p = \langle C_0, C_1, \ldots, C_t\rangle$, there is an associated amplitude $\rho_p = \prod_{k=0}^{t-1} \lambda_{i_{p,k},j_{p,k}}$ where $i_{p,k}$ and $j_{p,k}$ are the row and column in the local matrix that correspond to moving from configuration C_k to configuration C_{k+1}. Let $d_t = d_1^t d_2^{t-1}$, and let $a_{p,i} \in \mathbf{Q}$ be the values such that $\rho_p = \sum_{i=0}^{D-1}(a_{p,i}/d_t)\beta^i$. The following lemma shows that the $a_{p,i}$'s are integers.

LEMMA 6.6. *There exists a $g \in \mathbf{Z}$ such that for all input strings α, for all configurations $C \in \mathcal{C}$, for all $t \in \mathbf{Z}_{>0}$, for all $p \in P_{\alpha,t,C}$, for all $i \in \mathbf{Z}_{\geq 0}^{<D}$, $a_{p,i} \in \mathbf{Z}$, and $abs(a_{p,i}) \leq g^t$.*

Proof of Lemma 6.6.

$$\rho_p = \prod_{k=0}^{t-1} \lambda_{i_{p,k},j_{p,k}} = \prod_{k=0}^{t-1} \sum_{l=0}^{D-1} r_{i_{p,k},j_{p,k},l}\beta^l = \sum_{0\le l_0,\ldots,l_{t-1}\le D-1} \prod_{k=0}^{t-1} r_{i_{p,k},j_{p,k},l_k} \prod_{k=0}^{t-1} \beta^{l_k}.$$

Since d_1 is the common denominator of the $r_{i,j,k}$, d_1^t will be a common denominator of any product of t $r_{i,j,k}$'s. Since d_2 is a common denominator of the $s_{i,j,k}$'s, it is possible to show by induction that d_2^{t-1} is a common denominator of the product of t β^i's. Hence $d_t = d_1^t d_2^{t-1}$ is a common denominator of the sum above and for all paths p and $i \in \mathbf{Z}_0^{D-1}$, $a_{p,i} \in \mathbf{Z}$.

Let

$$m_1 = \max\{abs(r_{i,j,k}) \mid 0 \le i \le n, 0 \le j \le m, 0 \le k \le D-1\}$$

and

$$m_2 = \max\{abs(s_{i,j,k}) \mid 0 \le i,j,k \le D-1\}.$$

Then it can also be shown from the above equation that

$$abs(a_{p,i}) \le d_t D^t m_1^t D^{t-1} m_2^{t-1}.$$

Hence there exists a $g \in \mathbf{Z}$ such that $abs(a_{p,i}) < g^t$. □

We summarize the constants that we have associated with a QTM M with real algebraic amplitudes. Let $K = \mathbf{Q}[\beta] \subseteq \bar{Q} \cap \mathbf{R}$ be the field of amplitudes for M. Let the local matrix of M be an $n \times m$ matrix with $\lambda_{i,j}$ as the (i,j)th entry. Then

$D = D(M)$ be the degree of K over $\mathbf{Q}$;

$d_1 = d_1(M)$ is the least natural number such that $\lambda_{i,j} \in \frac{1}{d_1}\mathbf{Z}[\beta]$ for all i,j with $1 \le i \le n$, $1 \le j \le m$;

$d_2 = d_2(M)$ is the least natural number such that $\beta^i\beta^j \in (1/d_2)\mathbf{Z}[\beta]$ for all i,j with $1 \le i,j \le D$;

$g = g(M)$ is the least natural number determined in Lemma 6.6 such that for all $\alpha \in \mathbf{N}$ and all $t \in \mathbf{Z}_{>0}$ and for all paths p of t steps on input α, the amplitude ρ_p associated with p has the form $\rho_p = \sum_{i=0}^{D-1}(a_{p,i}/d_t)\beta^i$, where $d_t = d_1^t d_2^{t-1}$, $a_{p,i} \in \mathbf{Z}$, and $abs(a_{p,i}) \le g^t$ for all i.

LEMMA 6.7. *For all inputs α, for all $t \in \mathbf{Z}_{>0}$, the following hold.*

1. *If α is accepted in time t with probability* 0, *then*

$$\sum_{i=0}^{D-1} \sum_{C\in\mathcal{C}_\mathcal{A}} \left(\sum_{p\in P_{\alpha,t,C}} a_{p,i} \right)^2 = 0.$$

2. *If α is accepted in time t with probability > 0, then*

$$\sum_{i=0}^{D-1} \sum_{C\in\mathcal{C}_\mathcal{A}} \left(\sum_{p\in P_{\alpha,t,C}} a_{p,i} \right)^2 > 0.$$

Proof of Lemma 6.7. For all configurations, $C \in \mathcal{C}$, for all $t \in \mathbf{Z}_{>0}$, for input strings α, then

$$amp_{\alpha,t,C}(u) = \sum_{p\in P_{\alpha,t,C}} \rho_p = \sum_{p\in P_{\alpha,t,C}} \sum_{i=0}^{D-1} (a_{p_i}/d_t)\beta^i.$$

Switching the sums we get

$$amp_{\alpha,t,C}(u) = \sum_{i=0}^{D-1} \left(\left(\sum_{p \in P_{\alpha,t,C}} a_{p,i} \right) / d_t \right) \beta^i.$$

For all inputs α and $t \in \mathbf{Z}_{>0}$, if input α is accepted in time t with probability 0, then

$$\sum_{C \in \mathcal{C}_A} amp^2_{\alpha,t,C}(u) = 0,$$

and for all $C \in \mathcal{C}_A$, $amp_{\alpha,t,C}(u) = 0$. Hence, for all $i \in \mathbf{Z}_{\geq 0}^{<D}$, $\sum_{p \in P_{\alpha,t,C}} a_{p,i} = 0$, and therefore

$$\sum_{i=0}^{D-1} \sum_{C \in \mathcal{C}_\mathcal{A}} \left(\sum_{p \in P_{\alpha,t,C}} a_{p,i} \right)^2 = 0,$$

which proves case 1.

For all inputs α and $t \in \mathbf{Z}_{>0}$, if input α is accepted in time t with probability > 0, then

$$\sum_{C \in \mathcal{C}_A} amp^2_{\alpha,t,C}(u) > 0,$$

so there exists a $C \in \mathcal{C}_A$ such that $amp_{\alpha,t,C}(u) \neq 0$, which implies that there exists an $i \in \mathbf{Z}_{\geq 0}^{<D}$ such that $\sum_{p \in P_{\alpha,t,C}} a_{p,i} \neq 0$, and, therefore,

$$\sum_{i=0}^{D} \sum_{C \in \mathcal{C}_\mathcal{A}} \left(\sum_{p \in P_{\alpha,t,C}} a_{p.i} \right)^2 > 0,$$

which proves case 2. □

Theorem 6.4 will be proved by the three lemmas given below.

LEMMA 6.8. $\mathrm{NQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}} \subseteq \mathrm{PP} \subseteq \mathrm{P}^{\#\mathrm{P}}$.

Proof of Lemma 6.8. For all $S \subseteq \mathbf{N}$ such that $S \in \mathrm{NQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}}$, we will show that there exists a nondeterministic Turing machine M' that for all $x \in \mathbf{N}$ the following hold:

if $x \in S$, then M' on input x says "yes" on more than 1/2 its paths;

if $x \in \overline{S}$, then M' on input x says "no" on more than 1/2 its paths.

It will follow that $S \in \mathrm{PP}$ and hence $\mathrm{NQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}} \subseteq \mathrm{PP} \subseteq \mathrm{P}^{\#\mathrm{P}}$.

From the definition of $\mathrm{NQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}}$, there exist a $c \in \mathbf{N}_{>0}$ and an M membership $\mathrm{QTM}_{\bar{\mathbf{Q}} \cap \mathbf{R}}$ such that for all $x \in \mathbf{N}$ the following hold:

$x \in S \rightarrow$ after $|x|^c$ steps M on input x accepts with probability > 0;

$x \in \overline{S} \rightarrow$ after $|x|^c$ steps M on input x rejects with probability 0.

Let $K = \mathbf{Q}[\beta] \subseteq \bar{Q} \cap \mathbf{R}$ be the field of amplitudes for M. Let $n = n(M)$, $D = D(M)$, $d_1 = d_1(M)$, $d_2 = d_2(M)$, and $g = g(M)$ be the constants associated with M as defined before.

We observe that for all $x \in \mathbf{N}$, for all $t \in \mathbf{Z}_{>0}$, a path $< C_0, \ldots, C_t >$ of M of length t on input x can be specified by a natural number $P < n^t$ as follows. Write $P = \sum_{i=0}^{t-1} m_i n^i$ with $0 \leq m_i \leq n-1$. Inductively for $i = 0, \ldots, t-1$, from C_i one

chooses the $(m_i + 1)$st entry from the corresponding column in the local matrix for transition to determine the next configuration C_{i+1}. We shall identify P with the path determined by it on input x, and let ρ_P denote the amplitude associated with P. Let $d_t = d_1^t d_2^{t-1}$. Then from Lemma 6.6, $\rho_P = \sum_{i=0}^{D-1} (a_{P,i}/d_t)\beta^i$, where $a_{P,i} \in \mathbf{Z}$, and $abs(a_{P,i}) \leq g^t$ for all i. We will call $a_{P,i}$ the *ith integral coefficient* of ρ_P for $i = 0, \ldots, D-1$.

Let M' on input x proceed as follows.

1. Compute $t = |x|^c$.

2. Guess seven numbers $\langle P_1, G_1, P_2, G_2, E, i, F\rangle$ ("$\langle$" denotes any "reasonable" pairing function) where $P_1, P_2 \in \mathbf{Z}_{\geq 0}^{\leq n^t}$, $G_1, G_2 \in \mathbf{Z}_{\geq 0}^{\leq g^t+1}$, E, F are 0 or 1 and $i \in \mathbf{Z}_{\geq 0}^{<D}$.

3. Use P_1 to choose a path of length t for M on input x. Compute the configuration C_1, which results along this path, and $a_{P_1,i}$, the ith integral coefficient of the amplitude ρ_{P_1} associated with P_1. Similarly, use P_2 to choose a second path of length t for M on input x, and compute C_2 and $a_{P_2,i}$.

4. (trivial guesses)

(i) If $C_1 \neq C_2$, then output "yes" if $E = 0$ and "no" if $E = 1$.

(ii) If $C_1 \in \mathcal{C}_{\mathcal{R}}$, then output "yes" if $E = 0$ and "no" if $E = 1$, except when $P_1 = P_2 = E = i = F = 0$ and $G_1 = G_2 = g^t + 1$.

(iii) If $G_1 \geq abs(a_{P_1,i})$ or $G_2 \geq abs(a_{P_2,i})$, then output "yes" if $E = 1$ and "no" if $E = 0$, except when $P_1 = P_2 = E = i = F = 0$ and $G_1 = G_2 = g^t + 1$.

5. Else (nontrivial guesses)

(i) If $P_1 = P_2 = E = i = F = 0$ and $G_1 = G_2 = g^t + 1$, output "no."

(ii) If $a_{P_1,i}$ and $a_{P_2,i}$ have the same sign, then output "yes."

(iii) If $a_{P_1,i}$ and $a_{P_2,i}$ have opposite signs, then output "no."

Any path with $G_1 = g^t + 1$ or $G_2 = g^t + 1$ is trivial since $G_1 \geq abs(a_{P_1,i})$ or $G_2 \geq abs(a_{P_2,i})$, respectively, except for the special case when $P_1 = P_2 = E = i = F = 0$ and $G_1 = G_2 = g^t + 1$. The number of trivial guesses plus this additional special guess give two more "no" outputs than "yes" outputs.

For all configurations $C \in \mathcal{C}_{\mathcal{A}}$ which can be reached with t steps of M (including transitions that have amplitude of 0), for all $P_1, P_2 \in P_{x,t,C}$, for all $i \in \mathbf{Z}_{\geq 0}^{<D}$, for all $G_1 \in \mathbf{Z}_{\geq 0}^{<abs(a_{P_1,i})}$, for all $G_2 \in \mathbf{Z}_{\geq 0}^{<abs(a_{P_2,i})}$, the guess is nontrivial. In all other cases (except one) the guess will be trivial.

If we count the number of "yes" outputs minus the number of "no" outputs we get

$$\left(4 \sum_{C \in \mathcal{C}_{\mathcal{A}}} \sum_{i=0}^{D-1} \sum_{P_1 \in P_{x,t,C}} \sum_{P_2 \in P_{x,t,C}} a_{P_1,i} a_{P_2,i}\right) - 2,$$

since for a fixed set of paths P_1, P_2 leading to the same configuration, there are $abs(a_{P_1,i} a_{P_2,i})$ sets of pairs G_1, G_2 for each value of E and F.

The above equation can be rewritten as

$$\left(4 \sum_{C \in \mathcal{C}_{\mathcal{A}}} \sum_{i=0}^{D-1} \left(\sum_{p \in P_{x,t,C}} a_{p,i}\right)^2\right) - 2.$$

If $x \in S$ then x is accepted with positive probability, so from part 2 of Lemma 6.7 the sum is greater than 0, and there are more "yes" outputs than "no" outputs.

If $x \in \overline{S}$ then x is accepted with probability 0, so from part 1 of Lemma 6.7 this sum is -2 and there are more "no" outputs than "yes" outputs. □

LEMMA 6.9. $\mathrm{EQP}_{\mathbf{C}} \subseteq \mathrm{PP} \subseteq \mathrm{P}^{\#\mathrm{P}}$.

Proof of Lemma 6.9. By Theorem 6.2, $\mathrm{EQP}_{\mathbf{C}} = \mathrm{EQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}}$. Since $\mathrm{EQP}_K \subseteq \mathrm{NQP}_K$ for any field K, it follows from Lemma 6.8 that

$$\mathrm{EQP}_{\mathbf{C}} = \mathrm{EQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}} \subseteq \mathrm{NQP}_{\bar{\mathbf{Q}} \cap \mathbf{R}} \subseteq \mathrm{PP} \subseteq \mathrm{P}^{\#\mathrm{P}}. \qquad \square$$

LEMMA 6.10. $\mathrm{BQP}_{poly(1/\epsilon)} \subseteq \mathrm{PP} \subseteq \mathrm{P}^{\#\mathrm{P}}$.

Proof of Lemma 6.10. From section 3, $\mathrm{BQP}_{poly(1/\epsilon)} = \mathrm{BQP}_\theta$, where $\cos(\theta) = 3/5$.

For all $S \subseteq \mathbf{N}$ such that $S \in \mathrm{BQP}_\theta$, we will show that there exists a nondeterministic Turing machine M' that for all $x \in \mathbf{N}$ the following hold:

if $x \in S$, then M' on input x says "yes" on more than $1/2$ its paths;

if $x \in \overline{S}$, then M' on input x says "no" on more than $1/2$ its paths.

It will follow that $S \in \mathrm{PP}$ and hence $\mathrm{BQP}_{poly(1/\epsilon)} = BQP_\theta \subseteq \mathrm{PP} \subseteq \mathrm{P}^{\#\mathrm{P}}$.

From the definition of BQP_θ, there exists a $c \in \mathbf{N}_{>0}$ and M a QTM_θ such that for all $x \in \mathbf{N}$ the following hold:

$x \in S \rightarrow$ after $|x|^c$ steps M on input x accepts with probability $> 2/3$,

$x \in \overline{S} \rightarrow$ after $|x|^c$ steps M on input x rejects with probability $> 2/3$.

Let $n = n(M)$, $D = D(M)$, $d_1 = d_1(M)$, $d_2 = d_2(M)$, and $g = g(M)$ be the constants associated with M as defined before.

Let M' on input x proceed as follows.

1. Compute $t = |x|^c$.

2. Guess five numbers $\langle P_1, G_1, P_2, G_2, E \rangle$ ("$\langle$" denotes any "reasonable" pairing function) where $P_1, P_2 \in \mathbf{Z}_{\geq 0}^{\leq n^t}$, $G_1, G_2 \in \mathbf{Z}_{\geq 0}^{\leq g^t}$, and E is 0 or 1.

3. Use P_1 to choose a path of length t for M on input x. Compute the configuration C_1, which results along this path, and $a_{P_1,i}$, the ith integral coefficient of the amplitude ρ_{P_1} associated with P_1. Similarly, use P_2 to choose a second path of length t for M on input x, and compute C_2 and $a_{P_2,i}$.

4. (trivial guesses)

(i) If $C_1 \neq C_2$, then output "yes" if $E = 0$ and "no" if $E = 1$.

(ii) If $G_1 \geq abs(a_{P_1,0})$ or $G_2 \geq abs(a_{P_2,0})$, then output "yes" if $E = 0$ and "no" if $E = 1$.

5. Else (nontrivial guesses)

(i) If $a_{P_1,0}$ and $a_{P_2,0}$ have the same sign, then output "yes" if C_1 is an accept configuration and "no" if C_1 is a reject configuration.

(ii) If $a_{P_1,0}$ and $a_{P_2,0}$ have opposite signs, then output "yes" if C_1 is a reject configuration and "no" if C_1 is an accept configuration.

Since the entries in the local matrix are rational, the degree of $K = \mathbf{Q}$ equals 1, and so

$$amp_{\alpha,t,C}(u) = \sum_{p \in P_{\alpha,t,C}} a_{p,0}/d_t.$$

Again all the trivial paths have equal "yes" and "no" results.

For all configurations $C \in \mathcal{C}$ which can be reached with t steps of M (including transitions that have amplitude of 0), for all $P_1, P_2 \in P_{x,t,C}$, for all $G_1 \in \mathbf{Z}_{\geq 0}^{<abs(a_{P_1,0})}$, for all $G_2 \in \mathbf{Z}_{\geq 0}^{<abs(a_{P_2,0})}$, the guess is nontrivial. In all other cases the guess will be trivial.

If we count the number of "yes" outputs minus the number of "no" outputs we get

$$2\left(\sum_{C\in\mathcal{C}_{\mathcal{A}}}\sum_{P_1\in P_{x,t,C}}\sum_{P_2\in P_{x,t,C}}a_{P_1,0}a_{P_2,0}-\sum_{C\in\mathcal{C}_{\mathcal{R}}}\sum_{P_1\in P_{x,t,C}}\sum_{P_2\in P_{x,t,C}}a_{P_1,0}a_{P_2,0}\right),$$

since for a fixed set of paths P_1, P_2 leading to the same configuration, there are $abs(a_{P_1,0}a_{P_2,0})$ sets of pairs G_1, G_2 for each value of E.

The above equation can be rewritten as

$$2\left(\sum_{C\in\mathcal{C}_{\mathcal{A}}}\left(\sum_{p\in P_{x,t,C}}a_{p,i}\right)^2-\sum_{C\in\mathcal{C}_{\mathcal{R}}}\left(\sum_{p\in P_{x,t,C}}a_{p,i}\right)^2\right)$$

$$=2d_t^2\left(\sum_{C\in\mathcal{C}_{\mathcal{A}}}\left(amp^2_{\alpha,t,C}(u)\right)-\sum_{C\in\mathcal{C}_{\mathcal{R}}}\left(amp^2_{\alpha,t,C}(u)\right)\right).$$

If $x \in S$, then x is accepted with probability greater than $2/3$ and rejected with probability less than $1/3$ and so the above sum is greater than 0, giving more paths in M' with a "yes" result than a "no" result.

If $x \in \overline{S}$, then x is rejected with probability greater than $2/3$ and accepted with probability less than $1/3$ and so the above sum is less than 0, giving more paths in M' with a "no" result than a "yes" result.

This concludes the proof of Lemma 6.10 and therefore Theorem 6.4. □

7. Discussion. If one is concerned about minimizing (or precisely calculating) the polynomial slowdown in the approximations implicit in Theorem 3.3, then more recent results of Baker and others on transcendental numbers may be useful. Angles θ with $\theta/2\pi \in \bar{\mathbf{Q}} - \mathbf{Q}$ may be particularly good in this regard.

There are many unsettled issues and open questions involving quantum computation. For example, letting K be a field, is $\mathrm{BQP}_K = \mathrm{EQP}_K$? We have shown that $\mathrm{BQP}_{\mathbf{C}} \neq \mathrm{EQP}_{\mathbf{C}}$. It seems unlikely that $\mathrm{BQP}_{\mathbf{Q}} = \mathrm{EQP}_{\mathbf{Q}}$.

One can ask whether $\mathrm{NQP}_K = \mathrm{EQP}_K$, an analogue of the $NP = P$ question.

One can consider the relationship between classical classes and quantum classes. For example does EQP=P? We suspect not, but since EQP$\subseteq$ PSPACE, a yes answer would not be entirely out of the question. Does $\mathrm{BQP}_{\mathbf{Q}}$=BPP?

One can generalize the notion of a "classical" probabilistic Turing machine to allow for amplitudes in $\mathbf{C}$ and consider the relationship between natural classes on a classical probabilistic Turing machine and on a quantum machine. For example, is it the case that $\mathrm{BQP}_{\mathbf{C}} = \mathrm{BPP}_{\mathbf{C}}$ (when BPP_K is appropriately defined)? Does $\mathrm{BQP}_{\mathbf{Q}} = \mathrm{BPP}_{\mathbf{Q}}$? An affirmative answer would inform us that the power of quantum computation is not found in the use of the L_2-norm but rather in the use of general (possibly negative) amplitudes.

One can ask similar questions for QTM_θ rather than QTM_K. For example, let θ and θ' have cos = 3/5 and 5/12, respectively. Then $\mathrm{QTM}_\theta \asymp \mathrm{QTM}_{\theta'}$ and hence $\mathrm{BQP}_\theta = \mathrm{BQP}_{\theta'}$; however, does $\mathrm{EQP}_\theta = \mathrm{EQP}_{\theta'}$? If $\theta/2\pi \notin \mathbf{Q}$, does EQP_θ =P?

To what extent would demonstrating relationships between various complexity classes over various fields (or with various angles) have implications similar to those which arise when relationships between classical classes are demonstrated with respect to an oracle?

One could consider very general notions of machines (amplitudes in $\mathbf{C}$, arbitrary norms, for example). Would an investigation of these shed light on the basic open problems of computational complexity?

On a more concrete level, is the "shortest vector in a lattice" problem in $\mathrm{BQP}_{\mathbf{Q}}$ (this has been asked by numerous researchers)? Is primality in $\mathrm{EQP}_{\mathbf{C}}$? Is primality in $\mathrm{EQP}_{\mathbf{Q}}$? Can integers be multiplied in linear time on a QTM?

Acknowledgments. We thank Harold Stark for providing us with Lemma 3.6, Charles Bennett for contributing to our understanding of methods of approximating θ used in the proof of Theorem 3.3, and Don Coppersmith for enlightening comments regarding a previous version of this paper. Finally, we thank the anonymous referees for bringing Proposition 6.1 to our attention, for pointing out a mistake in an earlier version, and for many valuable comments.

REFERENCES

[B] A. Baker, *Transcendental Number Theory*, Cambridge University Press, London, 1979.

[Be] C. H. Bennett, *Logical reversibility of computation*, IBM J. Res. Develop., 17 (1973), pp. 525–532.

[BBBV] C. Bennett, E. Bernstein, G. Brassard, and U. Vazirani, *Strengths and weaknesses of quantum computing*, SIAM J. Comput., 26(1997), pp. 1510–1523.

[BV] E. Bernstein and U. Vazirani, *Quantum complexity theory*, in Proc. 25th ACM Symposium on Theory of Computation, San Diego, CA, 1993, pp. 11–20; SIAM J. Comput., 26 (1997), pp. 1411–1473.

[D1] D. Deutsch, *Quantum theory, the Church–Turing principle and the universal quantum computer*, Proc. Roy. Soc. London Ser. A, 400 (1985), pp. 96–117.

[D2] D. Deutsch, *Quantum computational networks*, Proc. Roy. Soc. London Ser. A., 425 (1989), pp. 73–90.

[DJ] D. Deutsch and R. Jouzsa, *Rapid solution of problems by quantum computation*, Proc. Roy. Soc. London Ser. A, 439 (1992), pp. 553–558.

[F] R. Feynman, *Simulating physics with computers*, Internat. J. Theoret. Phys., 21 (1982), pp. 467–488.

[Fe] N. I. Feldman, *An improvement of the estimate of a linear form in the logarithms of algebraic numbers*, Mat. Sb. (N.S.), 77 (119) (1968), pp. 423–436 (in Russian).

[J] N. Jacobson, *Basic Algebra* I, W. H. Freeman, San Francisco, 1980.

[Ni] I. Niven, *Irrational Numbers*, The Mathematics Association of America, Rahway, NJ, 1956.

[Sh] P. Shor, *Algorithms for quantum computation: Discrete log and factoring*, in Proc. 35th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 124–134.

[Si] D. Simon, *On the power of quantum computation*, SIAM J. Comput., 26 (1997), pp. 1474–1483.

[So] R. Solovay, *Virtual Reading Group Communication*, Internet, 14 August 1994.

[Y] A. Yao, *Quantum circuit complexity*, in Proc. 34th IEEE Symposium on Foundations of Computer Science, Palo Alto, CA, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 352–361.

SIAM J. COMPUT.
Vol. 26, No. 5, pp. 1541–1557, October 1997

012

STABILIZATION OF QUANTUM COMPUTATIONS BY SYMMETRIZATION *

ADRIANO BARENCO†, ANDRÉ BERTHIAUME‡, DAVID DEUTSCH§, ARTUR EKERT¶, RICHARD JOZSA‖, AND CHIARA MACCHIAVELLO**

Abstract. We propose a method for the stabilization of quantum computations (including quantum state storage). The method is based on the operation of projection into $\mathcal{SYM}$, the symmetric subspace of the full state space of R redundant copies of the computer. We describe an efficient algorithm and quantum network effecting $\mathcal{SYM}$–projection and discuss the stabilizing effect of the proposed method in the context of unitary errors generated by hardware imprecision, and nonunitary errors arising from external environmental interaction. Finally, limitations of the method are discussed.

Key words. quantum computation, error correction

AMS subject classifications. 68, 81

PII. S0097539796302452

1. Introduction.

Any realistic model of computation must conform to certain requirements imposed not by the mathematical properties of the model but by the laws of physics. Computations which require an exponentially increasing precision or exponential amount of time, space, energy, or any other physical resource are normally regarded as unrealistic and intractable.

Any actual computational process is subject to unavoidable hardware imprecision and spurious interaction with the environment, whose nature is dictated by the laws of physics. These effects introduce errors and destabilize the progress of the desired computation. It is, therefore, essential to have some method of stabilizing the computation against these effects.

For classical computation there is a simple and highly effective method of stabilization. Each computational variable is represented redundantly using many more physical degrees of freedom than are logically required, and a majority vote (or average) of all the copies is taken followed by resetting all the copies to the majority

* Received by the editors April 22, 1996; accepted for publication (in revised form) December 2, 1996. Part of this work was carried out during the Quantum Computation Workshops, conducted with the support of ELSAG-Bailey, Genova, and the Institute for Scientific Interchange, Torino. This work was also partly supported by the European Commission TMR Network grant FMRX-CT96-0087.

http://www.siam.org/journals/sicomp/26-5/30245.html

† Clarendon Laboratory, University of Oxford, Parks Road, Oxford OX1 3PU, U.K. (barenco@mildred.physics.ox.ac.uk). The research of this author was supported by Berrow's Fund at Lincoln College, Oxford.

‡ AT&T Labs–Research, 600 Mountain Avenue, Murray Hill, NJ 07974 (berthiau@research.att.com).

§ Clarendon Laboratory, University of Oxford, Parks Road, Oxford OX1 3PU, U.K. (deutsch@mildred.physics.ox.ac.uk).

¶ Clarendon Laboratory, University of Oxford, Parks Road, Oxford OX1 3PU, U.K. (ekert@mildred.physics.ox.ac.uk). The research of this author was supported by the Royal Society, London.

‖ School of Mathematics and Statistics, University of Plymouth, Plymouth, Devon PL4 8AA, U.K. (rjozsa@plymouth.ac.uk). The research of this author was supported by the Royal Society, London.

** Clarendon Laboratory, University of Oxford, Parks Road, Oxford OX1 3PU, U.K. (chiara@mildred.physics.ox.ac.uk). The research of this author was supported by the European HCM Programme.

answer. This process is applied periodically during the course of the computation. If we use R copies and the probability of producing the correct answer is $\frac{1}{2}+\eta$ then it can be shown ([1, p. 258]) that the probability E that the majority vote is wrong, is less than $\exp(-\eta^2 R/6)$. This is an extremely resource-efficient stabilization in that the probability of error decreases *exponentially* with the degree of redundancy R. Indeed, suppose that a polynomial-time algorithm runs for M steps, each of which is correct with probability $\frac{1}{2}+\eta$ and majority voting is used after each step. The probability that the final answer will be correct is greater than $(1-E)^M$. Thus any desired success probability $1-\delta$ may be achieved using a degree of redundancy $R = O(\log(M/\delta))$, which is only logarithmic in the input size.

The majority vote method just described cannot be applied in the case of quantum computation because quantum algorithms depend essentially on the maintenance of coherent superpositions of different computational states at each step. The laws of quantum mechanics forbid the identification of an unknown quantum state [17], [18] and forbid even the cloning of an unknown state [19]. Thus the majority voting method is inapplicable as we can neither determine the majority state nor reset the remaining copies to that state. In this paper we propose an alternative quantum mechanical method of stabilization which utilizes redundancy but which has no classical analogue. We discuss its applicability and limitations. The method was first proposed by Deutsch [2] and a brief outline of its underlying principles was given in [3].

An alternative approach to the stabilization of classical computation involves the use of error correcting codes [20]. A quantum mechanical generalization of this approach was recently introduced by Shor [9] and subsequently developed in [13], [10], [12], [11]. These methods are unrelated to those proposed in this paper and provide an interesting supplementary method of stabilizing quantum computation.

The process of simply repeating a whole computation a sufficient number of times may serve to stabilize it in certain circumstances. Suppose that we have a quantum algorithm which succeeds with probability $1-\epsilon$ (where ϵ may increase with input size) and suppose that we know when the computation has been successful. For example, the computation may produce a candidate factor of an input integer which can then be efficiently checked by trial division. For any input size L the success probability can be amplified to any prescribed level $1-\delta$ by repeating the algorithm a sufficiently large number, R, of times since the probability of at least one success in R repetitions is $1-\epsilon^R \to 1$ as $R \to \infty$. Suppose now that the success probability $1-\epsilon$ decreases with input size L as $1/\mathrm{poly(L)}$. Then we can maintain any prescribed success probability by allowing R to increase as a suitable polynomial function of L. Thus if the original algorithm was efficient (i.e., polynomial time) then its R-fold repetition is still efficient, i.e., the algorithm has been stabilized in an efficient manner. (Note that Shor's quantum factoring algorithm [7], [8] is of this type with success probability decreasing as $1/L$ with input size.) However, if the success probability falls *exponentially* with input size L then we must use $R \sim \exp(\mathrm{L})$ to maintain any constant level of success probability. This implies an exponential increase of physical resources for stabilization and hence this method is inefficient in this case. Unfortunately, just such an exponential decay of success probability appears to be a generic feature of any physical implementation of computation, as described below.

In quantum theory, the issue of preventing information from leaking into the environment from a system ("the computer") is generally known as the decoherence problem [4], [5], [6]. According to the analysis of [6], decoherence generally causes an *exponential* decrease in success probability with input size L. Decoherence is

a universal phenomenon and is expected to affect—to some extent—any physical implementation of quantum computation whatever. Thus without some *efficient* form of stabilization, quantum algorithms which are polynomially efficient in the error-free case (like Shor's factoring algorithm) cannot be considered polynomially efficient in practice.

Consider any efficient computation which gives the correct result at each step with probability $1-\epsilon$ where ϵ is constant. This would typically be the case if each step of a computation consisted of the application of an elementary gate operation having a standard tolerance of error. Then after N steps the probability of success is (at least) $(1-\epsilon)^N \sim \exp(-\epsilon N)$, which again decreases exponentially with N. Suppose that we have a stabilization scheme utilizing redundancy R which reduces the error in each step only by a factor $1/R$, i.e., $\epsilon \to \epsilon/R$ (rather than the *exponential* decrease given by classical majority voting). After N steps the probability of success is now $\exp(-\epsilon N/R)$. This can be kept at any prescribed level $1-\delta$ by taking $R = \epsilon N/(-\log(1-\delta))$ which is *polynomial* in N and hence in the input size L. Thus an exponentially growing error (such as results from decoherence) in a polynomial-time computation can be efficiently stabilized by a method which reduces the error per step only as $1/R$ with the degree of redundancy. Our proposed method below will have this property.

An essential ingredient in our stabilization method is the so-called "quantum watched pot" effect (or quantum Zeno effect) [16]. Our method will require the repeated projection of the quantum state of R computers into the symmetric subspace $\mathcal{SYM}$, a subspace of the total state space of the R computers. This projection has a nonzero failure probability so that (in view of the previous paragraph) the cumulative probability of repeated successful projection may be expected to fall exponentially with the number of projections. The quantum watched pot effect provides a means of maintaining the cumulative probability of successful projection arbitrarily close to unity. The basic principle is illustrated in the following simplified example. Consider a quantum system initially in state $|0\rangle$ which rotates into $|1\rangle$ with angular frequency ω. The state at time t (in the absence of any projections) is $\cos\omega t\,|0\rangle + \sin\omega t\,|1\rangle$. If we project this state into Λ_0, the subspace spanned by $|0\rangle$, then the probability of successful projection is $\cos^2\omega t$. If we project repeatedly n times between $t=0$ and $t=1$, i.e., at time intervals $\delta t = 1/n$, then the probability that all projections will be successful is

$$\left(\cos^2\frac{\omega}{n}\right)^n \approx \left(1-\frac{\omega^2}{n^2}\right)^n \to 1 \qquad \text{as} \qquad n\to\infty.$$

Thus if the projections are performed with sufficient frequency then the state may be confined to the subspace Λ_0 with arbitrarily high probability. This is the quantum watched pot effect. In quantum mechanics projection operations correspond to measurements on the system so the above may be loosely phrased as "a frequently observed state never evolves" or "a watched pot never boils," giving the origin of the terminology. A similar analysis holds for Λ_0 replaced by any subspace such as $\mathcal{SYM}$, and for any unitary evolution of a state initially lying in the subspace as elaborated in section 5.

It will be useful in the following to keep in mind the simplest possible example of the stabilization problem where the computer consists of one qubit (i.e., one two-level system) and is performing no computation. In fact this simple model captures the essential features of the stabilization problem for general quantum computations.

The problem of stabilization concerns the time evolution of an "accuracy" observable which has only two eigenvalues. As we shall see our analysis of error correction depends only on such simple observables and is independent of the substance of the computation. Thus we are addressing the problem of stabilizing the *storage* of an (unknown) quantum state of one qubit against environmental interaction and (suitably random) imprecision in the construction of the storage device.

2. The symmetric subspace. Our proposed stabilization method will exploit redundancy but in contrast to the classical majority voting method, it will be based on essentially quantum mechanical properties through use of the symmetric subspace of the full state space of R copies of a physical system. Consider R copies of a quantum system each with state space $\mathcal{H}$. Denote the full state space $\mathcal{H} \otimes \mathcal{H} \otimes \ldots \mathcal{H}$ by $\mathcal{H}^R$.

Remark 1. Here we require that the R copies be distinguishable, e.g., being located in separate regions of space so that the position coordinate provides an extra "external" degree of freedom for distinguishability. The state space $\mathcal{H}$ can be thought of as representing the "internal" degrees of freedom of each system. In our application these are the computational degrees of freedom of each replica of the computer. In our notation we suppress explicit mention of the distinguishing degree of freedom which is implicitly given by the written order of component states in a tensor product state (cf. *Remark* 2 below).

The symmetric subspace $\mathcal{SYM}$ of $\mathcal{H}^R$ may be characterized by either of the two following equivalent definitions.

DEFINITION 1. *$\mathcal{SYM}$ is the smallest subspace of $\mathcal{H}^R$ containing all states of the form $|\psi\rangle |\psi\rangle \ldots |\psi\rangle$ for all $|\psi\rangle \in \mathcal{H}$.*

DEFINITION 2. *$\mathcal{SYM}$ is the subspace of all states in $\mathcal{H}^R$ which are symmetric (i.e., unchanged) under the interchange of states for any pair of positions in the tensor product. (Here we interchange only the internal degrees of freedom leaving the external degrees fixed.)*

Remark 2. To clarify the notion of symmetrization in Definition 2 note that, for example, $|\phi\rangle |\psi\rangle + |\psi\rangle |\phi\rangle \in \mathcal{H}^2$ is in $\mathcal{SYM}$. If we were to show the external degrees of freedom then this state would be written $|\phi; x_1\rangle |\psi; x_2\rangle + |\psi; x_1\rangle |\phi; x_2\rangle$. Consequently, the notion of symmetrization in Definition 2 is different from bosonic symmetrization which requires symmetrization of *all* degrees of freedom. For a pair of bosons the previous state would be $|\phi; x_1\rangle |\psi; x_2\rangle + |\psi; x_2\rangle |\phi; x_1\rangle$.

Definition 1 has the following interpretation. Suppose that we have R copies of a quantum computer. If there were no errors then at each time the joint state would have the form

$$|\psi\rangle |\psi\rangle \ldots |\psi\rangle \in \mathcal{H}^R. \tag{1}$$

In the presence of errors the states will evolve differently resulting in a joint state of the form $|\psi_1\rangle |\psi_2\rangle \ldots |\psi_R\rangle$ or more generally a mixture of superpositions of such states. In quantum mechanics any test ("yes/no" question) that we can apply to a physical system must correspond to a *subspace* of the total state space. States of the form (1) for all $|\psi\rangle \in \mathcal{H}$ do not, by themselves, form a subspace of $\mathcal{H}^R$. According to Definition 1, $\mathcal{SYM}$ is the smallest subspace containing all possible error-free states. It thus corresponds to the "most probing" test we can legitimately apply, which will be passed by all error-free states. Recall that we cannot generally identify the actual quantum state during the course of the computation or indeed gain any information about it without causing some irreparable disturbance [18]. The characterization

given in Definition 2 is especially useful in treating mathematical properties of $\mathcal{SYM}$ as follows.

The equivalence of the two definitions may be proved by viewing the ith component space in the tensor product $\mathcal{H}^R$ as the space of complex polynomials of degree $\leq n-1$ in the variable x_i (where n is the dimension of $\mathcal{H}$). By the fundamental theorem of algebra any such polynomial may be factored as $(x_i-\alpha_1)(x_i-\alpha_2)\dots(x_i-\alpha_{n-1})$. Then Definition 1 defines the subspace of all polynomials $p(x_1,\dots,x_R)$ of degree $\leq n-1$ in each variable, which arise as sums of products of functions of the form

$$f_\alpha(x_1,x_2,\dots,x_R)=\prod_{i=1}^{R}(x_i-\alpha) \tag{2}$$

for any α. On the other hand, Definition 2 defines the space of all symmetric polynomials (of degree $\leq n-1$ in each variable). The equivalence of these subspaces then follows easily from basic properties of the standard elementary symmetric functions [14], which are defined as the coefficients of the powers of α in the expansion of (2).

The equivalence of the two definitions may also be understood via the following illustrative example which gives further insight into the structure of $\mathcal{SYM}$.

Example 1. Suppose that $\mathcal{H}$ is two-dimensional (i.e., a qubit) with computational basis states $|0\rangle$ and $|1\rangle$. Consider triple redundancy $R=3$ and the symmetric subspace $\mathcal{SYM}\subset\mathcal{H}^3$. Let us tentatively denote the symmetric subspaces of Definitions 1 and 2 by $\mathcal{SYM}_{def1}$ and $\mathcal{SYM}_{def2}$, respectively. We wish to show that these coincide. Note first that $\mathcal{SYM}_{def1}$ is the span of all states of the form $|\psi\rangle|\psi\rangle|\psi\rangle$, which are clearly symmetrical in the sense of Definition 2. Hence, $\mathcal{SYM}_{def1}\subseteq\mathcal{SYM}_{def2}$. For the reverse inclusion consider a general state in $\mathcal{H}^3$:

$$\begin{aligned}|\alpha\rangle = {} & a_0|0\rangle|0\rangle|0\rangle \\ & +a_1|1\rangle|0\rangle|0\rangle+a_2|0\rangle|1\rangle|0\rangle+a_3|0\rangle|0\rangle|1\rangle \\ & +a_4|1\rangle|1\rangle|0\rangle+a_5|1\rangle|0\rangle|1\rangle+a_6|0\rangle|1\rangle|1\rangle \\ & +a_7|1\rangle|1\rangle|1\rangle\,.\end{aligned} \tag{3}$$

Interchange of states for any given pair of positions (in the sense of Definition 2) preserves the number of $|0\rangle$'s and $|1\rangle$'s in each term so that $|\alpha\rangle$ will be in $\mathcal{SYM}_{def2}$ if and only if $a_1=a_2=a_3$ and $a_4=a_5=a_6$. Indeed, we see that $\mathcal{SYM}_{def2}$ is four dimensional with orthonormal basis states (labelled by the number of $|1\rangle$'s):

$$\begin{aligned}|e_0\rangle &= |0\rangle|0\rangle|0\rangle \\ |e_1\rangle &= (|1\rangle|0\rangle|0\rangle+|0\rangle|1\rangle|0\rangle+|0\rangle|0\rangle|1\rangle)/\sqrt{3} \\ |e_2\rangle &= (|1\rangle|1\rangle|0\rangle+|1\rangle|0\rangle|1\rangle+|0\rangle|1\rangle|1\rangle)/\sqrt{3} \\ |e_3\rangle &= |1\rangle|1\rangle|1\rangle\,.\end{aligned} \tag{4}$$

(The four normalizing factors $1,\sqrt{3},\sqrt{3},$ and 1 are square roots of the binomial coefficients ${}^3C_0,{}^3C_1,{}^3C_2,{}^3C_3$.) Now for any $|\psi_1\rangle$ of the form $a|0\rangle+|1\rangle$ we get directly that

$$|\psi_1\rangle|\psi_1\rangle|\psi_1\rangle=a^3|e_0\rangle+a^2\sqrt{3}|e_1\rangle+a\sqrt{3}|e_2\rangle+|e_3\rangle\,.$$

Repeating this for four different values of the parameter a we get the following:

$$\begin{pmatrix} a^3 & a^2 & a & 1 \\ b^3 & b^2 & b & 1 \\ c^3 & c^2 & c & 1 \\ d^3 & d^2 & d & 1 \end{pmatrix} \begin{pmatrix} |e_0\rangle \\ \sqrt{3}\,|e_1\rangle \\ \sqrt{3}\,|e_2\rangle \\ |e_3\rangle \end{pmatrix} = \begin{pmatrix} |\psi_1\rangle\,|\psi_1\rangle\,|\psi_1\rangle \\ |\psi_2\rangle\,|\psi_2\rangle\,|\psi_2\rangle \\ |\psi_3\rangle\,|\psi_3\rangle\,|\psi_3\rangle \\ |\psi_4\rangle\,|\psi_4\rangle\,|\psi_4\rangle \end{pmatrix}.$$

Choosing $a, b, c,$ and d so that the coefficient matrix is invertible, we see that the basis states (4) are all in $\mathcal{SYM}_{def1}$ so that $\mathcal{SYM}_{def2} \subseteq \mathcal{SYM}_{def1}$. Hence, these subspaces coincide.

From the above considerations (cf. especially (4)) we readily see that the dimension of $\mathcal{SYM}$ for R qubits is $R+1$ so that $\mathcal{SYM}$ is an exponentially small subspace of $\mathcal{H}^R$ (of dimension 2^R). This is also true in the general case. Suppose that $\mathcal{H}$ has dimension d with orthonormal basis $|0\rangle, |1\rangle \ldots |d-1\rangle$. Then $\mathcal{SYM}$ has an orthogonal basis labelled by all possible ways of making R choices from the d basis states with repetitions possible and the ordering of choices being irrelevant (c.f. (3) and (4)). The solution of this combinatorial problem gives

$$\text{Dimension of } \mathcal{SYM} = {}^{R+d-1}C_{d-1} = \frac{1}{(d-1)!}R^{d-1} + O(R^{d-2}), \tag{5}$$

which is a polynomial in R (for fixed d). Hence, $\mathcal{SYM}$ is again exponentially small inside $\mathcal{H}^R$ of dimension d^R.

3. Projection into $\mathcal{SYM}$. Our proposed method of stabilization consists of frequently repeated projection of the joint state of R computers into the symmetric subspace $\mathcal{SYM}$. According to the interpretation of $\mathcal{SYM}$ above, the error free component of any state always lies in $\mathcal{SYM}$ so that upon successful projection this component will be unchanged and part of the error will have been removed. Note, however, that the projected state is generally not error-free since, for example, $\mathcal{SYM}$ contains many states which are not of the simple product form $|\psi\rangle\,|\psi\rangle \ldots |\psi\rangle$. Nevertheless, the error probability will be suppressed by a factor of $1/R$ as discussed in subsequent sections. Thus the method is not one of error correction but rather of stabilization. By choosing R sufficiently large and the rate of symmetric projection sufficiently high, the residual error at the end of a computation can, in principle, be controlled to lie within any desired small tolerance.

The operation of projection into $\mathcal{SYM}$ is a computation in itself. For our stabilization method to be efficient it is essential that this operation be executable efficiently, i.e., in a number of steps that increases at most polynomially with L and R where $L = \log_2 d$ is the number of qubits required to hold the state of each computer entering into the symmetrization and R is the degree of redundancy. (Also note that R can clearly be at most a polynomial function of L in any efficient scheme.) Only then will a nominally efficient computation remain efficient after stabilization.

We next describe an algorithm for projecting into $\mathcal{SYM}$ and show that it is efficient in the above sense. Consider first a product state $|\Psi\rangle = |a_1\rangle\,|a_2\rangle \ldots |a_R\rangle \in \mathcal{H}^R$. To project $|\Psi\rangle$ into $\mathcal{SYM}$ we carry out the following steps.

Step 1. Introduce an ancilla in a standard state $|0\rangle$ with a state space $\mathcal{A}$ of at least $R!$ dimensions.

Step 2. Make an equal amplitude superposition of the ancilla

$$\mathcal{U} : |0\rangle \to \frac{1}{\sqrt{R!}} \sum_{i=0}^{R!-1} |i\rangle\,.$$

Step 3. Carry out the following computation: if the ancilla state is $|i\rangle$ then perform the ith permutation σ_i of the component states of $|a_1\rangle\,|a_2\rangle\ldots|a_R\rangle$

$$|a_1\rangle\,|a_2\rangle\ldots|a_R\rangle\,|i\rangle \to \left|a_{\sigma_i(1)}\right\rangle\left|a_{\sigma_i(2)}\right\rangle\ldots\left|a_{\sigma_i(R)}\right\rangle|i\rangle\,.$$

This results in the entangled state

$$\sum_i \left|a_{\sigma_i(1)}\right\rangle\left|a_{\sigma_i(2)}\right\rangle\ldots\left|a_{\sigma_i(R)}\right\rangle|i\rangle \in \mathcal{H}^R\otimes\mathcal{A}.$$

Step 4. Apply the reverse computation $\mathcal{U}^{-1}$ of step 2 to the ancilla. The resulting state may be written

$$|\Upsilon\rangle = \sum_i |\xi_i\rangle\,|i\rangle \in \mathcal{H}^R\otimes\mathcal{A}.$$

Since $\mathcal{U}$ transforms $|0\rangle$ to each $|i\rangle$ with equal amplitude it follows that $\mathcal{U}^{-1}$ transforms each $|i\rangle$ back to $|0\rangle$ with equal amplitude. Hence the coefficient of ancilla state $|0\rangle$ in $|\Upsilon\rangle$ is the required symmetrized state, i.e., an equal amplitude superposition of all permutations of the R factor states of $|a_1\rangle\,|a_2\rangle\ldots|a_R\rangle$.

Step 5. Measure the ancilla in its natural basis. If the outcome is "0" then $|\Psi\rangle$ has been successfully projected into $\mathcal{SYM}$. If the outcome is not "0" then the symmetrization has failed. (The issue of the probability of successful symmetrization is discussed in a later section.)

Finally note that by linearity of the process, it will symmetrize a general state in $\mathcal{H}^R$ (not just the product states considered above). If the input state is already symmetric then we get it back unchanged with certainty at the end.

Now consider the computational effort involved in the above steps. Let $d = \dim\mathcal{H}$ and write $L = \log_2 d$. Step 1 requires no computational effort. The ancilla requires $\log_2(R!) = O(R\log R)$ qubits. Step 2 may be achieved by applying the discrete Fourier transform [7], [8] to the ancilla. This requires $O((R\log R)^2)$ steps. For step 3 we note that a general permutation can be effected with $O(R\log R)$ swaps. Swapping states of L qubits requires $O(L)$ operations so overall step 3 requires $O(LR\log R)$ steps. Restoring the ancilla in step 4 requires the same number of operations as step 2. In step 5 we examine separately each of the $O(R\log R)$ qubits occupied by the ancilla, requiring $O(R\log R)$ steps. Overall we require $O(LR\log R + (R\log R)^2)$ steps which is less than $O(LR^2 + R^4)$. Hence the process is efficient.

4. A quantum network for $\mathcal{SYM}$ projection. We now describe how the operation of $\mathcal{SYM}$ projection can be implemented by a network of simple quantum gates. Consider first the following inductive definition of the general permutation of $k+1$ elements $a_1,\ldots,a_k,a_{k+1}$ [23]. Starting from the general permutation $a_{\sigma(1)},\ldots,a_{\sigma(k)}$ of the k elements $a_1,\ldots,a_k$ we adjoin a_{k+1} giving $a_{\sigma(1)},\ldots,a_{\sigma(k)},a_{k+1}$ and then perform separately the $k+1$ operations: identity, swap $a_{\sigma(1)}$ with a_{k+1}, swap $a_{\sigma(2)}$ with a_{k+1}, ... swap $a_{\sigma(k)}$ with a_{k+1}. This generates all possible permutations of $k+1$ elements. In terms of state symmetrization, if we have already symmetrized $|\psi_1\rangle\otimes\cdots\otimes|\psi_k\rangle$ (i.e., we have an equal superposition of all permutations of the states) then we can symmetrize $k+1$ states $|\psi_1\rangle\otimes\cdots\otimes|\psi_k\rangle\otimes|\psi_{k+1}\rangle$ by applying only the operation of state swapping (in suitable superposition). Thus the operation of symmetrization of R states can be built up from $|\psi_1\rangle$ by first symmetrizing $|\psi_1\rangle$ and $|\psi_2\rangle$, then successively

including $|\psi_3\rangle$ up to $|\psi_R\rangle$ always using only state swappings in suitably controlled superpositions.

The basic ingredient in this process is the "controlled swap gate" or Fredkin gate, acting on three input qubits. If the first ("control") quit is $|0\rangle$ (respectively, $|1\rangle$) then the other two ("target") qubits are unaffected (respectively, swapped). We describe this diagrammatically in Fig. 1. The operation of state swapping itself (i.e., $|\psi_1\rangle \otimes |\psi_2\rangle \mapsto |\psi_2\rangle \otimes |\psi_1\rangle$) can be implemented using three controlled–NOT gates as described in [15].

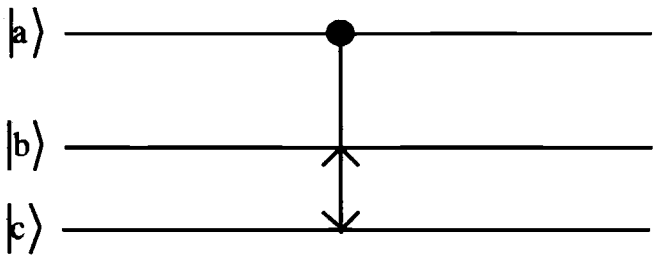


FIG. 1. *Schematic representation of a Fredkin gate. A Fredkin gate exchanges the state of the second and third qubit if and only if the first qubit is in state* $|a\rangle = |1\rangle$.

To symmetrize $k+1$ qubits given that the first k are already symmetrized we introduce k control qubits initially in state $|0\rangle\,|0\rangle \ldots |0\rangle$ and apply a suitable unitary transformation, denoted U_k to generate the superposition

$$\frac{1}{\sqrt{k+1}}\left(|00\ldots0\rangle + |10\ldots0\rangle + |01\ldots0\rangle + \cdots + |00\ldots1\rangle\right). \tag{6}$$

The unitary transformation U_k can be readily obtained by a quantum network consisting of a one bit gate performing the transformation

$$\frac{1}{\sqrt{k+1}}\begin{pmatrix} 1 & -\sqrt{k} \\ \sqrt{k} & 1 \end{pmatrix} \tag{7}$$

on the first quit and a sequence of $k-1$ two bit gates $T_{j,j+1}$ for $j = 1, \ldots, k-1$ acting on the jth and $j+1$th qubits. In the basis $\{|0\rangle, |1\rangle\}$, $T_{j,j+1}$ is given by:

$$T_{j,j+1} = \frac{1}{\sqrt{k-j+1}}\begin{pmatrix} \sqrt{k-j+1} & 0 & 0 & 0 \\ 0 & 1 & \sqrt{k-j} & 0 \\ 0 & -\sqrt{k-j} & 1 & 0 \\ 0 & 0 & 0 & \sqrt{k-j+1} \end{pmatrix}. \tag{8}$$

Having thus initialized the k control qubits, we then apply k Fredkin gates—the jth Fredkin gate (for $j = 1, \ldots k$) uses the jth control quit to control the swapping of the jth and $(k+1)$th target qubits. This leads to an entangled state of the k control qubits and the $k+1$ target qubits but after applying U_k^{-1} to the control qubits, the coefficient of $|0\rangle\,|0\rangle \ldots |0\rangle$ will be the required symmetrization of the $k+1$ qubits (c.f. step 4 of section 3). Finally, a measurement of the control qubits will effect the projection into $\mathcal{SYM}$ (cf. step 5 of section 3).

Thus to symmetrize R qubits we cascade the above construction with $k = 1, 2, \ldots$ up to $k = R-1$ requiring a total number $1+2+\cdots+(R-1) = R(R-1)/2$ of control qubits. The size of the overall network is clearly quadratic in R. For example, for the symmetrization of $R = 4$ qubits we obtain the network shown in Fig. 2.

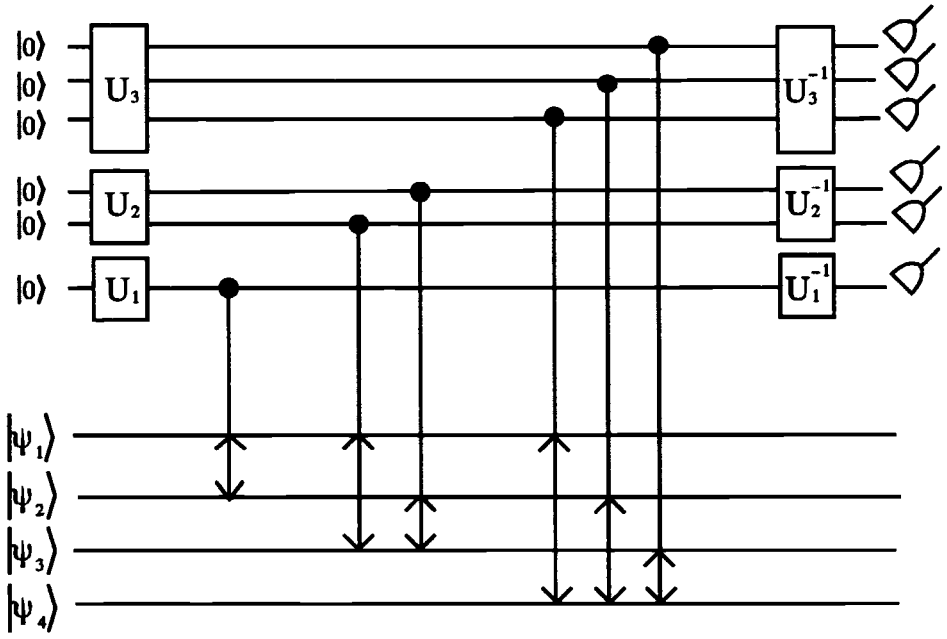


FIG. 2. *Quantum network for symmetrizing $R = 4$ qubits. Six auxiliary qubits initially in state $|0\rangle$ are needed. The auxiliary qubits are put into an entangled state and used to control the state swapping of the four computer qubits. The operations are then undone and the auxiliary qubits measured. If every auxiliary quit is found in state $|0\rangle$ the symmetrization has been successful.*

5. Stabilization against unitary errors. So far we have given an efficient algorithm for projection into the symmetric subspace and provided an intuitive reason why it would be expected to reduce the error while preserving the correct computation. We now turn to a quantitative study of the effect of $\mathcal{SYM}$-projection as a basis for stabilization in the presence of various modes of error production. It is convenient to separate the discussion into two parts considering the case where the joint state of the computers remains in a pure state, in this section, and the case of decoherence due to external environmental interaction in the following section.

Consider the simple model of R qubits initially in state (the "correct" state) $|0\rangle\,|0\rangle\ldots|0\rangle$ with computation being the identity, i.e., we are considering the state storage problem with R–fold redundancy. Suppose that the R storage devices are subject to independent hardware errors which cause the jth state to drift as $e^{iH_j t}\,|0\rangle$. Here the Hamiltonians H_j are random and independent. Since the devices were intended for state storage we assume that the rate of drift is suitably bounded. This is expressed by requiring that all eigenvalues of the H_j's are suitably small

$$|\text{eigenvalues of } H_j| \leq \epsilon, \qquad j = 1, \ldots, R \tag{9}$$

for some (small) constant ϵ. The stabilization process consists of projecting the joint state of the R copies into the symmetric subspace at short time intervals δt. For simplicity we will assume that the projection can be performed essentially instantaneously. Furthermore, we assume that (unlike the computation being stabilized) the projection process itself is error free. These and other assumptions will be discussed in section 7. Under these assumptions we can readily compare the growth of errors with and without the stabilization process.

In the basis $\{|0\rangle, |1\rangle\}$ write

$$H_j = \begin{pmatrix} a_j & c_j^* \\ c_j & b_j \end{pmatrix} \tag{10}$$

so that

$$|c_j| \leq |\lambda_1| + |\lambda_2| \leq 2\epsilon \tag{11}$$

(where λ_1, λ_2 are the eigenvalues of H_j). We will assume that δt is small and we retain only the lowest order terms in δt. After time δt the state will be

$$|\Psi(\delta t)\rangle = \bigotimes_{k=1}^{R} \{(1 + ia_k\delta t)\,|0\rangle + ic_k\delta t\,|1\rangle\}\,. \tag{12}$$

Thus without symmetrization the probability that the ith qubit shows an error is

$$|c_i|^2\delta t^2 \approx 4\epsilon^2\delta t^2. \tag{13}$$

If we expand out the product in (12) we obtain 2^R terms corresponding to the exponentially large dimension of the full space of R qubits. However, the amplitudes of terms involving k errors (i.e., products of $|0\rangle$'s and $|1\rangle$'s involving k $|1\rangle$'s) will have size $O(\delta t^k)$ and only the R terms involving one error will have size $O(\delta t)$. Thus the erroneous state (12) does not occupy these exponentially many dimensions of $\mathcal{H}^R$ equally. We noted previously that $\mathcal{SYM}$ is exponentially small inside $\mathcal{H}^R$ but the preceding observation indicates that $\mathcal{SYM}$–projection will not generally remove *exponentially* much of the error since only R of these exponentially many dimensions are entered (to lowest order) by the erroneous evolution. We now calculate the stabilizing effect of the $\mathcal{SYM}$-projection.

Consider the basis of $\mathcal{SYM}$ given by the $R+1$ orthonormal states (cf. (4) for the case $R = 3$):

$$|e_k\rangle = \frac{1}{\sqrt{{}^RC_k}} \sum_{\text{all } \text{“}k\text{-error”}\ \sigma\text{'s}} |\sigma\rangle\,, \qquad k = 0, \ldots, R. \tag{14}$$

Here the sum is over all ${}^RC_k$ possible strings of 0's and 1's of length R containing exactly k 1's and $R-k$ 0's. Under $\mathcal{SYM}$-projection the lowest order error terms (single-error terms) of (12) will project only onto $|e_1\rangle$. For the term with an error in the kth place we get

$$ic_k\delta t\,|0\rangle \ldots |0\rangle\,|1\rangle\,|0\rangle \ldots |0\rangle \qquad \underset{\longrightarrow}{\mathcal{SYM}\text{ proj}} \qquad \frac{ic_k\delta t}{\sqrt{{}^RC_1}}\,|e_1\rangle\,.$$

Thus the normalized projected state has the form

$$\left\{1 + i\delta t \sum_{j=1}^{R} a_j\right\} |0\rangle \ldots |0\rangle + \alpha_1\,|e_1\rangle + O(\delta t^2)$$

where

$$\alpha_1 = \frac{i\delta t}{\sqrt{R}} \sum_{j=1}^{R} c_j. \tag{15}$$

To estimate the size of α_1, using (11) we write $c_j \approx 2\epsilon e^{i\theta_j}$ where θ_j are random phases. The expectation value of α_1 is then clearly zero but from

$$\text{Expectation value of } \left|\sum_{j=1}^{R} e^{i\theta_j}\right| = \sqrt{R} \tag{16}$$

we get [1]

$$\text{Expectation value of } |\alpha_1|^2 = 4\epsilon^2\delta t^2. \tag{17}$$

Thus, somewhat surprisingly, this probability of a single symmetrized error does not decrease with R. However, it is associated with R copies and to see its residual effect on any *one* copy we use the following fact.

PROPOSITION 1. *Consider the state*

$$|\Xi\rangle = \sum_{k=0}^{R} \alpha_k |e_k\rangle \in \mathcal{SYM} \subseteq \mathcal{H}^R$$

where $|e_k\rangle$ *are as given in* (14). *If one qubit is measured in the basis* $\{|0\rangle, |1\rangle\}$ *the probability of obtaining* $|1\rangle$ *is* $\frac{1}{R}\sum_{k=0}^{R} |\alpha_k|^2 k$.

Proof. Since the state is symmetric the probability of obtaining the result $|1\rangle$ for the ith qubit is the same as this probability for the first qubit. Now $|e_k\rangle$ in (14) consists of ${}^{R}C_k$ orthogonal terms of which ${}^{R-1}C_{k-1}$ have $|1\rangle$ in the first place. Hence, the term $\alpha_k |e_k\rangle$ in $|\Xi\rangle$ contributes probability

$${}^{R-1}C_{k-1} \left|\frac{\alpha_k}{\sqrt{{}^{R}C_k}}\right|^2 = |\alpha_k|^2 k/R$$

of obtaining outcome $|1\rangle$. □

Applying this result to (15) and using (17) we see that after successful symmetrization the probability of error (to lowest order in δt) is $4\epsilon^2\delta t^2/R$, i.e., the error is suppressed by a factor of $1/R$ compared to the case (13) of no symmetrization and in each step the amplitude of correct computation is correspondingly enhanced.

The above result is conditional on the success of the symmetrization, i.e., that the state projects to $\mathcal{SYM}$ rather than $\mathcal{SYM}^{\perp}$. If the projections are done frequently enough then the cumulative probability that they *all* succeed can be made as close as desired to unity. This is a consequence of the so-called "quantum watched pot effect" [16]. Consider a normalized joint state $|\Xi\rangle$ of R copies initially in $\mathcal{SYM}$. Its initial probability of successful projection is 1 which is a *maximum*. Thus as the state evolves by some unitary transformation into the ambient space $\mathcal{H}^R$ the probability of successful projection will begin to change only to second order in time. If we project n times per unit time interval, i.e., $\delta t = 1/n$ then the cumulative probability that all projections in one unit time interval succeed, is

$$(1 - k\delta t^2)^n = \left(1 - \frac{k}{n^2}\right)^n \to 1 \text{ as } n \to \infty.$$

[1] *Remark* [21]. If instead of qubit systems we consider computers with dimensions large compared to the degree of redundancy R, then we would expect the individual random errors to be mutually orthogonal, so when the state is symmetrized their sum does not exhibit the cancelling effects which are present for qubits and lead to (16) and (17). However in that case, (17) and subsequent conclusions still hold because (16) may be replaced by Pythagoras' theorem, i.e., that the sum of R orthonormal vectors has length $\sqrt{R}$.

Here k is a constant depending on the rate of rotation of the state out of $\mathcal{SYM}$. For redundancy degree R and the model of random unitary errors considered above we find that k grows linearly with R (as can be seen by directly calculating the length of the $\mathcal{SYM}$-projection of (12) to $O(\delta t^2)$ terms). Thus to achieve a cumulative probability of successful projection of $1-\zeta$ in a unit time interval we would require a rate of symmetric projection which increases linearly with $-R/\log(1-\zeta)$.

The above conclusions—for a model of random independent unitary errors—will also apply to computations which are not the identity. Formally, we may view the computation in a moving basis relative to which the correct computation is the identity and the previous arguments are unchanged, i.e., none of the arguments depends on the actual identity of the computational basis states.

6. Stabilization against environmental interaction. We now consider the problem of state storage with R-fold redundancy, in the presence of decoherence, i.e., interaction with an external environment. In general each qubit will become entangled with an environment and the state of the qubit alone will no longer be describable by a pure state. It will be represented by a density matrix [16] resulting from forming a partial trace over the environment, of the joint (pure) state of the total qubit-environment system.

Consider R copies of the qubit initially all prepared in pure state $\rho_0 = |0\rangle\langle 0|$. We will assume that they interact with independent environments (this assumption is valid if the coherence length of the reservoir is less than the spatial separation between the copies [6]) so that after some short period of time δt the state of the R copies will have undergone an evolution

$$\rho^{(R)}(0) = \rho_0 \otimes \cdots \otimes \rho_0 \quad \longrightarrow \quad \rho^{(R)}(\delta t) = \rho_1 \otimes \cdots \otimes \rho_R, \tag{18}$$

where $\rho_i = \rho_0 + \sigma_i$ for some Hermitian traceless σ_i and the superscript R denotes the number of states involved. We will retain only terms of first order in the perturbations σ_i so that the overall state at time δt is

$$\begin{aligned} \rho^{(R)} = \rho_0 \otimes \cdots \otimes \rho_0 &+ \sigma_1 \otimes \rho_0 \otimes \cdots \otimes \rho_0 \\ &+ \rho_0 \otimes \sigma_2 \otimes \cdots \otimes \rho_0 \ldots \\ &+ \rho_0 \otimes \rho_0 \otimes \cdots \otimes \sigma_R \\ &+ O(\sigma_i \sigma_j), \end{aligned} \tag{19}$$

and we wish to compute the projection of the state (19) into the symmetric subspace $\mathcal{SYM}$. Then we construct the state of the ith qubit by partial trace over all qubits except the ith and finally compare the resulting state with $\rho_0 + \sigma_i$ and see that its purity has been suitably enhanced, bringing it closer to ρ_0.

The mathematical formalism for symmetrization of mixed states has some curious features which we digress to clarify before treating (19) itself. Consider a state $\rho_1 \otimes \rho_2$ of two qubits. The state $\frac{1}{2}(\rho_1 \otimes \rho_2 + \rho_2 \otimes \rho_1)$ is *not* a symmetric state and in fact $\rho \otimes \rho$ is not symmetric (i.e., it is not a density matrix supported on the subspace $\mathcal{SYM}$) unless ρ is pure! To see this consider ρ written in its diagonalizing basis of orthonormal eigenstates:

$$\rho = \lambda_1 |\lambda_1\rangle\langle\lambda_1| + \lambda_2 |\lambda_2\rangle\langle\lambda_2|. \tag{20}$$

Thus we can represent ρ as a mixture of its two eigenstates, and $\rho \otimes \rho$ as a mixture of the four orthonormal states $|\lambda_i\rangle \otimes |\lambda_j\rangle$ with a priori probabilities $p_{ij} = \lambda_i \lambda_j$. This latter mixture involves *nonsymmetric* states (like $|\lambda_1\rangle \otimes |\lambda_2\rangle$) so $\rho \otimes \rho$ is not symmetric.

One way of constructing the projection of $\rho \otimes \rho$ into $\mathcal{SYM}$ is to project each state of the above mixture into $\mathcal{SYM}$. Let $|\mu_{ij}\rangle$ denote the $\mathcal{SYM}$-projection of $|\lambda_i\rangle \otimes |\lambda_j\rangle$ and $|\hat{\mu_{ij}}\rangle$ denote the corresponding normalized state. The probability of successful projection is $q_{ij} = \langle \mu_{ij} | \mu_{ij} \rangle$. Then the $\mathcal{SYM}$-projection of $\rho \otimes \rho$ is the state corresponding to the mixture $|\hat{\mu_{ij}}\rangle$ with a priori probabilities $p_{ij}q_{ij}/(\sum p_{ij}q_{ij})$, which are the conditional probabilities of occurrence of states $|\hat{\mu_{ij}}\rangle$ given that the $\mathcal{SYM}$-projection was successful.

More formally we may introduce the (Hermitian) permutation operators $P_{12} =$ "identity" and $P_{21} =$ "swap" acting on pure states of two qubits and define the symmetrization operator:

$$S = \frac{1}{2}(P_{12} + P_{21}). \tag{21}$$

The $\mathcal{SYM}$-projection of a pure state $|\Psi_{12}\rangle$ of two qubits is just $S\,|\Psi_{12}\rangle$, which is then renormalized to unity. It follows that the induced map on *mixed* states of two qubits (including renormalization) is

$$\rho_1 \otimes \rho_2 \longrightarrow \frac{S(\rho_1 \otimes \rho_2)S^\dagger}{\operatorname{Tr} S(\rho_1 \otimes \rho_2)S^\dagger}. \tag{22}$$

The state of either qubit is obtained separately by partial trace over the other qubit.

As an example consider the symmetric projection of $\rho \otimes \rho$ followed by renormalization and partial trace (over either qubit) to obtain the final state $\tilde{\rho}$ of one qubit, given that the $\mathcal{SYM}$-projection was successful. A direct calculation based on (22) yields

$$\rho \mapsto \tilde{\rho} = \frac{\rho + \rho^2}{\operatorname{Tr}(\rho + \rho^2)}. \tag{23}$$

For any mixed state ξ of a qubit the expression $\operatorname{Tr} \xi^2$ provides a measure of the purity of the state, ranging from $1/4$ for the completely mixed state $I/2$ (where I is the unit operator) to 1 for any pure state. From (23) we get

$$\operatorname{Tr} \tilde{\rho}^2 > \operatorname{Tr} \rho^2$$

so that $\tilde{\rho}$ is *purer* than ρ. This example illustrates a generic fact (cf. below), that successful projection of a mixed state into $\mathcal{SYM}$ tends to enhance the purity of the individual systems. Indeed, consider further the state $\otimes^R \rho$ consisting of R independent copies of ρ. The symmetrization operator is

$$S = \frac{1}{R!} \sum_{\alpha=1}^{R!} P_\alpha, \tag{24}$$

where the sum ranges over all $R!$ permutations of the R indices. If we project $\otimes^R \rho$ into $\mathcal{SYM}$ and renormalize (as in (22)) and calculate the partial trace over all but one of the qubits, we obtain a reduced state $\tilde{\rho_R}$ which asymptotically tends to a *pure* state as $R \to \infty$. This limiting pure state is the eigenstate of ρ belonging to its largest eigenvalue.

Let us now return to the consideration of (19) and its $\mathcal{SYM}$-projection. The application of the symmetrization operator (24) to each of the R terms of $\rho_0 \otimes \cdots \otimes$

$\sigma_i \otimes \cdots \otimes \rho_0$ of equation (19) generates $R!^2$ terms of the form

$$\frac{1}{R!^2} P_\alpha \rho_0 \otimes \cdots \otimes \sigma_i \otimes \cdots \otimes \rho_0 P_\beta, \tag{25}$$

where P_α and P_β are permutation operators on the state space $\mathcal{H}^R$ of R qubits as above. To calculate the reduced density operator of the first qubit we take the partial trace over the $R-1$ remaining qubits. Note that the reduced states of all qubits individually are equal since the total overall state is symmetric. (To systematize the calculation of the partial traces we have found it very convenient to use the diagrammatic notation for tensor operations introduced by Penrose in [22].) For each σ_i we find that the $R!^2$ terms in (25) then reduce to the following cases:

(i) $(R-1)!^2$ terms each equal to $\sigma_i/R!^2$ corresponding to all permutations P_α and P_β which place σ_i in the first position in (25). In this case the partial trace contracts out all the ρ_0 terms leaving a coefficient of $1/R!^2$ (as the trace of any power of ρ_0 is 1).

(ii) $(R-1)!^2(R-1)R$ terms of the forms $\rho_0\sigma_i\rho_0$, $\rho_0\sigma_i$, $\sigma_i\rho_0$, or $\rho_0\mathrm{Tr}(\sigma_i\rho_0)$, each one divided by $R!^2$. These correspond to all pairs of permutations which result in σ_i contracted onto ρ_0 in all possible ways in the partial traces.

(iii) $(R-1)!^2(R-1)$ terms which result in σ_i being contracted onto itself in the partial traces. These terms are all zero since $\mathrm{Tr}\,\sigma_i = 0$.

Note that each term in (ii) has trace given by $\mathrm{Tr}\,\sigma_i\rho_0/R!^2$ and each term in (i) has zero trace. Thus the resulting density operator, before normalization, has a trace given by

$$1 + (R-1)\mathrm{Tr}(\rho_0\tilde{\sigma}), \tag{26}$$

where we have introduced $\tilde{\sigma} = \frac{1}{R}\sum_{i=1}^{R}\sigma_i$. We normalize the density operator by dividing the sum of all terms in (i) and (ii) for all $i = 1, \ldots, R$ by (26), the resulting symmetrized density operator $\tilde{\rho}$ can be written

$$\begin{aligned}\tilde{\rho} = {} & [1-(R-1)\mathrm{Tr}(\rho_0\tilde{\sigma})]\rho_0 + \frac{1}{R}\tilde{\sigma} \\ & + (R-1)[A\rho_0\tilde{\sigma}\rho_0 + B(\rho_0\tilde{\sigma} + \tilde{\sigma}\rho_0) + C\rho_0\mathrm{Tr}(\tilde{\sigma}\rho_0)] + O(\sigma_i\sigma_j),\end{aligned} \tag{27}$$

where A, B, and C depend on R and $A + 2B + C = 1$.

If a general mixed state ξ of a qubit is measured in the basis $\{|0\rangle, |1\rangle\}$ then the probability that the outcome is 0 is given by $\langle 0|\,\xi\,|0\rangle = \mathrm{Tr}\,\rho_0\xi$. This provides the success probability in our present model. Thus the average success probability *before* symmetrization of the perturbed qubits is

$$\frac{1}{R}\sum_i \mathrm{Tr}\,\rho_0(\rho_0 + \sigma_i) = 1 + \mathrm{Tr}\,\rho_0\tilde{\sigma} \tag{28}$$

(note that consequently $\mathrm{Tr}\,\rho_0\tilde{\sigma}$ is necessarily negative). *After* symmetrization, using (27) we see that

$$\mathrm{Tr}\,\rho_0\tilde{\rho} = 1 + \frac{1}{R}\mathrm{Tr}\,\rho_0\tilde{\sigma}. \tag{29}$$

Hence, the probability of error has again been reduced by a factor of R—exactly as found in the previous section.

We can calculate the average purity of the R copies before symmetrization by calculating the average trace of the squared states:

$$\frac{1}{R}\sum_{i=1}^{R}\mathrm{Tr}((\rho_0+\sigma_i)^2) = 1 + 2\mathrm{Tr}(\rho_0\tilde{\sigma}). \tag{30}$$

After symmetrization each qubit has purity

$$\mathrm{Tr}(\tilde{\rho}^2) = 1 + 2\frac{1}{R}\mathrm{Tr}(\rho_0\tilde{\sigma}). \tag{31}$$

Since $\mathrm{Tr}\,\tilde{\rho}^2$ is closer to 1 than (30), the resulting symmetrized system $\tilde{\rho}$ is left in a purer state. Indeed it follows from (29) that $\tilde{\rho}$ approaches the unperturbed state ρ_0 as R tends to infinity.

7. Limitations. Error correction is itself a quantum computation. The above analysis has ignored the inevitable build up of errors in the computer performing that computation. Indeed for the symmetrization of R qubits the projection algorithm requires an ancilla of at least $R!$ dimensions, i.e., $O(R\log R)$ qubits (in fact $O(R^2)$ in our explicit network). Thus the correcting apparatus is slightly larger than the total system being corrected so the error correction ought to be subject to a similar level of error as is present in the original system. In a situation where the redundancy degree R is small compared to the number L of qubits per computer, the correcting apparatus (still of $O(R^2)$ qubits) will be small compared to the size RL of the system being corrected. However, as seen in section 5, the stabilization of a linear computation on input size L requires redundancy degree $R \sim L$ so that the correcting apparatus and the computer are again of comparable size. This means that each error correcting step introduces errors of a similar, or even greater, probability than those it is correcting. This does not, however, necessarily render it ineffective. Consider the following illustrative example. A certain clock is accurate to one second per day. Each day at noon it is reset using a standard time signal, the resetting operation being accurate only to one minute, i.e., 60 times worse than the error being corrected. Nevertheless, after 10 years the corrected clock will still be in error by at most one minute. If left uncorrected the error could be almost an hour. In our stabilization method the analogue of "resetting noon to within one minute" is projection into $\mathcal{SYM}$ with some error tolerance. Thus although the projection is imperfect, the state never drifts very far from $\mathcal{SYM}$ as it would do in the absence of any stabilization.

The main factor limiting the efficiency of our proposed method will be the frequency with which the error correcting operations can be physically performed. As noted at the end of section 5, to achieve a cumulative probability $1-\delta$ of repeated successful projection in a unit time interval, the rate of symmetric projection must increase linearly with the degree of redundancy R. Also as noted in section 1, the stabilization of a computer with input size L, running for L steps, requires R to increase linearly with L. Hence, we need the overall rate of symmetric projection to increase linearly with L even for a linear time computation. Thus, beyond a certain input size, each symmetrization will have to be performed in a time shorter than that needed to perform the elementary quantum gate operations. Since increasing the rate of computation by a factor k presumably requires resources exponential in k, our method would necessarily require exponential resources for sufficiently large L. This property is shared by all quantum error correction schemes that have been proposed to date. Hence quantum algorithms (such as Shor's factoring algorithm), which are

polynomially efficient in the absence of errors, would not be efficient if physically implemented. We wish to stress that the traditional notion of efficiency (based on the distinction between polynomial and exponential growth) is an asymptotic notion referring to computations on unboundedly large inputs. This may not be appropriate in assessing the feasibility of particular computations in practice. For example, if a quantum computer could factorize a 1000-digit integer in a reasonable time it may still exceed the abilities of any classical computer for the foreseeable future albeit that the factorization of 2000-digit integers might be infeasible on *any* computer.

8. Conclusion. If the technology to implement the scheme we have described were available, it would provide a method of stabilizing general coherent computations though not with exponential efficiency. This is because although only polynomially many steps are required in the stabilization computation, these need to be performed in a fixed time, a characteristic time of error growth per bit.

Acknowledgments. We wish to thank Dorit Aharonov, Ethan Bernstein, Asher Peres, and Umesh Vazirani for developmental discussions, and Rolf Landauer for critical appraisal. We are grateful for the opportunity of collaboration provided by ELSAG-Bailey, Genova, and the Institute for Scientific Interchange, Torino.

REFERENCES

[1] C. Papadimitriou (1994), *Computational Complexity*, Addison–Wesley, Reading, MA.
[2] D. Deutsch (1993), *The Quantum Theory of Computation*, talk presented at the Rank Prize Funds Mini–Symposium on Quantum Communication and Cryptography, Broadway, England.
[3] A. Berthiaume, D. Deutsch, and R. Jozsa (1994), *The stabilization of quantum computations*, in Proc. Workshop Physics Computation, PhysComp94, IEEE Computer Society Press, Los Alamitos, CA, pp. 60–62.
[4] W. H. Zurek (1991), *Decoherence and the transition from quantum to classical*, Physics Today, 44, p. 36.
[5] R. Landauer (1995), *Is quantum mechanics useful?*, Phil. Trans. Roy. Soc., 353, pp. 367–376.
[6] G. M. Palma, K.-A. Suominen, and A. Ekert (1996), *Quantum computation and dissipation*, Proc. Roy. Soc. London Ser. A, 452, pp. 567–584.
[7] P. Shor (1994), *Algorithms for quantum computation: Discrete logarithms and factoring*, in Proc. 35th Annual Symposium Foundations of Computer Science, S. Goldwasser ed., IEEE Computer Society Press, Los Alamitos, CA, pp. 124–134.
[8] A. Ekert and R. Jozsa (1996), *Quantum computation and Shor's factoring algorithm*, Rev. Modern Phys., 68, pp. 733–753.
[9] P. Shor (1995), *Scheme for reducing decoherence in quantum memory*, Phys. Rev. A, 52, pp. R2493–R2496.
[10] A. Steane (1996), *Multiparticle interference and quantum error correction*, Proc. Roy. Soc. London Ser. A, 452, pp. 2551–2577.
[11] R. Laflamme, C. Miquel, J. P. Paz, and W. H. Zurek (1996), *Perfect quantum error correction code*, Phys. Rev. Lett., 77, pp. 198–201.
[12] A. Ekert and C. Macchiavello (1996), *Quantum error correction and communication*, Phys. Rev. Lett., 77, pp. 2585–2588.
[13] A. Calderbank and P. Shor (1996), *Good quantum error correcting codes exist*, Phys. Rev. A, 54, pp. 1098–1106.
[14] S. Lang (1993), *Algebra*, 3rd ed., Addison–Wesley, Reading, MA.
[15] A. Barenco, D. Deutsch, A. Ekert, and R. Jozsa (1995), *Conditional quantum dynamics and logic gates*, Phys. Rev. Lett., 74, pp. 4083–4087.
[16] A. Peres (1993), *Quantum Theory: Concepts and Methods*, Kluwer Academic Publishers, Norwell, MA.
[17] A. Peres (1988), *How to differentiate between non-orthogonal states*, Phys. Lett. A, 128, pp. 19–20.
[18] C. A. Fuchs and A. Peres (1996), *Quantum state disturbance versus information gain: Uncertainty relations for quantum information*, Phys. Rev. A, 53, pp. 2038–2045.

[19] W. K. Wootters and W. Zurek (1982), *A single quantum cannot be cloned*, Nature, 299, pp. 802-803.
[20] F. J. MacWilliams and N. J. Sloane (1977), *The Theory of Error Correcting Codes*, North–Holland, Amsterdam.
[21] Dorit Aharonov, private communication, 1995.
[22] R. Penrose and W. Rindler (1984), *Spinors and Spacetime*, Vol. 1, Appendix, Cambridge University Press, London.
[23] D. Knuth (1973), *The Art of Computer Programming*, Vol. 2, Addison–Wesley, Reading, MA.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1559–1580, December 1997

THE RANDOM ADVERSARY: A LOWER-BOUND TECHNIQUE FOR RANDOMIZED PARALLEL ALGORITHMS*

PHILIP D. MACKENZIE†

Abstract. The random-adversary technique is a general method for proving lower bounds on randomized parallel algorithms. The bounds apply to the number of communication steps, and they apply regardless of the processors' instruction sets, the lengths of messages, etc. This paper introduces the random-adversary technique and shows how it can be used to obtain lower bounds on randomized parallel algorithms for load balancing, compaction, padded sorting, and finding Hamiltonian cycles in random graphs. Using the random-adversary technique, we obtain the first lower bounds for randomized parallel algorithms which are provably faster than their deterministic counterparts (specifically, for load balancing and related problems).

Key words. parallel algorithms, parallel computation, PRAM model, randomized parallel algorithms, expected time, lower bounds, load balancing

AMS subject classifications. 68Q10, 68Q22, 68Q25

PII. S0097539791224030

1. Introduction.

Randomization has been a useful tool in developing fast parallel algorithms for a vast spectrum of problems, from computational geometry and graph theory to routing and load balancing. Often these randomized parallel algorithms are significantly faster than the best possible deterministic parallel algorithms. This prompts the question, "To what extent can randomization improve the speed of parallel algorithms?" We have developed the random-adversary technique to help answer this question.

The random-adversary technique can be thought of as an extension or generalization of the random-restriction technique [18]. It is the most general technique to date, and in fact, to date, all known asymptotic lower bounds for the randomized parallel random access machine (PRAM) model (which assume no restrictions on processors' instruction sets nor restrictions on memory) can be proven using the random-adversary technique. In this paper, we will focus on lower bounds for the strongest of the PRAM models, the concurrent-read concurrent-write (CRCW) PRAM. There has been a tremendous amount of work in the area of lower bounds on this model, and tight lower bounds have been shown for many fundamental problems in the deterministic case [4, 5, 8, 9, 11, 14, 16, 25]. The only previous parallel randomized lower bounds known for this model (again, which assume no restrictions on processors' instruction sets nor restrictions on memory), however, were the lower bounds for parity and related problems [1, 4, 31]. Using the random-adversary technique, we have been able to prove lower bounds on the randomized CRCW PRAM model for many other problems, including the fundamental load-balancing problem. Moreover, the random-adversary technique seems to be a very natural way for proving these lower bounds. One key feature of the method is that one abstracts the difficulty of the problem (by

*Received by the editors December 23, 1991; accepted for publication (in revised form) August 17, 1995.

http://www.siam.org/journals/sicomp/26-6/22403.html

†Department of Mathematics and Computer Science, Boise State University, Boise, ID 83725 (philmac@diamond.idbsu.edu). This research was performed while the author was at Sandia National Laboratories and was supported in part by Department of Energy contract DE-AC04-76DP00789, at the University of Texas supported by Texas Advanced Research Projects grant 003658480, and at the University of Michigan supported by an AT&T Fellowship and by NSF/DARPA grant CCR-9004727.

specifying the inputs that are difficult for an algorithm to deal with) before getting into the detailed mechanics of the proof. Hopefully, this idea, and the outline of the technique given later, will make it easier for researchers to prove lower bounds for parallel randomized algorithms solving other problems. We certainly feel that given its natural and fairly easy structure, the random-adversary technique should be the first method one should try in proving any new lower bounds for parallel randomized algorithms.

1.1. Load balancing. One of the most important applications of our technique so far is to the problem of load balancing, which is one of the most fundamental problems in parallel computing. While load balancing is implicit in most deterministic PRAM algorithms, it is often necessary to have an explicit load-balancing procedure in randomized PRAM algorithms. In these algorithms some processors might finish quickly, and to make full use of them, they must be reallocated to help processors which have not finished. Without this reallocation, the time of the algorithm will be dependent on the processor with the most difficult subproblem. Unfortunately, the time for load balancing itself can also increase the running time of an algorithm, and that is what we will analyze in this paper.

We consider the following *load-balancing* problem, as defined by Gil, Matias, and Vishkin [21].

Load balancing. Given h objects distributed among n processors, redistribute the objects so that each processor gets $O(1 + h/n)$ objects.

Note that under any definition of load balancing, at least one processor will end with $\Omega(1+h/n)$ objects. Consequently, by solving the load-balancing problem defined above, the objects will be balanced in an asymptotically optimal manner. Then, assuming the objects are equal length tasks, after this load-balancing operation the tasks can be completed by the processors in asymptotically optimal time. (Note that for $h \geq n$, a more applicable definition for load balancing might be to redistribute objects so that a constant fraction of the processors contain at least one object. However, for $h \leq n$, the definition given above seems to be more appropriate. We are more concerned with the case of $h \leq n$, but for generality, we expand the definition of the problem to deal with all h.)

In general we would like to perform this load balancing in constant time, since then at each step, processors that have finished their tasks could be reassigned to help some other processors right away. We show, however, that load balancing requires $\Omega(\log^* n)$ expected time. We note that there might be a possibility of *batching* or *pipelining* the load balancing, and so an algorithm which would take $\Theta(f(n))$ time with a constant-time load-balancing operation still might be made to take $o(f(n) \log^* n)$ time. That is beyond the scope of the paper.

Load-balancing procedures have been used in Chlebus et al. [12] to perform simulations of strong CRCW PRAM models on weaker CRCW PRAM models, in MacKenzie and Stout [37] for padded sorting, finding nearest neighbors, and constructing the Voronoi diagram, in Gil and Matias [20] to perform parallel hashing, in Hagerup [27] to perform integer chain-sorting, and in Goodrich [24] to construct upper envelopes, answer range queries, and perform hidden surface elimination. Recently, Gil, Matias, and Vishkin [21], Hagerup [27], and Goodrich [24] have all developed $\Theta(\log^* n)$ expected time randomized load-balancing algorithms for worst-case inputs, so the lower bound given in this paper is tight. (The algorithm in Goodrich [24] is for a slightly less general load-balancing problem.)

We will also show $\Omega(\log^* n)$ time lower bounds for the related problems of padded

sorting [37] and compaction [22, 28] on the CRCW PRAM. The definitions for these problems are as follows.

Padded sort. Given n values taken from a uniform distribution over the unit interval $[0, 1]$, arrange them in sorted order in an array of size $n + o(n)$, with the value NULL in all unfilled locations.

Compaction. Given an array of n cells with at most h containing one item each and all others being empty, insert the items into an array of size $O(h)$.

Matias and Vishkin [39] give a $\Theta(\log^* n)$ expected time algorithm for compaction with arbitrary $h \in \{1, \ldots, n\}$, and Hagerup and Raman [29] give a $\Theta(\log^* n)$ expected time algorithm for padded sort. Consequently, our lower bounds are tight.

Recently, Chaudhuri [10] gave an $\Omega(\log \log n)$ deterministic lower bound for compaction, proving that randomization is necessary in obtaining a faster solution. Our lower bound for compaction is the first for a problem that has a provably faster parallel randomized solution than parallel deterministic solution.

We should mention that a further result obtained from the lower bound on compaction is an $\Omega(\log^* n)$ time separation between the standard randomized CRCW PRAM and the randomized CRCW-bit PRAM [6], in which $O(\log^{(k)} n)$ processors are allowed to simultaneously write into different bits of the same word, for some constant $k \geq 1$. This follows from the constant time algorithm for compaction in the randomized CRCW-bit PRAM model given by Goodrich [24].

1.2. Random graph properties. By a *random graph on n vertices*, denoted $G_{n,p}$, we mean that each of the $\binom{n}{2}$ edges is included with probability p, where p is a constant, $0 < p < 1$. Random graphs have been extensively studied and have very interesting properties. For instance, with high probability, in $G_{n,p}$ a Hamiltonian cycle exists; every breadth first spanning tree has height 2; and, for n even, there is a maximal matching which includes all the vertices.

Bollobás, Fenner, and Frieze [7] give an algorithm which constructs a Hamiltonian cycle in $G_{n,p}$ if one exists. This algorithm runs in polynomial expected time for $p \geq 1/2$ [7, Theorem 1.2]. Gurevich and Shelah [26] and Thomason [40] independently improve on this result, giving algorithms which run in linear expected time for any constant p. These algorithms are optimal, since linear time is needed just to write the output. Frieze [17] gives a parallel algorithm which constructs a Hamiltonian cycle from $G_{n,p}$ in $O((\log \log n)^2)$ expected time and uses $n \log^2 n$ processors. MacKenzie and Stout [38] show that one can find a Hamiltonian cycle, a maximal matching, and a breadth first spanning tree in $G_{n,p}$, all in $\Theta(\log^* n)$ expected time using only $n/\log^* n$ processors.

Implicit in each of the problems above is finding an *edge cover*. Using the random-adversary technique we show that finding an edge cover in $G_{n,p}$ from its adjacency matrix requires $\Omega(\log^* n)$ expected time on an n processor CRCW PRAM. MacKenzie and Stout [38] give a simpler proof specific to this problem, but we wish to show how the much more general random-adversary technique can produce the same lower bound.

1.3. Parity on CRCW PRAMs. The random-adversary technique can also prove a lower bound for computing parity using a randomized CRCW PRAM algorithm. The problem of parity is important in that lower bounds for computing parity (using a polynomial number of processors) provide lower bounds for sorting, bit summation, majority, and many other problems (see [4]).

Beame and Håstad [4] gave an $\Omega(\log n/\log \log n)$ lower bound for determinstic

CRCW PRAM algorithms, which matched the known upper bound. Using the results of Ajtai and Ben-Or [1] on converting randomized circuits into deterministic circuits, one could show that this lower bound also holds for randomized CRCW PRAM algorithms. This can also be shown more directly. Beame and Håstad show that given a uniform distribution of inputs, with nonzero probability an algorithm taking fewer than $\Omega(\log n/\log\log n)$ steps will output an incorrect answer, but Håstad [32] shows that in the circuit model, this can be extended to high probability. With minor adjustments, the high probability result also holds in the CRCW PRAM model, and it was known that by using Yao's theorem [41], one could obtain the equivalent lower bound for randomized algorithms over a worst case input. This is actually the first example of the random-adversary technique, though not in its most general form. (It would not be sufficient to prove many of the other lower bounds shown in this paper.)

1.4. Related results. The random-adversary technique has been used in MacKenzie [36] to prove lower bounds for computing some Boolean functions on exclusive-write PRAMs which asymptotically match the previous best lower bounds in Dietzfelbinger, Kutyłowski, and Reischuk [13] (and, in fact, improve the constants on some of those lower bounds) and to prove lower bounds on restricted domain compaction problems on exclusive-write PRAMs.

The random-adversary technique has also been used in Goldberg, Jerrum, and MacKenzie [23] to prove a lower bound for a fundamental routing problem on the optical communication parallel computer (as defined by Anderson and Miller [2] under the name "local memory PRAM"). We refer the reader to that paper for a full overview of the results and a discussion of the model and the specific routing problem.

Finally, the random-adversary technique has been used in MacKenzie [35] to prove a lower bound for compaction on the queue-read queue-write (QRQW) PRAM (see Gibbons, Matias, and Ramachandran [19] for a description of the QRQW PRAM).

2. Definitions. In this paper, we will be concerned only with the PRAM model. See [23] for the definition changes required for the optical communication parallel computer (OCPC) model. Relevant definitions for other models should be relatively easy to develop.

In the PRAM model, processors communicate by reading and writing to a global shared memory. The PRAM model is further subdivided depending on whether concurrent accesses are allowed to memory on reads and/or writes. An exclusive-read (ER) model does not allow concurrent reads to a memory cell, whereas a concurrent-read (CR) model does allow concurrent reads. An exclusive-write (EW) model does not allow concurrent writes to a memory cell, whereas a concurrent-write (CW) model does allow concurrent writes. On concurrent writes, there is also the issue of a contention protocol. In the priority CW model, if two or more processors write to any memory cell, the lowest-numbered processor succeeds in writing its value to that cell. (For more information on PRAM models see, for example, [34].)

Consider an algorithm with multiple inputs. An instantiation of the set of inputs to this algorithm will be called an *input map*. (This will be defined formally in the next section.) We define a *randomized algorithm* as one in which each processor can generate some number of random bits. In our lower bounds, we make no assumption on the number of random bits a processor can generate. The *expected time* of a randomized algorithm will be the average time of the algorithm over the distribution of input maps and the random bits. If we make no assumptions on the distribution of input maps, then the expected time will simply be the worst of the average times (taken over the random bits) for any input map.

We define *high probability* as meaning probability at least $1 - n^{-1}$, and we define *very high probability* as meaning probability at least $1 - e^{-n^{\alpha}}$ for some constant $\alpha > 0$.

For any base $z \geq 2$, we define $\log_z^{(1)} n$ as $\log_z n$ and $\log_z^{(i)} n$ as $\log_z(\log_z^{(i-1)} n)$. We define $\log_z^* n$ as the smallest integer i such that $\log_z^{(i)} n \leq 1$. When z is omitted we assume base 2. It is well known that $\log^* n$ is an extremely slow growing function of n, and, in fact, $\log^* n \leq 5$ for $n \leq 2^{65536}$. Lemma A.1 in the appendix shows a relation between $\log^*$ functions with different bases.

3. Random-adversary technique. The random-adversary technique allows one to prove a lower bound on the time required for a parallel randomized algorithm to solve a given problem. The first step of the technique is to decide on an input distribution for the problem. By Yao's theorem (see below), a lower bound on deterministic algorithms over this distribution provides the same lower bound for randomized algorithms.

The next step is to create a random adversary that proceeds through the given deterministic algorithm step by step, fixing some of the inputs in order to ensure some desired properties. (As shown below, this entails filling in the details of a procedure called REFINE.) Note that the random adversary is similar to a standard deterministic adversary in most parallel lower-bound proofs. However, unlike deterministic adversaries that can fix inputs arbitrarily, the random adversary must fix inputs according to the chosen input distribution, i.e., using the procedure RANDOMSET, as described below. Also, depending on how RANDOMSET fixes the inputs, the desired properties might not hold. Therefore, it is possible that the random adversary might have to make repeated calls to RANDOMSET to ensure the desired properties.

The final step is to show that these desired properties (such as knowledge about the inputs still being widely dispersed among the processors and that the number of inputs left unset is still large) hold with some given probability.

In the rest of this section we formalize this method.

3.1. Definitions. Let P be a problem with multiple inputs and let I be the set of inputs to P. Let Q be the set of possible values to which each input could be set. Define a *partial input map* to be a function f from I to $\{\{*\} \cup Q\}$. Here "$*$" will denote a "blank" or "unset" input. A partial input map is an *input map* if no inputs are mapped to "$*$." Let f_* denote the partial input map which maps every input to "$*$." A partial input map f' is called a *refinement* of a partial input map f if for all $i \in I$, and $q \in Q$, $f(i) = q$ implies $f'(i) = q$. (We denote this by $f' \leq f$.) If we wish to restrict our attention to a subset of the possible input maps, we would call that subset the *relevant input maps.* Likewise, we would say that a partial input map is a *partial relevant input map* if it has a refinement that is a relevant input map. We will often omit the word relevant when it is clear from the context.

3.2. Yao's theorem. The following theorem shows that a lower bound on the average time of a deterministic algorithm over any distribution of input maps implies the same lower bound for the expected time of any randomized algorithm over a worst-case input map or over the same distribution of input maps.

THEOREM 3.1 (see Yao [41]). *Let $\mathcal{D}$ be a distribution of input maps. Let T_1 be the expected running time for a randomized algorithm solving problem P, either over a worst-case input map or over the distribution $\mathcal{D}$. Let T_2 be the average running time over the distribution $\mathcal{D}$, minimized over all possible deterministic algorithms that solve P. Then $T_1 \geq T_2$.*

A nice proof of this theorem is given in Fich et al. [16].

Function RANDOMSET(f, S)
For each $i \in S$ (sequentially)
    Set $f(i)$ according to the conditional distribution of i given that
        the input is drawn from $\mathcal{D}$ and is a refinement of f
Return f
End RANDOMSET

FIG. 3.1. *The RANDOMSET function.*

This theorem greatly simplifies the problem of proving lower bounds for randomized algorithms, as it converts the original problem to one where the only randomness comes from the distribution of input maps, and this can often be set as one wishes. It is of course necessary to choose a distribution that will be difficult for any deterministic algorithm. Note that the distribution cannot place all the probability on one input map (i.e., a worst-case input map), since then a simple deterministic algorithm which checks for this input map and outputs the precomputed answer will succeed with probability 1.

3.3. RANDOMSET procedure. We will assume the distribution chosen is $\mathcal{D}$. Function RANDOMSET in Figure 3.1 can be used to randomly generate an input map one input at a time. It is called with a partial input map f obtained through calls to RANDOMSET and a set S of elements which are mapped to "$*$." The elements in S are then randomly set one by one according to the distribution $\mathcal{D}$, conditional on f.

CLAIM 3.2. *If f is generated solely by calls to RANDOMSET, then f will be generated according to the distribution $\mathcal{D}$.*

Proof. The proof is straightforward. □

3.4. REFINE and GENERATE. Say f is t-good if it satisfies certain properties, which will be defined with respect to the problem P and the input distribution $\mathcal{D}$. For some $T \leq n$, we would like to prove that the problem P cannot be solved in T steps. Let A be an algorithm which allegedly solves problem P over the input distribution $\mathcal{D}$ in T steps.

Given this algorithm A, we create a procedure REFINE, which tells the random adversary how to fix the inputs at each step. Formally, REFINE(t, f) takes a time t and a partial input map f and returns a new partial input map f' that is a refinement of f. We need to prove that the procedure REFINE has two important properties, the first of which is concerned with preservation of "t-goodness."

LEMMA 3.3. *If $t < T$ and* REFINE *is called with parameters (t, f), where f is t-good, then with probability at least $1 - n^{-2}$* REFINE *will return a partial input map f' that is $(t+1)$-good.*

The second property is that REFINE is unbiased. Consider the function GENERATE in Figure 3.2 that starts with the partial input map $f_0 = f_*$ and applies REFINE T times to generate a sequence of partial input maps $f_0 = f_* \geq f_1 \geq \cdots \geq f_T \geq f$ in which each $f_t = \text{REFINE}(t, f_{t-1})$ is a refinement of f_{t-1}, and f is an input map generated according to the conditional distribution over $\mathcal{D}$ from the set of refinements of f_T. Then we need to prove the following lemma.

LEMMA 3.4. *The input map f returned by* GENERATE *is generated according to the distribution $\mathcal{D}$.*

```
Function GENERATE
    Let f_0 = f_*
    Let f = f_0
    Let t = 1
    While t ≤ T Do
        If for some p, f(p) = "*" Then
            Let f_t = REFINE(t, f)
        Else
            Let f_t = f
        Let f = f_t
        Let t = t + 1
    Let P = {p | f(p) = "*"}
    Return RANDOMSET(f, P)
End GENERATE
```

FIG. 3.2. *The GENERATE function.*

In all REFINE procedures we construct in this paper, all inputs are set by calls to RANDOMSET. Consequently, by Claim 3.2, Lemma 3.4 will always hold.

Note that from Lemma 3.3 we also have the following.

LEMMA 3.5. *With probability at least* $1-n^{-1}$*, the partial input map* f_T *is* T*-good.*

Proof. Let Z_t be a binary random variable which is equal to 1 exactly when REFINE returns a t-good function at step t. Then the probability of failing at any step $t \leq T$ is

$$\sum_{t=1}^{T} \Pr(Z_t = 0 \mid Z_{t-1} = 1, \ldots, Z_1 = 1).$$

By Lemma 3.3, this is at most $Tn^{-2} \leq n^{-1}$. □

In summary, to fill in the random-adversary framework for a specific problem P, we must specify

1. an input distribution $\mathcal{D}$,
2. a definition for t-good,
3. a function REFINE,
4. a time T, and
5. a proof for Lemma 3.3.

Once this is done, we can use Lemmas 3.4 and 3.5, along with the specific definition of t-good, to prove the desired lower bound. Since this is highly problem dependent, we do not include it in the description of the general random-adversary technique.

3.5. Comparison to random restriction. In the random-restriction procedure first used in Furst, Saxe, and Sipser [18], a random set of inputs is randomly set according to a uniform distribution over binary inputs. The random adversary extends this idea to carefully chosen sets of inputs, and possibly other input distributions, some in which the inputs are not independent. Also, with random restrictions, there was never a notion of possibly setting more inputs in a single step. In the random adversary, the adversary is allowed to randomly set more inputs if the induction hypothesis does not yet hold.

A recent paper by Impagliazzo, Paturi, and Saks [33] gives a proof of a lower bound for the depth of threshold circuits computing majority using a technique similar to the random adversary, where inputs to be randomly set are chosen carefully. However, the inputs are not set according to any predefined distribution. Thus Yao's theorem would not apply, and this would not translate into a lower-bound proof for randomized threshold circuits.

4. PRAM-specific definitions. Let A be any deterministic algorithm for an n processor PRAM, and let f be any input map. $\text{Trace}(p, 0, f)$ is defined to be the tuple $\langle p \rangle$. $\text{Trace}(p, t, f)$ (for $t > 0$) is defined to be the tuple $\langle p, \lambda_1, \ldots, \lambda_t \rangle$ in which λ_j is the contents of the cell read in step j, if any, and λ_j is the null symbol otherwise. Similarly, $\text{Trace}(c, 0, f)$ is defined to be the tuple $\langle c, \lambda_0 \rangle$, where λ_0 is the initial value in cell c. $\text{Trace}(c, t, f)$ (for $t > 0$) is defined to be the tuple $\langle c, \lambda_0, \ldots, \lambda_t \rangle$ in which λ_j is the contents of the cell after step j.

For the following, assume that v is either a processor or a cell. Let g be any relevant partial input map. We will define $\text{Know}(v, t, g)$ as the smallest set of inputs such that the following property PK is satisfied: for any relevant input maps f_1 and f_2 that refine g and have $f_1(q) = f_2(q)$ for all $q \in \text{Know}(v, t, g)$, $\text{Trace}(v, t, f_1)$ is the same as $\text{Trace}(v, t, f_2)$. (Intuitively, v is not dependent on inputs outside $\text{Know}(v, t, g)$, since these could not affect its trace, and v is dependent on every input inside $\text{Know}(v, t, g)$ by the fact that it is the minimum set of inputs which could affect its trace.)

CLAIM 4.1. *For all v, t, g, Know(v, t, g) exists and is unique.*

Proof. $\text{Know}(v, t, g)$ exists since at least one set (the set of all inputs) satisfies PK, and there are a finite number of sets from which the smallest can be chosen.

$\text{Know}(v, t, g)$ is unique since if two sets satisfy PK, then their intersection satisfies PK (i.e., there cannot be two different smallest sets, since the intersection of them is smaller than either and also satisfies PK). To see this consider two sets A and B which satisfy PK. Consider two input maps f_1 and f_2 that refine g and have $f_1(q) = f_2(q)$ for all $q \in A \cap B$. Define f_3 as follows.

$$f_3(q) = \begin{cases} f_1(q) & \text{for } q \in A, \\ f_2(q) & \text{for } q \notin A. \end{cases}$$

Then by the property PK, $\text{Trace}(v, t, f_1) = \text{Trace}(v, t, f_3)$, and $\text{Trace}(v, t, f_3) = \text{Trace}(v, t, f_2)$. Thus $\text{Trace}(v, t, f_1) = \text{Trace}(v, t, f_2)$. □

Let $\text{AffProc}(i, t, g)$ contain each processor p for which $i \in \text{Know}(p, t, g)$. Let $\text{AffCell}(i, t, g)$ contain each cell c for which $i \in \text{Know}(c, t, g)$.

5. General CRCW PRAM lower bound. Before we prove a lower bound on load balancing, we will prove a general lower bound on the amount of information which can be transferred between processors given a general random input.

Assume the set of possible values for inputs $Q = \{v_1, \ldots, v_{|Q|}\}$. The input distribution we will use is that each input is independently assigned value v_j ($1 \leq j \leq |Q|$) with probability p_j for some $p_1, \ldots, p_{|Q|}$ with $p_1 + \cdots + p_{|Q|} = 1$. Let $p_0 = \min\{p_1, \ldots, p_{|Q|}\}$, and $P = 1/p_0$.

We now define the following constants and functions of n: $k_0 = 1$, for $i > 0$ $k_i = (2400P^3)^{k_{i-1}}$, $r_0 = 0$, and for $i \geq 1$, $r_i = (n/8)\sum_{j=1}^{i} 2^{-j}$. Let $T = \log^* n - \log^*(2400P^3) - 3$. Then the following facts are easily proved (use Lemma A.1 for Fact 5.1).

FACT 5.1. $k_T \leq \log \log n$.

FACT 5.2. *For all $i \geq 1$, $r_i \leq n/8$.*

```
    Function REFINE(t, f)
(1)     Let g = f
(2)     For each c ∈ L
(3)         Let g = RANDOMSET(g, Know(c, t, g))
(4)         Let W = WRITE(c, g)
(5)         For each p ∈ W (in order by proc. number)
(6)             Let g = RANDOMSET(g, Know(p, t, g))
(7)             If p does not write to c
(8)                 Next p
(9)         Next c
(10)    Let f' = g
(11)    Return f'
End REFINE
```

FIG. 5.1. *The REFINE procedure.*

A partial input map f is called *t-good* if the following three conditions are satisfied.

1. For each processor or cell v, $|\text{Know}(v, t, f)| \leq k_t$.
2. For each input i, $|\text{AffProc}(i, t, f)| \leq k_t$ and $|\text{AffCell}(i, t, f)| \leq k_t$.
3. f maps at most r_t inputs to something other than "$*$."

We now describe the algorithm REFINE (shown in Figure 5.1) which is called with a time t and a partial input map f, and which returns a partial input map f' which is a random refinement of f. This random refinement is based on the action of algorithm A at step $t+1$. (A step is assumed to be a write followed by a read.) Let U be the set of inputs that f maps to "$*$." Let U' be the set of inputs that f' maps to "$*$." Say a processor p *write-affects* a cell c with a partial input map g if for some input map g' which refines g, p writes to cell c at step $t+1$. Say a cell c *read-affects* a processor p with a partial input map g if for some input map g' which refines g, p reads from cell c at step $t+1$. Let $\text{WRITE}(c, g)$ be the set of processors that write-affect cell c with a partial input map g, and let $\text{READ}(c, g)$ be the set of processors that c read-affects with a partial input map g. Let L be the set of cells for which $|\text{WRITE}(c, f)| \geq (400P^2)^{k_t}$ or $|\text{READ}(c, f)| \geq (400P^2)^{k_t}$. In algorithm REFINE, we simply go through the processors (in order by processor number) which write-affect (with the current partial input map) each cell in L, and randomly set the inputs in their knowledge sets until one is sure to write or none of them write. Note that at step (4), we use the set $\text{WRITE}(c, g)$ instead of $\text{WRITE}(c, f)$. Since $g \leq f$, $\text{WRITE}(c, g) \subseteq \text{WRITE}(c, f)$, but the sets may not be equal. We must use $\text{WRITE}(c, g)$ so as not to set all the inputs in a set $\text{Know}(p, t, g)$ if p has already been forced not to write to cell c by previous refinements to f.

We say REFINE *fixes* a processor p if REFINE executes step (6) for processor p.

The following claim formally proves the intuitive idea that when the inputs that affect the state of a processor or cell are all fixed, then the state of the processor or cell is fixed.

CLAIM 5.3. *Let f be a partial input map, and let v be a processor or cell. Then if f' refines f and for all $i \in \text{Know}(v, t, f)$, $f'(i)$ does not equal "$*$," then $|\text{Know}(v, t, f')| = 0$.*

Proof. Take any input maps f_1 and f_2 which refine f', Then f_1 and f_2 also refine f and for every input $q \in \text{Know}(v, t, f)$, $f_1(q) = f_2(q)$. By the definition of

Know(v,t,f), this implies that Trace(v,t,f_1) = Trace(v,t,f_2). Thus the minimum set that satisfies the necessary conditions for Know(v,t,f') is the empty set. □

The following claim formally proves that REFINE causes the contents of each cell $c \in L$ to become fixed. It does this by fixing the contents of c previous to the step and then fixing processors that possibly write to c one-by-one until either (1) a processor writes a fixed value to the cell and no lower numbered processor writes to the cell, or (2) no processors write to the cell.

CLAIM 5.4. *If f is t-good, and* REFINE(t,f) *returns f', then for every $c \in L$, $|Know(c,t+1,f')| = 0$.*

Proof. From Step 3 of algorithm REFINE and Claim 5.3, |Know$(c,t,f')| = 0$. Let p be the lowest-numbered processor that REFINE causes to write to c, if any. Otherwise, let $p = \infty$. Let g be the partial input map computed just before the set W is found in Step 4 for cell c. Note that for each processor $p' \in$ WRITE(c,g) with $p' \leq p$, |Know$(p',t,f')| = 0$, and thus the action of p' is fixed at step $t+1$ for any input map which refines f'. If $p = \infty$, then for any input map which refines f', no processor writes to c (i.e., no processor affects Trace$(c,t+1,f')$), and thus Know$(c,t+1,f')$ will contain only those inputs in Know(c,t,f'), which is empty. If $p \neq \infty$, then the contents of c will contain the value written by p (i.e., only the inputs in the Know(p,t,f') could affect the Trace$(c,t+1,f')$), and thus Know$(c,t+1,f')$ will contain only those inputs in Know(p,t,f'), which is empty. □

Recall that $U' = \{i \in I | f'(i) = \text{“}*\text{”}\}$.

LEMMA 5.5. *If f is t-good and* REFINE(t,f) *returns f', then* (1) *for each cell c, $|Know(c,t+1,f')| \leq k_{t+1}$;* (2) *for each processor p, $|Know(p,t+1,f')| \leq k_{t+1}$;* (3) *for each input $i \in U'$, $|AffCell(i,t+1,f')| \leq k_{t+1}$; and* (4) *for each input $i \in U'$, $|AffProc(i,t+1,f')| \leq k_{t+1}$.*

Proof. By Claim 5.4, for each cell $c \in L$, Know$(c,t+1,f')$ contains no inputs. For each cell c not in L, Know$(c,t+1,f')$ contains at most those inputs in Know(c,t,f) and those inputs in Know(p,t,f) for each processor p of the $(400P^2)^{k_t}$ processors in WRITE(c,f), and thus affect Trace$(c,t+1,f')$. Therefore |Know$(c,t+1,f')| \leq k_t + k_t(400P^2)^{k_t}$.

For a processor p, let C be the set of cells c such that $p \in$ READ(c,f). Then $|C| \leq |Q|^{k_t}$, since that is the maximum number of possible settings of inputs in Know(p,t,f). Know$(p,t+1,f')$ contains at most the inputs in Know(p,t,f) plus the inputs in Know$(c,t+1,f')$ for each $c \in C$. Therefore |Know$(p,t+1,f')| \leq k_t + |Q|^{k_t}(k_t + k_t(400P^2)^{k_t})$.

For an input $i \in U'$, let P' be the set of processors p for which $i \in$ Know(p,t,f). Then $|P'| \leq k_t$. For a processor $p \in P'$, let C_p be the set of cells c such that $p \in$ WRITE(c,f). Then $|C_p| \leq |Q|^{k_t}$. Thus i could be contained in Know$(c,t+1,f')$ for at most those cells $c \in \bigcup_{p \in P'} C_p$ and each of the at most k_t cells c for which $i \in$ Know(c,t,f). Then i is contained in Know$(c,t+1,f')$ for at most $k_t + k_t|Q|^{k_t}$ cells c.

For an input $i \in U'$, let C be the set of cells c for which $i \in$ Know$(c,t+1,f')$. Then $|C| \leq k_t + k_t|Q|^{k_t}$. Note that $C \cap L = \emptyset$. For a cell $c \in C$, let P_c be the set of processors p such that $p \in$ READ(c,f). Then $|P_c| \leq (400P^2)^{k_t}$. Then i could be contained in Know$(p,t+1,f')$ for at most those processors $p \in \bigcup_{c \in C} P_c$ and the at most k_t processors p for which $i \in$ Know(p,t,f). Then i is contained in Know$(p,t+1,f')$ for at most $k_t + (k_t + k_t|Q|^{k_t})(400P^2)^{k_t}$ processors p.

The four claims in the lemma then follow from the fact that $1 \leq |Q| \leq P$ and Lemma A.2. □

We say that REFINE(t, f) is *successful* if it calls RANDOMSET with at most $\frac{n}{8(2^{t+1})}$ inputs.

LEMMA 5.6. *If f is t-good and REFINE is successful, then f' is $t+1$-good.*

Proof. This follows from Lemma 5.5 and the fact that

$$r_{t+1} - r_t = \frac{n}{8(2^{t+1})}. \qquad \square$$

Without loss of generality, let the cells $c \in L$ be numbered 1 to $|L|$ in the order in which they are processed by REFINE. Let $f'_0 = f$, and let f'_c be the partial input map obtained after REFINE is finished with cell c.

Let M be the set of partial input maps which could possibly be obtained after REFINE is finished with cell $c-1$. For any $g \in M$, let $W_{c,g}$ be the set of processors in WRITE(c, g) and assume without loss of generality that these processors are numbered from 0 to $|W_{c,g}| - 1$. Construct a set $S_{c,g}$ from $W_{c,g}$ inductively by inserting into $S_{c,g}$ the lowest-numbered processor p from $W_{c,g}$ such that Know(p, t, g) is disjoint from Know(p', t, g) for every processor p' already in $S_{c,g}$. Say the jth processor inserted into $S_{c,g}$ has *rank* j.

CLAIM 5.7. *The number of the jth processor inserted into $S_{c,g}$ is at most $k_t^2(j-1)$.*

Proof. Let p be one of the first $j-1$ processors inserted into $S_{c,g}$. Then $|\text{Know}(p, t, g)| \le k_t$, and since for each $i \in \text{Know}(p, t, g)$, $|\text{AffProc}(i, t, g)| \le k_t$, i is contained in Know(p', t, g) for at most k_t processors p'. In total for processor p, Know$(p, t, g) \cap$ Know$(p', t, g) \neq \emptyset$ for at most k_t^2 processors p'. Then at most $k_t^2(j-1)$ processors p' have Know$(p, t, g) \cap$ Know$(p', t, g) \neq \emptyset$ for some p which is one of the first $j-1$ processors inserted into $S_{c,g}$. $\square$

Let $[g]$ be the event that g is the partial input map obtained after REFINE is finished with cell $c-1$. For a set $G \subseteq M$, let $[G]$ be the event that the partial input map obtained after REFINE is finished with cell $c-1$ is contained in G. Let X_c be the random variable denoting the number of processors in $S_{c,g}$ that REFINE fixes while fixing cell c. (Note that the probability distribution of X_c is taken over the calls to RANDOMSET which define which $g \in M$ is used, and over the calls to RANDOMSET for fixing the processors in $S_{c,g}$.) Let Y be a random variable with a geometric distribution with parameter P^{-k_t}.

CLAIM 5.8. *For any real $a \ge 1$ and any $g \in M$ with* $\Pr([g]) > 0$, *$Pr(X_c > a|[g]) \le Pr(Y > a)$.*

Proof. Let $j = \lfloor a \rfloor$. Given $[g]$, the probability that REFINE has not forced any processors numbered no larger than the jth ranked processor in $S_{c,g}$ to write is less than the probability that it has not forced the first j processors in $S_{c,g}$ to write. The probability of the processor of rank i writing given that the processors of ranks 1 through $i-1$ did not write is at least P^{-k_t}, since they are independent, and RANDOMSET sets the at most k_t inputs in Know(i, t, g) randomly. Then

$$\Pr(X_c > a|[g]) \le \prod_{i=1}^{j}(1 - P^{-k_t}) = (1 - P^{-k_t})^j = \Pr(Y > a). \qquad \square$$

CLAIM 5.9. *For any real $a \ge 1$ and any $G \subseteq M$ with* $\Pr([G]) > 0$, *$Pr(X_c > a|[G]) \le Pr(Y > a)$.*

Proof. Note that for any real a, the event $(X_c > a) \wedge [G]$ can be formulated as

$$\bigvee_{g \in G} ((X_c > a) \wedge [g]).$$

Also note that the terms in this disjunction are disjoint. Then using Claim 5.8,

$$\begin{aligned}\Pr(X_c > a|[G]) &= \frac{\Pr((X_c > a) \wedge [G])}{\Pr([G])} \\ &= \frac{\sum_{g \in G} Pr(X_c > a \wedge [g])}{\Pr([G])} \\ &= \frac{\sum_{g \in G} Pr(X_c > a|[g]) \Pr([g])}{\Pr([G])} \\ &\leq \Pr(Y > a) \frac{\sum_{g \in G} \Pr([g])}{\Pr([G])} \\ &= \Pr(Y > a). \qquad \square\end{aligned}$$

For any natural numbers $b_1, \ldots, b_{c-1}$, let $G_{b_1,\ldots,b_{c-1}}$ be the set of partial input maps for which $X_1 = b_1, \ldots, X_{c-1} = b_{c-1}$.

LEMMA 5.10. *Given independent random variables $Y_1, \ldots, Y_{|L|}$ with geometric distributions with parameter P^{-k_t},*

$$Pr(X_1 + \cdots + X_{|L|} > a) \leq Pr(Y_1 + \cdots + Y_{|L|} > a).$$

Proof. Using Claim 5.9, for any natural numbers $b_1, \ldots, b_{c-1}$, if $\Pr([G_{b_1,\ldots,b_{c-1}}]) > 0$,

$$\begin{aligned}Pr(X_c > a|X_1 = b_1, \ldots, X_{c-1} = b_{c-1}) &= Pr(X_c > a|[G_{b_1,\ldots,b_{c-1}}]) \\ &\leq Pr(Y > a).\end{aligned}$$

Then the lemma follows from Theorem B.1. $\square$

LEMMA 5.11. *If f is t-good then* REFINE(t, f) *is successful with probability at least $1 - n^{-2}$.*

Proof. From Claim 5.7, given that REFINE fixes $X_1 + \cdots + X_{|L|}$ processors in the sets $S_{c,g}$, $(1 \leq c \leq |L|)$, it calls RANDOMSET with at most $k_t^3(X_1 + \cdots + X_{|L|})$ inputs (at most k_t inputs from Know(p, t, g) for each processor $p \in W_{c,g}$ is fixed).

Let $m = |L|$ and let $m^* = 2nP^{k_t}/(400P^2)^{k_t}$. Note that by the definition of the set L, $m \leq m^*$.

From Lemma 5.10 and using the Chernoff bound for sums of geometric distributions given in the appendix to bound $Y_1 + \cdots + Y_L$ (with $\beta = \frac{3m^*}{L} - 1 \geq 2$, $p = 1 - q = P^{-k_t}$, and the fact that $\frac{1}{2} \leq q < 1$), we see that

$$\begin{aligned}Pr(X_1 + \cdots + X_L > 3m^*(1 - P^{-k_t})P^{k_t}) &\leq Pr(Y_1 + \cdots + Y_L > 3m^*(1 - P^{-k_t})P^{k_t}) \\ &\leq e^{-m^*/4},\end{aligned}$$

for $k_t \geq 1$. Note that for sufficiently large n, when $k_t \leq \log\log n$ and $P \leq \log n$, $e^{-m^*/4} \leq e^{-\sqrt{n}}$. With very high probability then, the number of inputs set by RANDOMSET in this step is bounded by

$$\begin{aligned}k_t^3(3m^*P^{k_t}) &\leq \frac{3k_t^3 nP^{2k_t}}{(400P^2)^{k_t}} \\ &\leq \frac{n}{8(2^{k_t})}\end{aligned}$$

for $k_t \geq 1$. Since $k_0 = 1$, and k_t grows much faster than t, it is easy to see that $k_t \geq t+1$ for all $t \geq 0$. Thus we have shown that with very high probability, at most $n/8(2^{t+1})$ processors are set by RANDOMSET at step t. □

LEMMA 5.12. *If $t < T$ and* REFINE *is called with parameters (t, f), where f is t-good, then with probability at least $1 - n^{-2}$* REFINE *will return a partial input map f' that is $(t+1)$-good.*

Proof. This follows from Lemma 5.11 and Lemma 5.6. □

From Lemma 5.12, Fact 5.1, and Fact 5.2 we obtain the following corollary.

COROLLARY 5.13. *For any processor or cell p, $|Know(p, T, f_T)| \leq k_T \leq \log\log n$, and the number of inputs set by RANDOMSET is at most $r_T \leq n/8$ with probability at least $1 - n^{-1}$.*

6. Load-balancing lower bound. Instead of directly proving a lower bound on the general load-balancing problem, it will be convenient to prove a lower bound on the following variation of the load-balancing problem.

Chromatic load balancing (CLB). Let $m \geq 1$, and let Q be a set of $8m$ colors. Assume that there is a set of n groups of $4m$ objects each, and each group of objects is randomly assigned a color from Q. Then the *chromatic load-balancing* problem is to choose any color and distribute the objects of that color into n groups of at most m objects.

Without loss of generality, we will assume the objects are tagged with their original group number and their original rank (1 to $4m$) within that group.

Enhanced chromatic load balancing (ECLB). The *enhanced chromatic load-balancing* problem is the same as the CLB problem with the added requirement that one must produce an $n \times 4m$ array of pointers such that in each row corresponding to a group of the chosen color, and each column corresponding to the rank of a given object, there must be a pointer to the destination group of the object.

CLAIM 6.1. *Given a solution to the CLB problem, one can construct a solution to the ECLB problem on a priority CRCW PRAM in m additional steps.*

Proof. Assign one processor per destination group (of the CLB solution) to step through the at most m objects assigned to that group. For each object with tag (group, rank), have the processor write that destination in the array at location (group, rank). □

LEMMA 6.2. *For $m = o(\log^* n)$, a deterministic algorithm which solves the ECLB problem on a priority CRCW PRAM requires $\Omega(\log^* n)$ expected time.*

Proof. Assume we have finished step T of a deterministic algorithm that solves the ECLB problem. Then $k_T \leq \log\log n$ and $r_T \leq n/8$. Consider a color $q \in Q$. Consider a group g that has not been assigned a color by f_T (i.e., for such a group i, $f_T(i) =$ "$*$") and an object o in group g. Let c be the cell holding the array location (g, o) in the pointer array. Consider the contents of c (the pointer in (g, o)) assuming that the input map which was refined from f_T assigned the color q to all inputs (groups) in Know(c, T, f_T). Do this for all objects in all such groups that have not been assigned a color by f_T. This defines a *potential object map* F. Then F is a function with a domain of size at least $(7n/8)(4m) = 28nm/8$ and a range of size at most n. By a simple counting argument, we can find n disjoint sets of $m+1$ cells, all of which point to the same destination group. (To see this, consider iteratively removing sets of $m+1$ cells that point to the same destination group. One can do this until there are at most m cells pointing to any destination group, or at most nm leftover cells. Then the number of sets removed is at least $(28nm/8 - nm)/(m+1) \geq n$.)

Let S be one of the disjoint sets of cells that we have just found. Each of the $m+1$

cells $c \in S$ have $\text{Know}(c, T, f_T) \leq k_T \leq \log\log n$, and thus at most $(m+1)\log\log n$ inputs affect the contents of these cells. Each of these inputs is in $\text{Know}(c', T, f_T)$ for at most $\log\log n$ other cells c', so at most $(m+1)(\log\log n)^2$ cells are affected by the same inputs that affect the cells in set S. Thus from the n disjoint sets of $m+1$ cells which are mapped by F to the same processor, we can find a subset of $B = \lfloor n/(m+1)(\log\log n)^2 \rfloor$ sets whose cell contents are completely independent. Number these sets from 1 to B.

CLAIM 6.3. *With very high probability, at least one of these B sets uses the same pointers as F.*

Proof. We can see that the probability of all cells c in one of these sets using the pointers from F is at least the probability that f maps all inputs in $\text{Know}(c, T, f_T)$ for each c in the set to the color q, which is at least $1/(8m)^{(m+1)\log\log n}$.

Now for all sets i, let X_i be a random variable which is one if this event occurs, and zero otherwise. Let $X_i' = 1 - X_i$. Let Y_i be a random variable which is one with probability $1/(8m)^{(m+1)\log\log n}$, and zero otherwise. Let $Y_i' = 1 - Y_i$. Then for all sets i and any real number a, $Pr(X_i' > a) \leq Pr(Y_i' > a)$. Since the X_i''s are independent, they obviously satisfy the conditions of Theorem B.1, and thus

$$Pr(X_1' + \cdots + X_B' > a) \leq Pr(Y_1' + \cdots + Y_B' > a).$$

The number of sets in which this event occurs can then be bounded from below using the Chernoff bound for binomial random variables given in the appendix, as follows.

$$\begin{aligned}
&Pr(X_1 + \cdots + X_B < B/2(8m)^{(m+1)\log\log n}) \\
&\leq Pr(X_1' + \cdots + X_B' > B - B/2(8m)^{(m+1)\log\log n}) \\
&\leq Pr(Y_1' + \cdots + Y_B' > B - B/2(8m)^{(m+1)\log\log n}) \\
&\leq Pr(Y_1 + \cdots + Y_B < B/2(8m)^{(m+1)\log\log n}) \\
&\leq e^{-B/8(8m)^{(m+1)\log\log n}} \\
&= \exp\left\{ - \left\lfloor \frac{n}{(m+1)(\log\log n)^2} \right\rfloor \frac{1}{8(8m)^{(m+1)\log\log n}} \right\}.
\end{aligned}$$

For sufficiently large n, this implies a very high probability bound. Also for sufficiently large n, $B/2(8m)^{(m+1)\log\log n} \geq 1$, and thus with very high probability, at least one set of $m+1$ objects will be mapped to the same processor. □

By Claim 6.3, with very high probability the mapping provided by the algorithm for the color q at this point will not be a valid solution to the ECLB problem. (Remember we required that at most m objects map to any one destination group.)

Since with very high probability the mapping will be invalid for any color we choose, and since there are only $8m$ colors, with very high probability there will be no valid solution to the ECLB problem. The lemma follows since $T = \Omega(\log^* n)$. □

COROLLARY 6.4. *For $m = o(\log^* n)$, a deterministic algorithm which solves the CLB problem on a priority CRCW PRAM requires $\Omega(\log^* n)$ expected time.*

Proof. This follows from Claim 6.1 and Lemma 6.2. □

THEOREM 6.5. *Solving the* load-balancing *problem on a randomized priority CRCW PRAM requires $\Omega(\log^* n)$ expected time.*

Proof. Assume there is an algorithm that solves load balancing in expected time t. Then by Yao's theorem, for any input distribution, there is a deterministic algorithm which solves load balancing over that distribution in expected time t. Consider the

chromatic load-balancing problem (with $m = \log\log^* n$) and choose one of the $8m$ colors. (We choose $\log\log^* n$ because it increases with n and is $o(\log^* n)$, as required in Corollary 6.4.) Let $\mathcal{D}$ be the input distribution of the objects of that color, and let A be the algorithm given by Yao's theorem for distribution $\mathcal{D}$. We can then solve the chromatic load-balancing problem using the following procedure. First run A for the objects of the chosen color. Without loss of generality, for some constant C assume A assigns at most $C(1 + h/n)$ objects to each processor, when h objects are given as input. Also assume n is large enough so that $2C < m$. If at most m objects are assigned to each processor then one can easily assign each processor's objects to a destination group. If not, then run a $\Theta(\log n)$ time algorithm to solve chromatic load balancing (for example, using prefix operations).

We now analyze the expected time of this procedure. On average, there will be $4nm/8m$ objects of the chosen color, and with very high probability, there will be at most n objects of that color. If there are at most n objects of that color, at most $2C$ of them will be assigned to any one processor by A. Because of the very high probability of this occurring, the $\Theta(\log n)$ time algorithm (that is run when more than m objects are assigned to any processor) will not asymptotically increase the expected time of the procedure. Thus the chromatic load-balancing problem can be solved in the same asymptotic expected time as A, and by Corollary 6.4, $t = \Omega(\log^* n)$. □

7. Related lower bounds. In this section, we show further applications of the lower-bound technique.

THEOREM 7.1. *Solving* compaction *on a randomized priority CRCW PRAM requires* $\Omega(\log^* n)$ *expected time.*

Proof. Assume there is an algorithm that solves compaction in expected time t. Then by Yao's theorem, for any input distribution, there is a deterministic algorithm which solves compaction over that distribution in expected time t. Consider the chromatic load-balancing problem (with $m = \log\log^* n$) and choose one of the $8m$ colors. Consider an item to be a group of objects of that color, and let $\mathcal{D}$ be the input distribution of the items. Let A be the algorithm given by Yao's theorem for distribution $\mathcal{D}$. We can then solve the chromatic load-balancing problem using the following procedure. First run A with $h = n/4m$. Without loss of generality, for some constant C assume A inserts the items into an array of size Ch, when h is the parameter given in the definition of compaction and the input consists of at most h items. Also assume n is large enough so that $C < m$. If A succeeds, then one can easily assign each item to four destination groups (i.e., m objects to each destination group), and this solves the chromatic load-balancing problem. If A does not succeed, then run a $\Theta(\log n)$ time algorithm to solve chromatic load balancing (for example, using prefix operations).

We now analyze the expected time of this procedure. On average, there will be $n/8m$ items, and with very high probability, there will be at most $n/4m$ items. If there are at most $n/4m$ items, then A will succeed. Because of the very high probability of this occurring, the $\Theta(\log n)$ time algorithm (that is run when A fails) will not asymptotically increase the expected time of the procedure. Thus the chromatic load-balancing problem can be solved in the same asymptotic expected time as A, and by Corollary 6.4, $t = \Omega(\log^* n)$. □

THEOREM 7.2. *Solving the* padded-sort *problem on a randomized priority CRCW PRAM requires* $\Omega(\log^* n)$ *expected time.*

Proof. (We actually prove that this lower bound holds for sorting into any array of size linear in n, not just $n + o(n)$.) We can reduce the chromatic load-balancing

problem (with $m = \log\log^* n$) to the padded-sort problem with no asymptotic increase in running time as follows. Assign the colors individual integers from 0 to $8m-1$. For each group with color i, uniformly choose a random real number from the range $(i/8m, (i+1)/8m]$. Thus each group will be assigned a number from $(0,1]$ and these will be uniformly distributed. Now assume we have a padded-sort algorithm which will place these numbers in sorted order into an array A of size kn for some constant k. Run this padded-sort algorithm. Next we find a color which is mapped to $\leq 3kn/8m$ consecutive positions in constant time as follows.

Assign one processor to each group. Say a processor is of color i if it is assigned to a group of color i. For each $i \in \{0,\ldots,8m-1\}$ perform the following steps. Each processor of color i writes its group's position in A to position i in a new array $P[0\,..\,8m-1]$. (It can easily be detected if a color i has no representative processors, and in that case, color i can be used as a trivial solution to the chromatic load-balancing problem.) Then each processor of color i subtracts $P[i-1]$ from $P[i+1]$ and checks if the result is $\leq 3kn/8m$. Assuming all the colors are present, then one color will find its result is $\leq 3kn/8m$. (Otherwise, every set of three consecutively numbered colors is mapped into $> 3kn/8m$ consecutive positions, and thus $|A| > (3kn/8m)(8m/3) = kn$.) Each processor of color i whose check succeeds writes i to a variable V. We choose the color corresponding to the contents of V.

Assume this color is j. Since $3kn/8m \leq n/4$ for large enough n, we know that the groups of color j are mapped into $A[P[j-1], P[j-1]+n/4]$, so we can easily assign four destination groups to each group of color j (i.e., m objects to each destination group). This solves the chromatic load-balancing problem. ∎

8. Random graphs. The input will consist of an adjacency matrix with a 1 entry if an edge exists, and a 0 entry otherwise. Thus $Q = \{0,1\}$. For a random graph $G_{n,p}$ there will be will be $\binom{n}{2}$ inputs, and the input distribution will be that each input is 1 with probability p and 0 with probability $1-p$.

Let T, the function REFINE, and the notion of t-good be defined as in the general CRCW PRAM lower bound. Then Lemma 5.12 holds.

THEOREM 8.1. *For any constant $0 < p < 1$, any problem which requires the construction of an edge cover (with high probability) in a random graph $G_{n,p}$ represented as an adjacency matrix requires $\Omega(\log^* n)$ expected time on an n processor CRCW PRAM.*

Proof. Assume there is an algorithm that finds an edge cover in a random graph $G_{n,p}$ in expected time t. Then by Yao's theorem, there is a deterministic algorithm that finds an edge cover in a random graph $G_{n,p}$ in the same expected time t. Thus a lower bound for deterministic algorithms will provide the same lower bound for any (randomized) algorithm.

Assume we have finished step T of a deterministic algorithm that finds an edge cover in a random graph $G_{n,p}$. Assume the edge cover is given in an output array $EC[1..n]$ of size n such that $EC[i] = j$ if (i,j) is the alleged edge covering vertex i. (For the rest of the proof, we will also use the index i to denote the memory cell corresponding to the array element $EC[i]$.) By Lemma 5.12, $k_T \leq \log\log n$ and $r_T < n/8$. Consider all the vertices which are only covered by edges which f maps to "$*$." There will be at least $3n/4$ of these. We will find a subset of the array indices corresponding to these vertices for which the sets $\mathrm{Know}(i, T, f_T)$ are disjoint. To do this, we will use a greedy algorithm and obtain a set S of size

$$\frac{3n}{4k_T^2} \geq \frac{3n}{4(\log\log n)^2}.$$

Now consider for each index $i \in S$ the value $EC[i]$, if $\text{Know}(i, T, f_T)$ contains only inputs set to 0 by the final input map f. Then the existence of edge $(i, EC[i])$ does not affect $EC[i]$ (assuming the algorithm doesn't choose an edge that is known not to exist).

The probability of this occurring for any index in S is $\geq (1-p)^{\log\log n}$ and is independent of any other cells in S. Thus the average number of indices j in which the existence of $(j, EC[j])$ does not affect $EC[j]$ will be at least

$$\frac{3n}{4(\log\log n)^2}(1-p)^{\log\log n} = \Omega(\sqrt{n}).$$

Assuming n is large, and using a Chernoff bound, we can show that this occurs for at least half of these indices with high probability. Thus $\Omega(\sqrt{n})$ vertices will have a $1-p$ chance of not being covered. Since an edge can only possibly cover two vertices, at most two vertices will point to the same alleged edge. Thus $\Omega(\sqrt{n})$ vertices point to different alleged edges. Then using another Chernoff bound, it is easy to show that with high probability, many of these will point to nonexistent edges. Thus with high probability the edge cover is invalid. Since $T = \Omega(\log^* n)$, the expected time of this deterministic algorithm must be $\Omega(\log^* n)$. □

COROLLARY 8.2. *For any constant $0 < p < 1$, in a random graph $G_{n,p}$ represented as an adjacency matrix, constructing a Hamiltonian cycle, spanning tree, or maximal matching requires $\Omega(\log^* n)$ expected time on an n processor CRCW PRAM.*

Proof. With high probability a Hamiltonian cycle, a spanning tree, and a maximal matching exist in $G_{n,p}$, and constructing any of these implies construction of an edge cover. Thus by Theorem 8.1, the stated lower bound holds. □

9. Conclusion. We have developed a technique which provides lower bounds on randomized PRAM algorithms. Using this technique, we have been able to prove tight lower bounds for many problems, including the fundamental problem of load balancing, even on the most powerful CRCW PRAM model. In view of the increasing amount of attention being paid to the area of fast randomized algorithms on the PRAM, we believe this lower-bound technique is very important. We hope that this general technique can be used to prove lower bounds on other problems for which very fast randomized algorithms have been developed, including integer chain-sorting [27] and parallel hashing [21].

Appendix A. Technical lemmas.

LEMMA A.1. *For $z \geq 4$, $\log^* n \leq \log_z^* n + \log^* z$.*

Proof. First we claim that for $z \geq 4$, $2\log z \leq z$. This is true as long as $z - 2\log z \geq 0$. Let $f(z) = z - 2\log z$. Then $f'(z) = 1 - (2\log e)/z$. We can check that $f(4) = 0$ and $f'(z) > 0$, for $z \geq 4$, which proves the claim.

Now we claim that $\log^*(z \log_z^{(\log_z^* n - k)} n) \leq k + \log^* z$, for $1 \leq k \leq \log_z^* n - 1$ and $z \geq 4$. We show this by induction. For $k = 1$,

$$\log^*(z \log_z^{(\log_z^* n - 1)} n) \leq \log^* z^2 \leq \log^* 2^z = 1 + \log^* z.$$

For $k > 1$,

$$\begin{aligned}\log^*(z \log_z^{(\log_z^* n - k)} n) &= \log^* z^{1+\log_z^{(\log_z^* n - k+1)} n} \\ &= \log^* 2^{\log z(1+\log_z^{(\log_z^* n - k+1)} n)} \\ &\leq \log^* 2^{2\log z \log_z^{(\log_z^* n - k+1)} n}\end{aligned}$$

$$\begin{aligned}
&= 1 + \log^* \left(2 \log z \log_z^{(\log_z^* n - k + 1)} n\right) \\
&\leq 1 + \log^* \left(z \log_z^{(\log_z^* n - k + 1)} n\right) \\
&\leq 1 + k - 1 + \log^* z \\
&= k + \log^* z,
\end{aligned}$$

where the first inequality uses the fact that $\log_z^{(\log_z^* n - k + 1)} n \geq 1$, the third uses the fact that $z \geq 4$ implies $2 \log z \leq z$, and the fourth uses the induction property.

Now we prove the lemma. For $n \leq z$, the lemma is obvious, since $\log^* n \leq \log^* z$. For $n > z$, using our previous claims we can see that

$$\begin{aligned}
\log^* n &= \log^* 2^{\log n} = \log^* 2^{\log z \log_z n} = 1 + \log^*(\log z \log_z n) \\
&\leq 1 + \log^*(z \log_z^{(\log_z^* n - (\log_z^* n - 1))} n) \\
&\leq 1 + \log_z^* n - 1 + \log^* z \\
&= \log_z^* n + \log^* z. \qquad \square
\end{aligned}$$

LEMMA A.2. *For $k, q \geq 1$, $k + (400q^2)^k(k + kq^k) \leq (2400q^3)^k$.*

Proof. For $k, q \geq 1$,

$$k + (400q^2)^k(k + kq^k) \leq k + (400q^2)^k(2kq^k) \leq k + 2k(800q^3)^k \leq 3k(800q^3)^k.$$

Now we must show that that

$$(2400q^3)^k - (3k(800q^3)^k) \geq 0,$$

or more simply that

$$800^k(3^k - 3k) \geq 0.$$

Since $800^k \geq 0$, we must show that $3^k - 3k \geq 0$. Let $f(k) = 3^k - 3k$. Then $f'(k) = 3^k \ln 3 - 3$. Since $f(1) = 0$ and $f'(k) > 0$ for $k \geq 1$, we know that $f(k) \geq 0$ for $k \geq 1$. $\square$

Appendix B. Probabilistic bounds. First we prove a theorem which allows us to bound the sum of discrete dependent random variables by discrete independent random variables. We note that a similar result for continuous random variables can be proved analogously, using the more advanced probabilistic techniques involved in conditional expectations of continuous random variables. For our purposes, however, we will need to use only discrete random variables.

THEOREM B.1. *Take n discrete random variables $X_1, \ldots, X_n$, all with the same range, which might be dependent, and n discrete random variables $Y_1, \ldots, Y_n$, all with the same range as the X_i's, which are independent of each other and the X_i's, such that for any $i \in \{1, \ldots, n\}$, for any real number a, and for any $b_1, \ldots, b_{i-1} \in \text{range}(X_1)$ such that $\Pr(X_1 = b_1, \ldots, X_{i-1} = b_{i-1}) \geq 0$,*

$$\Pr(X_i > a | X_1 = b_1, \ldots, X_{i-1} = b_{i-1}) \leq Pr(Y_i > a).$$

Then for any real number a

$$\Pr(X_1 + \cdots + X_n > a) \leq \Pr(Y_1 + \cdots + Y_n > a).$$

Proof. Let $Q_{b_1,\ldots,b_{n-1}}$ be the event $(X_1 = b_1, \ldots, X_{i-1} = b_{i-1}, Y_{i+1} = b_i, \ldots, Y_n = b_{n-1})$. We can see that

$$\begin{aligned}
&\Pr(X_1 + \cdots + X_i + Y_{i+1} + \cdots + Y_n > a) \\
&= \sum_{\substack{b_1,\ldots,b_{n-1} \in \text{range}(X_1) \\ \Pr(Q_{b_1,\ldots,b_{n-1}})>0}} \Pr(X_i > a - b_1 - \cdots - b_{n-1} | Q_{b_1,\ldots,b_{n-1}}) \Pr(Q_{b_1,\ldots,b_{n-1}}) \\
&\leq \sum_{\substack{b_1,\ldots,b_{n-1} \in \text{range}(X_1) \\ \Pr(Q_{b_1,\ldots,b_{n-1}})>0}} \Pr(Y_i > a - b_1 - \cdots - b_{n-1} | Q_{b_1,\ldots,b_{n-1}}) \Pr(Q_{b_1,\ldots,b_{n-1}}) \\
&= \Pr(X_1 + \cdots + X_{i-1} + Y_i + \cdots + Y_n > a),
\end{aligned}$$

in which the inequality holds because X_i is independent of all Y_j's, and by the condition placed on X_i in the theorem.

Using this fact, we can see that

$$\begin{aligned}
\Pr(X_1 + \cdots + X_n > a) &\leq \Pr(X_1 + \cdots + X_{n-1} + Y_n > a) \\
&\vdots \\
&\leq \Pr(X_1 + Y_2 + \cdots + Y_n > a) \\
&\leq \Pr(Y_1 + \cdots + Y_n > a). \qquad \square
\end{aligned}$$

We use the Chernoff bound to bound the distribution of a random variable Z which is the sum of n independent random variables. For a binomial random variable $Z \sim B(n, p)$, where Z is the sum of n independent Bernoulli trials with probability of success p, Angluin and Valiant [3] show that for $0 < \beta < 1$, one can obtain the bounds

$$\Pr(Z \geq (1 + \beta)np) \leq e^{-\beta^2 np/3},$$

and

$$\Pr(Z \leq (1 - \beta)np) \leq e^{-\beta^2 np/2}.$$

For a random variable Z, which is the sum of n independent random variables with geometric distributions with parameter p, we obtain the bound (where $q = 1 - p$ and $\beta > 0$)

$$\Pr(Z \geq (1 + \beta)nq/p) \leq \begin{cases} e^{-(q\beta)^2 n/2e^{q\beta}} & \text{if } q\beta < 1, \\ e^{-q\beta n/4} & \text{if } q\beta \geq 1. \end{cases}$$

We prove this here. First we prove that for $x \geq 0$,

$$\frac{1 + x}{e^x} \leq e^{-x^2/2e^x}.$$

This can be shown by using the facts that for all real y, $1 + y \leq e^y$ and

$$e^y = \sum_{i \geq 0} \frac{y^i}{i!},$$

as follows.

$$\begin{aligned}\frac{1+x}{e^x} &= \frac{\sum_{i\geq 0}\frac{x^i}{i!} - \sum_{i\geq 2}\frac{x^i}{i!}}{e^x} \\ &= 1 - \frac{\sum_{i\geq 2}\frac{x^i}{i!}}{e^x} \\ &\leq 1 - \frac{\frac{x^2}{2}}{e^x} \\ &\leq e^{-x^2/2e^x}.\end{aligned}$$

Next we prove that for $x \geq 1$,

$$\frac{1+x}{e^x} \leq e^{-x/4}.$$

This follows from the fact that $e^{3x/4} - x - 1 \geq 0$ for $x \geq 1$, which can be derived from the facts that $e^{3/4} - 2 \geq 0$, and that the derivative of $e^{3x/4} - x - 1$ ($\frac{3}{4}e^{3x/4} - 1$) is positive for $x \geq 1$.

Now let $X_1, \ldots, X_n$ be independent random variables with geometric distributions with parameter p, and let $Z = X_1 + \cdots + X_n$. Let $\beta > 0$ and $t > 0$. Then following along the lines of Hagerup and Rüb [30],

$$\begin{aligned}\Pr(Z \geq (1+\beta)nq/p) &= e^{-t(1+\beta)nq/p}e^{t(1+\beta)nq/p}P(e^{tZ} \geq e^{t(1+\beta)nq/p}) \\ &\leq e^{-t(1+\beta)nq/p}E(e^{tZ}).\end{aligned}$$

Then since $X_1, \ldots, X_n$ are independent and identically distributed, we get

$$\begin{aligned}E(e^{tZ}) = E(e^{t(X_1+\cdots+X_n)}) &= E(e^{tX_1}\cdots e^{tX_n}) = \prod_{i=1}^{n} E(e^{tX_i}) \\ &= \left(E(e^{tX_1})\right)^n = \left(\frac{p}{1-qe^t}\right)^n,\end{aligned}$$

where the last equation holds for $t < \ln(1/q)$ [15, p. 269]. Putting $t = \ln((1+\beta)/(1+q\beta))$ yields

$$\begin{aligned}\Pr(Z \geq (1+\beta)nq/p) &\leq \left(\frac{1+q\beta}{1+\beta}\right)^{(1+\beta)nq/p}(1+q\beta)^n \\ &= \left(1 - \frac{p\beta}{1+\beta}\right)^{(1+\beta)nq/p}(1+q\beta)^n \\ &\leq e^{-q\beta n}(1+q\beta)^n,\end{aligned}$$

which can be bounded using the bounds on $\frac{1+x}{e^x}$ derived above. Note that for $0 < q < 1$ and $\beta > 0$, $\ln((1+\beta)/(1+q\beta)) < \ln(1/q)$.

Acknowledgments. Many thanks to Eric Hao, Doug Van Wieren, and the anonymous referees for careful readings of earlier versions of this paper and many helpful comments and suggestions. Also thanks to Leslie Goldberg for the improved (and easier to understand) description of the random-adversary technique that appears in this version.

REFERENCES

[1] M. Ajtai and M. Ben-Or, *A theorem on probabilistic constant depth computations*, in Proc. 16th ACM Symp. on Theory of Computing, Washington, DC, Special Interest Group on Algorithms and Computation Theory (SIGACT), 1984, pp. 471–474.

[2] R. J. Anderson and G. L. Miller, *Optical Communication for Pointer Based Algorithms*, Tech. Report CRI 88-14, University of Southern California, Los Angeles, CA, 1988.

[3] D. Angluin and L. G. Valiant, *Fast probabilistic algorithms for Hamiltonian circuits and matchings*, J. Comput. System Sci., 18 (1979), pp. 155–193.

[4] P. Beame and J. Håstad, *Optimal bounds for decision problems on the CRCW PRAM*, J. Assoc. Comput. Mach., 36 (1989), pp. 643–670.

[5] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, *Highly parallelizable problems*, in Proc. 21st ACM Symp. on Theory of Computing, Seattle, WA, SIGACT, 1989, pp. 309–319.

[6] O. Berkman and U. Vishkin, *Recursive *-tree parallel data-structure*, SIAM J. Comput., 22 (1993), pp. 221–242.

[7] B. Bollobás, T. I. Fenner, and A. M. Frieze, *An algorithm for finding Hamilton paths and cycles in random graphs*, Combinatorica, 7 (1987), pp. 327–341.

[8] R. B. Boppana, *Optimal separations between concurrent-write parallel machines*, in Proc. 21st ACM Symp. on Theory of Computing, Seattle, WA, SIGACT, 1989, pp. 320–326.

[9] D. Breslauer and Z. Galil, *A lower bound for parallel string matching*, SIAM J. Comput., 21 (1992), pp. 856–862.

[10] S. Chaudhuri, *Sensitive functions and approximate problems*, in Proc. 34th IEEE Symp. on Found. of Comput. Sci., Palo Alto, CA, IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 1993, pp. 186–193.

[11] S. Chaudhuri and J. Radhakrishnan, *The complexity of parallel prefix problems on small domains*, in Proc. 33rd IEEE Symp. on Found. of Comput. Sci., Pittsburgh, PA, IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 1992, pp. 638–647.

[12] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik, *New simulations between CRCW PRAMs*, in Proc. 7th Internat. Conf. on Fundamentals of Comput. Theory, Vol. 380, Lecture Notes in Computer Science, 1989, Springer-Verlag, New York, pp. 95–104.

[13] M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk, *Exact lower time bounds for computing Boolean functions on CREW PRAMs*, J. Comput. System Sci., 48 (1994), pp. 231–254.

[14] J. Edmonds, *Lower bounds with smaller domain size on concurrent write parallel machines*, in Proc. Struc. in Complexity Theory, IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 1991, pp. 322–331.

[15] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. 1, John Wiley & Sons, Inc., New York, 1950.

[16] F. E. Fich, F. Meyer auf der Heide, P. Ragde, and A. Wigderson, *One, two, three . . . infinity: Lower bounds for parallel computation*, in Proc. 17th ACM Symp. on Theory of Computing, Providence, RI, SIGACT, 1985, pp. 48–58.

[17] A. M. Frieze, *Parallel algorithms for finding Hamilton cycles in random graphs*, Inform. Process. Lett., 25 (1987), pp. 111–117.

[18] M. Furst, J. B. Saxe, and M. Sipser, *Parity, circuits, and the polynomial-time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13–27.

[19] P. B. Gibbons, Y. Matias, and V. Ramachandran, *The QRQW PRAM: Accounting for contention in parallel algorithms*, in 5th ACM-SIAM Symp. on Discrete Alg., Arlington, VA, ACM SIGACT and SIAM Activity Group on Discrete Mathematics, SIAM, Philadelphia, PA, 1994, pp. 638–648.

[20] J. Gil and Y. Matias, *Fast hashing on a PRAM – designing by expectation*, in 2nd ACM-SIAM Symp. on Discrete Alg., San Francisco, CA, ACM SIGACT and SIAM Activity Group on Discrete Mathematics, SIAM, Philadelphia, PA, 1991, pp. 271–280.

[21] J. Gil, Y. Matias, and U. Vishkin, *Towards a theory of nearly constant time parallel algorithms*, in Proc. 32nd IEEE Symp. on Found. of Comput. Sci., San Juan, PR, IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 1991, pp. 698–710.

[22] J. Gil and L. Rudolph, *Counting and packing in parallel*, in Proc. 15th Internat. Conf. on Parallel Process, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 1000–1002.

[23] L. A. Goldberg, M. Jerrum, and P. D. MacKenzie, *An* $\Omega(\sqrt{\log \log n})$ *lower bound for routing in optical networks*, in Proc. 6th ACM Symp. on Parallel Alg. and Arch., Cape May, NJ, SIGACT, Special Interest Group on Computer Architecture (SIGARCH) in cooper-

ation with the European Association for Theoretical Computer Science (EATCS), 1994, pp. 147–156.

[24] M. T. Goodrich, *Using approximation algorithms to design parallel algorithms that may ignore processor allocation*, in Proc. 32nd IEEE Symp. on Found. of Comput. Sci., 1991, San Juan, PR, IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, pp. 711–722.

[25] V. Grolmusz and P. Ragde, *Incomparability in parallel computation*, Discrete Appl. Math., 29 (1990), pp. 63–78.

[26] Y. Gurevich and S. Shelah, *Expected computation time for Hamiltonian path problem*, SIAM J. Comput., 16 (1987), pp. 486–502.

[27] T. Hagerup, *Fast Parallel Space Allocation, Estimation and Integer Sorting*, Tech. Report MPI-I-91-106, Max-Planck-Institut für Informatik, Saarbrücken, 1991.

[28] T. Hagerup and M. Nowak, *Parallel retrieval of scattered information*, in Proc. 16th Internat. Coll. on Automata, Languages, and Programming, Stresa, Italy, EATCS, 1989, pp. 439–450.

[29] T. Hagerup and R. Raman, *Waste makes haste: Tight bounds for loose parallel sorting*, in Proc. 33rd IEEE Symp. on Found. of Comput. Sci., Pittsburgh, PA, IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 1992, pp. 628–637.

[30] T. Hagerup and C. Rüb, *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33 (1990), pp. 305–308.

[31] J. Håstad, *Almost optimal lower bounds for small depth circuits*, in Proc. 18th ACM Symp. on Theory of Computing, Berkeley, CA, SIGACT, 1986, pp. 6–20.

[32] J. Håstad, *Computational Limitations for Small Depth Circuits*, MIT Press, Cambridge, MA, 1987.

[33] R. Impagliazzo, R. Paturi, and M. E. Saks, *Size-depth trade-offs for threshold circuits*, in Proc. 25th ACM Symp. on Theory of Computing, San Diego, CA, SIGACT, 1993, pp. 541–550.

[34] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, J. van Leeuwen, ed., MIT Press/Elsevier, Cambridge, MA, 1990, pp. 869–941.

[35] P. D. MacKenzie, *A lower bound for the QRQW PRAM*, in Proc. 7th IEEE Symp. on Parallel and Distr. Proc., San Antonio, TX, IEEE Computer Society Technical Committee on Computer Architecture and Technical Committee on Distributed Computing, 1995, pp. 231–237.

[36] P. D. MacKenzie, *Lower bounds for randomized exclusive-write PRAMs*, in Proc. 7th ACM Symp. on Parallel Alg. and Arch., Santa Barbara, CA, ACM SIGACT, ACM SIGARCH in cooperation with EATCS, 1995, pp. 254–263.

[37] P. D. MacKenzie and Q. F. Stout, *Ultra-fast expected time parallel algorithms*, in 2nd ACM-SIAM Symp. on Discrete Alg., San Francisco, CA, ACM SIGACT and SIAM Activity Group on Discrete Mathematics, SIAM, Philadelphia, PA, 1991, pp. 414–423.

[38] P. D. MacKenzie and Q. F. Stout, *Optimal parallel construction of Hamiltonian cycles and spanning trees in random graphs*, in 5th ACM Symp. on Parallel Alg. and Arch., Velen, Germany, ACM SIGACT, ACM SIGARCH in cooperation with EATCS, 1993, pp. 224–229.

[39] Y. Matias and U. Vishkin, *Converting high probability into nearly-constant time – with applications to parallel hashing*, in Proc. 23rd ACM Symp. on Theory of Computing, New Orleans, LA, SIGACT, 1991, pp. 307–316.

[40] A. Thomason, *A simple linear expected time algorithm for the Hamilton cycle problem*, Discrete Math., 75 (1989), pp. 373–379.

[41] A. C.-C. Yao, *Probabilistic computations: Toward a unified measure of complexity*, in Proc. 18th IEEE Symp. on Found. of Comput. Sci., Berkeley, CA, IEEE Computer Society Technical Committee on Mathematical Foundations of Computing, 1977, pp. 222–227.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1581–1611, December 1997

002

RECONFIGURING ARRAYS WITH FAULTS PART I: WORST-CASE FAULTS*

RICHARD J. COLE†, BRUCE M. MAGGS‡, AND RAMESH K. SITARAMAN§

Abstract. In this paper we study the ability of array-based networks to tolerate worst-case faults. We show that an $N \times N$ two-dimensional array can sustain $N^{1-\epsilon}$ worst-case faults, for any fixed $\epsilon > 0$, and still emulate T steps of a fully functioning $N \times N$ array in $O(T + N)$ steps, i.e., with only constant slowdown. Previously, it was known only that an array could tolerate a constant number of faults with constant slowdown. We also show that if faulty nodes are allowed to communicate, but not compute, then an N-node one-dimensional array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate a fault-free array with constant slowdown, and this bound is tight.

Key words. fault tolerance, array-based network, mesh network, network emulation

AMS subject classifications. 68M07, 68M10, 68M15, 68Q68

PII. S0097539793255011

1. Introduction.

In a truly large parallel computer, some components are bound to fail. Knowing this, a programmer can write software that explicitly copes with faults in the computer. But building fault tolerance into every piece of software is cumbersome. The programmer would prefer to program a fault-free virtual computer and leave the job of coping with faults to the hardware. Ideally, the emulation of the fault-free computer should entail little slowdown, even if there are many faults in the actual hardware.

The emulation of the fault-free computer consists of two tasks. The faulty computer must emulate the computations performed by the processors of the fault-free computer, and it must emulate the communications between those processors. Emulating the computations does not incur much slowdown. The computation performed by each faulty processor is simply mapped to a fault-free processor. But once the computations are moved around, processors that are neighbors in the fault-free computer may no longer be neighbors in the faulty computer. Thus, there is a risk that the communications will be slowed down. The solution to this problem depends on the communication topology of the computer.

One of the most popular ways to construct a parallel computer is to arrange the processors as a two-dimensional or three-dimensional array. Commercial machines including the Cray T3D [10] and MasPar MP-1 [3] have this topology, as do experimental machines such as iWarp [4] and the J-Machine [18]. In this paper we study the ability of machines like these to tolerate faults. We show, for example, that an

* Received by the editors September 7, 1993; accepted for publication (in revised form) October 20, 1995. This research was conducted while the first author was visiting NEC Research Institute, the second author was employed at NEC Research Institute, and the third author was a student at Princeton University.

http://www.siam.org/journals/sicomp/26-6/25501.html

† Courant Institute, New York University, New York, NY 10012 (cole@cs.nyu.edu). This author's research was supported in part by NSF grants CCR-92-02900 and CCR-95-03309.

‡ School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (bmm@cs.cmu.edu). This author's research was supported in part by an NSF National Young Investigator Award, CCR-94-57766, with matching funds provided by NEC Research Institute, and by ARPA contract F33615-93-1-1330.

§ Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (ramesh@cs.umass.edu). This author's research was supported in part by NSF grant CCR-94-10077.

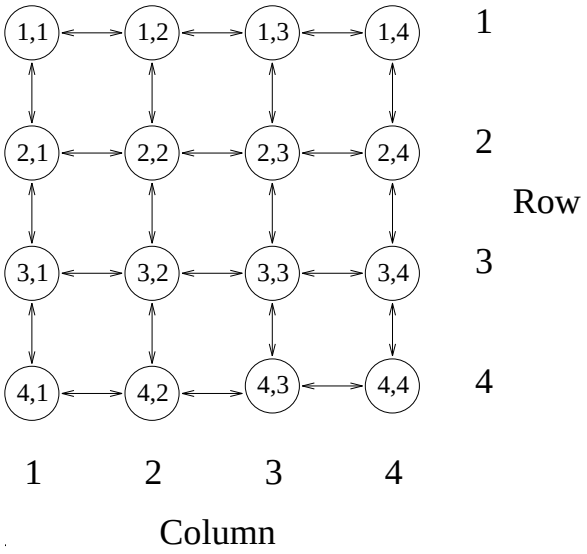


FIG. 1. *A* 4×4 *mesh.*

$N \times N$ two-dimensional array can sustain $N^{1-\epsilon}$ worst-case faults, for any fixed $\epsilon > 0$, and still emulate a fault-free $N \times N$ array with constant slowdown.

1.1. Arrays and the fault model. A d-dimensional *array* with side length N consists of N^d nodes, each labeled with a distinct d-tuple $(r_1, r_2, \ldots, r_d)$, where $1 \leq r_i \leq N$ for $1 \leq i \leq d$. Two nodes are connected by a pair of oppositely directed edges if their labels differ by 1 in precisely one coordinate. For example, in a four-dimensional array with side length 8, nodes $(3, 2, 4, 8)$ and $(3, 2, 3, 8)$ are neighbors, but $(3, 2, 4, 8)$ and $(3, 2, 3, 7)$ are not. A two-dimensional array is also called a *mesh*. A 4×4 mesh is shown in Figure 1. For each i, the mesh nodes labeled (i, j), where $1 \leq j \leq N$, are said to belong to the ith row. For each j, the mesh nodes labeled (i, j), where $1 \leq i \leq N$, are said to belong to the jth column. Sometimes two nodes are considered to be neighbors if they differ in precisely one coordinate and their values in that coordinate are 1 and N. In this case we say that the array has *wraparound* edges. A two-dimensional array with wraparound edges is also called a *torus*. All of the results in this paper hold whether or not the array has wraparound edges.

The nodes in an array represent processors and the edges represent communication links. We assume that the array operates in a synchronous fashion. At each time step, each node can receive a message from each of its neighbors, perform a simple local computation, and then send a message to each of its neighbors.

In this paper we assume that only nodes fail, that these failures are static, and that their locations are known. We also assume that the faults appear in a worst-case pattern, i.e., that an adversary decides where to put the faults in the network. We allow information about the locations of the faults to be used in reconfiguring the network. We assume that a faulty node can neither compute nor communicate. All of our results can be extended to handle edge failures by viewing an edge failure as the failure of one of the nodes incident on the edge. In section 4, we use a weaker fault model for one-dimensional arrays by allowing faulty nodes to communicate but not compute. We observe that even in this weaker fault model linear arrays cannot tolerate as many worst-case faults as two-dimensional arrays. In another paper, we

consider random fault patterns. In the case of random faults, we assume that each node fails independently with some fixed probability p.

1.2. Embeddings. The simplest way to show that a network with faults, H, can emulate a fault-free network, G, is to find an embedding of G into H. We call H the *host* network and G the *guest* network. An embedding maps nodes of G to nonfaulty nodes of H, and edges of G to nonfaulty paths in H. The three important measures of an embedding are its load, congestion, and dilation. The *load* of an embedding is the maximum number of nodes of G that are mapped to any node of H. The *congestion* of an embedding is the maximum number of paths that use any edge of H. The *dilation* is the maximum length of any path. Given an embedding of G into H, H can emulate each step of the computation of G by routing a packet for each edge of G along the corresponding path in H. Leighton, Maggs, and Rao [11] showed that if the embedding has load l, congestion c, and dilation d, then the packets can be routed so that the slowdown of the emulation is $O(l + c + d)$.

In order for an embedding-based emulation scheme to have constant slowdown, the load, congestion, and dilation of the embedding must all be constant. Unfortunately, by placing $f(N)$ faults in an N-node two- or three-dimensional array H, where $f(N)$ is any function that is $\omega(1)$, it is possible to force either the load, congestion, or dilation of every embedding of an array G of the same size and dimension to be larger than a constant [7, 8, 12]. Similarly, if $\Theta(N)$ faults are placed in H at random, then with high probability every embedding of G in H will have $\omega(1)$ load, congestion, or dilation. Thus, in order to tolerate more than a constant number of worst-case faults or constant-probability failures, a more sophisticated emulation technique is required.

1.3. Redundant computation. All of the emulations in this paper use a technique called *redundant computation*. The basic idea is to allow H to emulate each node of G in more than one place. This extra freedom makes it possible to tolerate more faults, but it adds the complication of ensuring that different emulations of the same node of G remain consistent over time. The technique of redundant computation was previously used to tolerate faults in hypercubic networks [13], and to construct work-preserving emulations in fault-free networks [6, 9, 15, 16, 17, 21].

1.4. Previous work. A large number of researchers have studied the ability of arrays and other networks to tolerate faults. The most relevant papers are described below.

Raghavan [20] devised a randomized algorithm for solving one-to-one routing problems on $N \times N$ meshes. He showed that even if each node fails with some fixed probability $p \leq .29$, then for almost all random fault patterns, any packet that can reach its destination does so in $O(N \log N)$ steps, with high probability. Mathies [14] improved the $p \leq .29$ bound to $p \approx .4$.

Kaklamanis et al. [8] improved Raghavan's result by devising a deterministic routing algorithm. For almost all random fault patterns, the algorithm guarantees that any packet that can reach its destination does so within $O(N)$ steps. This algorithm can also tolerate worst-case faults. If there are k faults in the network, it runs in time $O(N + k^2)$. Kaklamanis et al. also showed that an $N \times N$ mesh with constant-probability failures or $\Theta(N)$ worst-case faults can sort or route N^2 items, or multiply two $N \times N$ matrices in $O(N)$ time. They also showed that, with high probability, an $N \times N$ mesh with constant-probability failures can emulate a fault-free $N\sqrt{\log N} \times N\sqrt{\log N}$ mesh with $O(\log N)$ slowdown. (Throughout this paper the base of the function log is 2.)

Aumann and Ben-Or [2] used Rabin's information dispersal technique [19] to show that an $N \times N$ mesh H with slack s, $s = \Omega(\log N \log\log N)$, can emulate a fault-free $N \times N$ mesh G with slack s with constant slowdown, even if every node or edge in H fails with some fixed probability $p > 0$ at some point *during* the emulation. (In a slack s computation, each node v in G emulates s virtual nodes. In each *superstep*, v emulates one step of each virtual node, and each virtual node can transmit a message to one of v's four neighbors.) Aumann and Ben-Or assumed that in a single step, an edge in H can transmit a message that is $\log N$ times as large as the largest message that can be transmitted in a single step by G.

Bruck, Cypher, and Ho [5] showed that by adding some spare nodes and edges to a mesh, it is possible for the mesh to sustain many faults and still contain a working fault-free $N \times N$ mesh as a subgraph. In particular, they showed that by adding $O(k^3)$ spare nodes, it is possible to tolerate k worst-case faults, and by adding k spare nodes, for $k = O(N^{2/3})$, it is possible to tolerate k random faults, with high probability. In both cases, the networks have bounded degree. Tamaki [24] showed how to construct an $O(N)$-node network with degree $O(\log\log N)$ with the property that, for any $d \geq 2$, even if every node fails with constant probability, with high probability the network contains a fault-free N-node d-dimensional array as a subgraph. He also showed how to construct a bounded-degree network with the property that even if $N^{(1-2^{-d})/d}$ worst-case faults are placed in the network, the network is guaranteed to contain an N-node d-dimensional array as a subgraph. Ajtai et al. [1] analyzed the technique of adding spare nodes to larger classes of networks that include meshes. In all of these constructions the very large scale integration (VLSI) layout area requirements of the networks with spare nodes and edges are much larger than those of the arrays that they contain as subgraphs.

Leighton, Maggs, and Sitaraman [13] showed that an N-node butterfly can tolerate $N^{1-\epsilon}$ worst-case faults, for any fixed $\epsilon > 0$, and still emulate a fault-free N-node butterfly with constant slowdown. They proved the same result for the shuffle-exchange network. They also showed that, for any constant $k > 0$, an N-node mesh of trees can tolerate $\log^k N$ worst-case faults and still emulate a fault-free mesh of trees with constant slowdown. Finally, they showed that, with high probability, an N-node butterfly (or shuffle-exchange network) can tolerate constant-probability failures with slowdown $2^{O(\log^* N)}$. Tamaki independently showed that, with high probability, an N-node butterfly can be embedded in an N-node butterfly containing constant-probability node failures with load $O(1)$, congestion $O((\log\log N)^{8.2})$, and dilation $O((\log\log N)^{2.6})$ [22]. In [23], he proved a similar result for a class of networks called cube-connected arrays.

1.5. Our results. In section 2 we show that an $N \times N$ array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate T steps of a fault-free array in $O(T + N)$ steps, i.e., with constant slowdown. Previously it was only known that a constant number of worst-case faults could be tolerated with constant slowdown. Section 2 introduces most of the terminology that is used throughout this paper.

In section 3 we present a method called *multiscale emulation* for tolerating $N^{1-\epsilon}$ worst-case faults on an $N \times N$ mesh with constant slowdown, for any fixed $\epsilon > 0$. This result nearly matches the $O(N)$ upper bound on the number of worst-case faults that can be tolerated with constant slowdown.

In section 4 we show that if faulty nodes are allowed to communicate but not compute, then an N-node one-dimensional array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate a fault-free N-node linear array only with

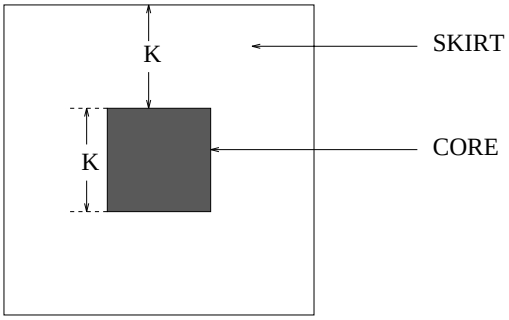


FIG. 2. *A finished box.*

constant slowdown. We also show that an N-node linear array cannot tolerate more than $\log^k N$ worst-case faults without suffering more than constant slowdown, provided that the emulation is static. In a *static* emulation, each host node a emulates a fixed set $\psi(a)$ of guest nodes. Redundant computation is allowed; a guest node u may belong to $\psi(a)$ and $\psi(b)$ for distinct host nodes a and b. In this case we say that there are multiple *instances* of the guest node u. For each guest time step, host node a emulates the computation performed by each node u in $\psi(a)$. Furthermore, for every guest edge $e = (v, u)$ into u, for each instance u' of u in the host, there is a corresponding instance v' of v at some host node such that for each guest time step *the same* instance v' sends a packet for the edge e to u'. (Note that v' may also send packets to other instances of u.) The emulations that use redundant computation in this paper and in [6, 9, 13, 21] are all static.

2. A simple method for tolerating worst-case faults on the mesh. In this section, we show that, for any constant $k > 0$, an $N \times N$ mesh with $\log^k N$ worst-case faults can emulate any computation of an $N \times N$ fault-free mesh with only constant slowdown. The procedure for reconfiguring the computation around faults consists of two steps. The first is a process by which the faults are enclosed within square regions of the mesh called *boxes*. We call this step the *growth process*. We describe this process in section 2.1. The next is an emulation technique that maps the computation of the fault-free mesh (the guest) to nodes in the faulty mesh (the host). The boxes grown in the first step determine how the mapping of the computation is done. This process is described in section 2.2. For simplicity, we assume that the mesh has wraparound edges. This assumption can be easily done away with at the cost of considering some special cases for faults near the border of the mesh.

2.1. The growth process. The growth process grows boxes on the faulty mesh, i.e., the host. There are two types of boxes. The first type is called a *core*. A core has too many faults in it to perform any role in the emulation. The second type is a *finished box*. A finished box can emulate a submesh of the same side length with constant slowdown. (The side length of a box or submesh is the number of nodes on each side, i.e., a box or submesh with side length k has k^2 nodes.) A finished box of side length $3k$ consists of a core of side length k surrounded by a *skirt*, which contains no faults, of width k as shown in Figure 2. (Like side length, width is measured in nodes.)

At every stage of the growth process, the algorithm maintains a set of boxes, some of which are cores while others are finished boxes. At the beginning of the growth

process, every fault is enclosed in a box with unit side length that is a core. In every stage of the growth process, we pick a core—say, of side length k—and grow a skirt of width k around it. If the core or the skirt intersects some other core, we merge the two cores to form a new core whose boundary is the smallest square that contains both cores. If the core or the skirt intersects a finished box, we find the smallest square box that contains both the core and the core of the finished box and turn this bounding box into a core. We also remove the finished box from the list of finished boxes. If the skirt does not intersect any other boxes, the newly created box is labeled a finished box. We continue applying these rules until either some core grows to be too large to grow a skirt around it or no core remains and no two finished boxes intersect. For the former outcome, some core must have side length greater than $N/3$. (Recall that the mesh has wraparound edges.) In Lemma 2.1 we show that this cannot happen if there are fewer than $(\log N)/2$ faults.

Our rules for growing cores assign each fault to a unique core. Initially, every fault is assigned to the unit-sized core enclosing it. Inductively, when two or more cores are merged to form a new core, every fault assigned to the old cores is now assigned to the new core. A core is said to *contain* all the faults assigned to it. Note that if two cores overlap, it is possible for a fault to be geometrically located inside of a core and yet not be contained by that core.

LEMMA 2.1. *If the number of faults is less than* $(\log N)/2$*, then the growth process terminates with nonoverlapping finished boxes.*

Proof. Let $F(k)$ denote the minimum number of faults that a core of side length k must contain. We show by induction that $F(k) \geq (\log k)/2 + 1$. As the base case, $F(1) = 1$, which satisfies the hypothesis. Assume that we have a core of side length $k > 1$. This core must have been created by merging two cores according to one of the two merging rules stated previously. Let x and y denote the side lengths of these two cores. In both cases, $x + y \geq \lfloor k/2 \rfloor + 1$. Using the inductive hypothesis, we have

$$\begin{aligned} F(k) &\geq F(x) + F(y) \\ &\geq (\log x)/2 + 1 + (\log y)/2 + 1. \end{aligned}$$

The values of x and y that minimize the right-hand side of this inequality are $x = \lfloor k/2 \rfloor$ and $y = 1$. Substituting these values, we have

$$\begin{aligned} F(k) &\geq (\log \lfloor k/2 \rfloor)/2 + 2 \\ &\geq (\log k)/2 + 1. \end{aligned} \tag{1}$$

This proves our inductive hypothesis.

Now suppose that there is a core of side length greater than $N/3$. Then it must contain at least $F(\lfloor N/3 \rfloor + 1)$ faults, which is more than $(\log N)/2$, which is a contradiction. Therefore, the growth process must terminate with a set of nonintersecting finished boxes. □

2.2. The emulation. In this section, we show that if the growth process terminates with a set of nonintersecting finished boxes, then the host H can emulate the guest G with constant slowdown.

The emulation of G by H is described as a pebbling process. There are two kinds of pebbles. With every node v of G and every time step t, we associate a state pebble (s-pebble), $\langle v, t \rangle$, which contains the entire state of the computation performed at node v at time t. With each directed edge e in G and every time step t, we associate

a communication pebble (c-pebble), $[e,t]$, which contains the message transmitted along edge e at time step t.

The host H will emulate each step t of G by creating at least one s-pebble $\langle v,t\rangle$ for each node v of G and a c-pebble $[e,t]$ for each edge e of G. A node of H can create an s-pebble $\langle v,t\rangle$ only if it contains s-pebble $\langle v,t-1\rangle$ and all of the c-pebbles $[e,t-1]$, where e is an edge into v. The creation of an s-pebble takes unit time. A node of H can create a c-pebble $[g,t]$ for an edge g out of v only if it contains an s-pebble $\langle v,t\rangle$. The creation of a c-pebble also takes unit time. Finally, a node of H can transmit a c-pebble to a neighboring node in H in unit time. A node of H is not permitted to transmit an s-pebble since an s-pebble may contain a lot of information. All of our emulations are static, i.e., each node of H emulates a fixed set of nodes of G, and for each guest edge $e=(v,u)$ and each instance of u, there is a corresponding instance of v such that for each guest time step, the host node creating s-pebbles for that instance of v sends a c-pebble $[e,t]$ to the host node creating s-pebbles for u. Initially, each node of H contains an s-pebble $\langle v,0\rangle$ for each node v of G that is mapped to it.

Using the growth process of the previous section, we grow a collection of nonintersecting finished boxes on the faulty mesh H. If the faulty mesh H has fewer than $(\log N)/2$ faults, then the growth process will terminate with a set of nonintersecting finished boxes. Every node of H that does not belong to any of the finished boxes will emulate the computation of the corresponding node of G. Every finished box of H will be responsible for emulating the corresponding submesh of G. However, since some of the nodes inside the core of a finished box are faulty, we must make sure that no computation is mapped to them. In fact, there will be no computation mapped to any node inside a core. All the computations will be mapped to the skirt of the finished box, which is completely fault free. Since we would like each finished box to do its share of the emulation with constant slowdown, we need to avoid long communication delays caused by the fact that the core is unusable. As we shall see, we can hide the latency involved in sending c-pebbles long distances across the mesh using a technique called *redundant computation*. In an emulation that performs redundant computation, some nodes of G are emulated by more than one node in H.

The computation of G corresponding to a finished box is mapped with replication to the skirt of that finished box as follows (see Figure 3). Suppose that the core has side length k and the finished box has side length $3k$. We begin by dividing the submesh of G corresponding to a finished box into two regions, the *patch* and the *outerskirt*. The patch is a square region of side length $2k$ whose center is also the center of the finished box. The outerskirt consists of the entire submesh of G with a square of side length k removed from its center. As shown in Figure 3, the patch and the host overlap in an annular region of width k. The patch and the outerskirt are mapped to the finished box as follows. The outerskirt is the same size and shape as the skirt of the finished box; every node in the outerskirt is mapped to its corresponding node in the skirt. The patch, which is a square of side length $2k$, is mapped to a square of side length $k/2$ called the *patch region* shown in Figure 3; the patch region is contained within the skirt of the finished box. This is done in the simplest manner by mapping squares of side length 4 of the patch to one node of the square in the finished box.

We now observe some properties of the mapping. A *ring* is a set of nodes that form the four sides of a square. The nodes in the finished box to which the border of the patch and the inner border of the outerskirt are mapped form rings in the finished box. We call these rings the *border rings*, or b-rings for short. The nodes on the

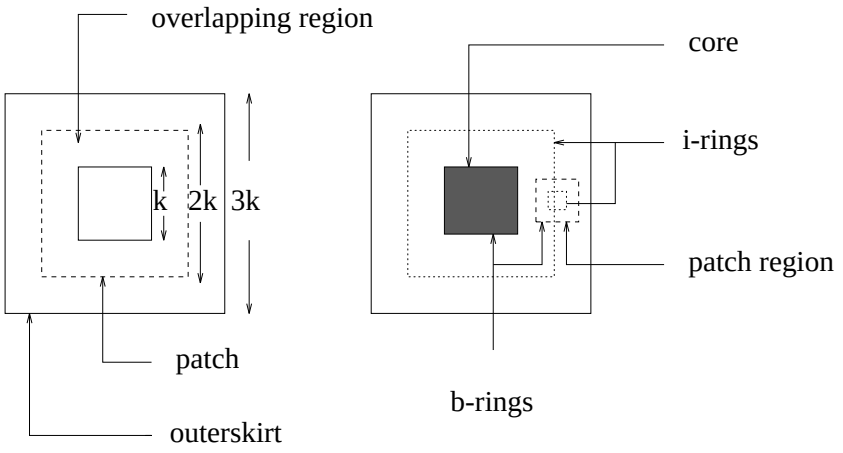


FIG. 3. *Mapping the computation inside a finished box. A finished box in H is shown on the right and the corresponding submesh in G is shown on the left.*

border of the patch have duplicates in the interior of the outerskirt. Similarly, the nodes on the inner border of the outerskirt have duplicates in the interior of the patch. These duplicate nodes in the interior of the patch and the interior of the outerskirt are also mapped to rings in the finished box. We call these rings the *interior rings*, or i-rings for short.

In order for a node m in H to create an s-pebble for a node v in G, it must receive c-pebbles for each of the edges into v in G. If v is in the interior of the patch or the outerskirt, then the s-pebbles for the neighbors of v are created either by m or by the neighbors of m. In this case the required c-pebbles can be obtained in constant time. The s-pebbles for the neighbors of a node v on the border of the patch or outerskirt, however, may not be created near m in H. In our emulation, every node v on the border receives the c-pebbles for *all* of its incoming edges from its duplicate v' that is mapped to a node m' on one of the i-rings in the finished box. These c-pebbles are first created by the four neighbors of v', then sent to m' (in constant time), then forwarded on to m along a path that we call a *communication path*. Thus, for each node m' on an i-ring, we will need to route a communication path to its duplicate m on a b-ring. Note that these paths are determined off-line, before the start of the emulation. The skirt, which is fault free, can be used as a crossbar to route the paths with dilation $O(k)$ and constant congestion. For the sake of brevity, the details are omitted.

We now describe the actual emulation. Each node m of H executes the following algorithm which proceeds as a sequence of macrosteps. Each macrostep consists of the following three substeps.

(1) Computation step: For each node v of G that has been assigned to m, m creates a new s-pebble $\langle v, t\rangle$, provided that m has already created $\langle v, t-1\rangle$ and has received c-pebbles $[e, t-1]$ for every edge e into v.
(2) Communication step: For every node v such that s-pebble $\langle v, t\rangle$ was created by m in the computation step of the current macrostep, and for every edge e out of v, node m creates c-pebble $[e, t]$. If $[e, t]$ is needed by a neighbor m' of m, then m sends $[e, t]$ to m'.
(3) Routing step: If m lies on an i-ring, then m makes copies of any c-pebbles

that were sent to m during the communication step of the current macrostep. Then (whether or not m lies on an i-ring), for every c-pebble $[e, t]$ at m that has not yet reached its destination, m forwards it one step along its communication path.

LEMMA 2.2. *A macrostep takes a constant number of time steps.*

Proof. At each node of H, there are at most 17 s-pebbles to be updated in the computation step and hence this step takes constant time. Each s-pebble update can cause at most 4 c-pebbles (the outdegree of the node in G) to be sent. Thus the communication step takes only constant time. If the s-pebble is on the i-ring, it must send four additional c-pebbles to its duplicate on the b-ring. Since the paths used for routing have constant congestion and since a c-pebble in transit to its destination moves in every macrostep, there are at most a constant number of c-pebbles resident in a node at any time step that have not yet reached their destinations. Therefore, the routing step also takes constant time. □

THEOREM 2.3. *Any computation on a fault-free mesh G that takes time T can be emulated by the faulty mesh H with less than $(\log N)/2$ worst-case faults in $O(T+N)$ time steps.*

Proof. From Lemma 2.1, we know that since the number of worst-case faults in H is less than $(\log N)/2$, the growth process terminates with a set of nonintersecting finished boxes. The computation of G is mapped inside each of these finished boxes as described earlier in this section, and each node m of H performs the emulation algorithm. We will show that only $O(T + N)$ macrosteps are required to emulate a T-step computation of G. The theorem will then follow from Lemma 2.2.

The *dependency tree* of an s-pebble represents the functional dependency of this s-pebble on other s-pebbles and can be defined recursively as follows. As the base case, if $t = 0$, the dependency tree of $\langle v, t\rangle$ is a single node, $\langle v, 0\rangle$. If $t > 0$, the creation of s-pebble $\langle v, t\rangle$ requires an s-pebble $\langle v, t-1\rangle$ and all c-pebbles $[e, t-1]$ such that e is an incoming edge of node v in G. These c-pebbles are sent by some other s-pebbles $\langle u, t-1\rangle$, where u is a neighbor of v in G. The dependency tree of $\langle v, t\rangle$ is defined recursively as follows. The root of the tree is $\langle v, t\rangle$. The subtrees of this tree are the dependency trees of $\langle v, t-1\rangle$ and $\langle u, t-1\rangle$, for all s-pebbles $\langle u, t-1\rangle$ that send c-pebbles to $\langle v, t\rangle$.

We now look at the dependency tree of the s-pebble that was created last. Let the emulation of T steps of G take T' time (in macrosteps) on H. Let $\langle v, T\rangle$ be an s-pebble that was updated in the last macrostep. For every tree node s, we can associate a time (in macrosteps) $\tau(s)$ when that s-pebble was created. We choose a *critical path*, $s_T, s_{T-1}, \ldots, s_0$, of tree nodes from the root to the leaves of the tree as follows. Let $s_T = \langle v, T\rangle$ be the root of the tree. s_T requires the s-pebble $\langle v, T-1\rangle$ and c-pebbles $[e, T-1]$. Let ϕ be the function that maps an s-pebble, $\langle v, t\rangle$ to the node in H that contains it. If the s-pebble $\langle v, T-1\rangle$ was created after all the c-pebbles were received then choose s_{T-1} to be $\langle v, T-1\rangle$. Otherwise, choose the s-pebble that sent the c-pebble that arrived last at node $\phi(\langle v, T\rangle)$ to be s_{T-1}. After choosing s_{T-1}, we choose the rest of the sequence recursively in the subtree with s_{T-1} as the root. We define a quantity l_i as follows. If $\phi(s_i)$ and $\phi(s_{i-1})$ are the same node or neighbors in H, then $l_i = 1$. Otherwise, l_i is the length of the path by which a c-pebble generated by s_{i-1} is sent to s_i. From the definition of our critical path, $\tau(s_i) - \tau(s_{i-1})$ equals l_i. This is because a c-pebble moves once along its communication path in every

macrostep. Therefore

$$T' = \sum_{0<i\leq T} (\tau(s_i) - \tau(s_{i-1})) = \sum_{0<i\leq T} l_i.$$

Now suppose that some l_i is greater than 1. Then $\phi(s_i)$ must lie on the b-ring of a finished box, and some c-pebble must have taken a communication path of length l_i from a node m on an i-ring of the same finished box. In this case either $\phi(s_{i-1}) = m$ or $\phi(s_{i-1})$ is a neighbor of m. The key observation is that since $\phi(s_{i-1})$ is either on or next to an i-ring, in going down the critical path from s_{i-1} to s_0 we can encounter no more communication paths until we reach an s-pebble embedded in the b-ring; i.e., the values of $l_{i-1}, l_{i-2}, \ldots, l_{\max\{i-q,1\}}$ are all equal to 1 for some $q = \Theta(l_i)$. As $l_i = O(N)$, for all i, $T' = \sum_i l_i = O(T + N)$. □

It is possible to apply the construction described in this section recursively to show that, for any constant $k > 0$, an $N \times N$ mesh can sustain $\log^k N$ worst-case faults and still emulate a fault-free mesh with slowdown. Because a stronger result is proven in section 3, the proof is omitted.

THEOREM 2.4. *For any constant $k > 0$, an $N \times N$ mesh with $\log^k N$ worst-case faults can emulate T steps of the computation of a fault-free $N \times N$ mesh in $O(T+N)$ steps with constant slowdown.*

The construction described in this section assumes that large buffers are available at each node in the host to hold c-pebbles that reach their destinations early. Early arrivals can be prevented by slowing down the computation of some nodes and by finding communication paths with the property that all paths within a given finished box have the same length.

3. Multiscale emulation. In this section, we show that an $N \times N$ mesh, H, with any set of $N^{1-\epsilon}$ faults, for any fixed $\epsilon > 0$, can emulate any computation of a fault-free $N \times N$ mesh, G, with constant slowdown.

The major difference between the emulation scheme in this section and that in section 2 is that in this section we allow a finished box to contain smaller finished boxes. In emulating the region of the guest mesh assigned to it, a finished box will in turn assign portions of this computation to each of the smaller boxes that it contains. These smaller boxes might in turn contain even smaller boxes and hence the term *multiscale emulation*.

For simplicity, we assume that the mesh has wraparound edges. This assumption can be easily done away with at the cost of considering some special cases for faults near the border of the mesh.

3.1. The growth process. In this section, we show how to grow boxes on the faulty mesh H. There are two types of boxes: cores and finished boxes. A *core* is not capable of performing any portion of the emulation. A *finished box* consists of a core surrounded by a skirt. An (α-β)-*ensemble* is a collection of possibly intersecting finished boxes. Every finished box B in an (α-β)-ensemble has a distinct round number. The *intersecting region* of a finished box B in the ensemble is defined to be the region formed by nodes that lie both in the skirt of B and in some other finished box with a smaller round number than B. The boxes in an (α-β)-ensemble satisfy the following properties.

(1) Every fault in the mesh H is contained in and assigned to the core of some finished box in the ensemble.
(2) If the core of a finished box has side length k, then the width of the skirt of the finished box is $\lfloor \alpha k \rfloor$.

(3) The sum of the side lengths of the finished boxes in the intersecting region of every finished box B is at most β times the width of the skirt of B.

The growth process produces an (α-β)-ensemble of boxes, where $0 < \alpha, \beta < 1$. It proceeds in rounds until there are no more cores left. Initially, every fault is enclosed in a core with side length $\lceil 1/\alpha \rceil$. Each round produces either a new core or a new finished box. Each new finished box is numbered with the round in which it was created. At the beginning of each round, a core of the smallest side length (say, k) is selected and a skirt of width $\lfloor \alpha k \rfloor$ is grown around it to form a box (call this box B). If $k + 2\lfloor \alpha k \rfloor > N$, then the side length of B will have to be larger than the size of the mesh itself and this is not possible. If this condition arises the growth process halts and is said to have failed. If this condition does not arise then one of the following steps is executed after which the growth process proceeds to the next round.

Expand Step. If the sum of the side lengths of the finished boxes in the intersecting region of B is more than β times the width of the skirt of B (i.e., $\beta\lfloor \alpha k \rfloor$), then we find the smallest bounding box that contains the core of B as well as the cores of all the finished boxes that intersect the skirt or core of B and turn this box into a new core. The finished boxes whose cores were included in this new core cease to exist, and their faults are assigned to the new core.

Create Step. Otherwise, if the sum of the side lengths of the finished boxes in the intersecting region of B is at most $\beta\lfloor \alpha k \rfloor$, then we declare box B to be a finished box.

Note that in the expand step the intersecting region of B is computed using the collection of finished boxes that exist during that round.

LEMMA 3.1. *The growth process produces an (α-β)-ensemble of finished boxes, provided that it does not fail.*

Proof. We must show that all three of the properties of an (α-β)-ensemble are satisfied when the growth process does not fail. The growth process must terminate since at each round either the expand step increases the side length of a core without changing the number of cores, or the create step decreases the total number of cores by one. Property 1 is satisfied initially and since the expand step forms a new core by enclosing a group of old cores, by induction this property will hold after every round. Property 2 is satisfied by construction. Finally, when a finished box B is created in the create step, the sum of the side lengths of the finished boxes in the intersecting region of B is at most β times the width of the skirt of B. New finished boxes created in later rounds do not affect this intersecting region since they all have greater round numbers than B. Some of the finished boxes with round numbers less than B may cease to exist due to the application of the expand step in some later rounds. However, this can only decrease the sum of the side lengths of the finished boxes in the intersecting region of B. Thus, Property 3 will be true for all of the finished boxes when the growth process terminates. □

THEOREM 3.2. *For any fixed constants β and ϵ, where $0 < \beta < 1$ and $0 < \epsilon < 1$, there is a constant α, where $0 < \alpha < 1$, such that for sufficiently large N, for any set of $N^{1-\epsilon}$ faults in H the growth process grows an (α-β)-ensemble of finished boxes.*

Proof. We must show that for any fixed ϵ and β there is a constant α such that the growth process never fails, i.e., no core of side length more than $N/(2\alpha + 1)$ is ever created. Then, by using Lemma 3.1, we can infer the theorem.

The key idea is to prove a lower bound, $F(k)$, on the number of faults that any core of side length k must contain. Let $\delta = \epsilon/2$. We show by induction on k that $F(k) \geq Ak^{1-\delta}$, for some constant A. In order to satisfy the basis of the induction, we will choose A to be small enough that $F(k) \geq Ak^{1-\delta}$ for small values of k. Inductively,

suppose that a new core of side length k is formed in some round. Let x be the side length of the core selected in this round and let $y_1, y_2, \ldots, y_m$ be the side lengths of the other cores that were enclosed to form the new core. Since the new core contains these cores, we have $k > x$ and $k > y_i$, for $1 \le i \le m$. Since the number of faults in the new core is at least as large as the number of faults in the cores used to form it, using the inductive hypothesis we have

$$\begin{aligned} F(k) &\ge F(x) + F(y_1) + \cdots + F(y_m) \\ &\ge Ax^{1-\delta} + Ay_1^{1-\delta} + \cdots + Ay_m^{1-\delta}. \end{aligned} \tag{2}$$

Let y_1 and y_2 be the side lengths of the two largest cores. If a new core was formed, then it must have been formed in the expand step. Thus, the side length k of the new core is at most $(y_1 + y_2)(1 + \alpha) + x(1 + 2\alpha)$. Thus, to prove the inductive hypothesis, it suffices to show that

$$x^{1-\delta} + \sum_{i=1}^{m} y_i^{1-\delta} \ge [(y_1 + y_2)(1 + \alpha) + x(1 + 2\alpha)]^{1-\delta}. \tag{3}$$

Since the cores with side lengths y_i belong to finished boxes created in earlier rounds, $y_i \le x$, for all i. Furthermore, since a new core was created, $\sum_{i=1}^{m} (y_i + 2\lfloor \alpha y_i \rfloor) \ge \beta \lfloor \alpha x \rfloor$, which implies that $\sum_{i=1}^{m} y_i \ge \beta\alpha x/6$ for $0 < \alpha < 1$. Since $\beta < 1$, there must be a largest index $j \ge 2$ such that $\beta\alpha x/6 \le \sum_{i=1}^{j} y_i \le 2x$. Let $y = \sum_{i=1}^{j} y_i$. Then $y \ge y_1 + y_2$. Also, because of the convexity of the function $f(z) = z^{1-\delta}$, $y^{1-\delta} \le \sum_{i=1}^{j} y_i^{1-\delta} \le \sum_{i=1}^{m} y_i^{1-\delta}$. Thus, inequality (3) must hold if

$$[y(1 + \alpha) + x(1 + 2\alpha)]^{1-\delta} \le x^{1-\delta} + y^{1-\delta} \tag{4}$$

holds for all y such that $\beta\alpha x/6 \le y \le 2x$.

Proving that inequality (4) holds for sufficiently small α requires some elementary (but painstaking) calculations. Let $\lambda = y/x$. In terms of λ, we need to show that for sufficiently small α, the inequality

$$[(1 + \alpha)\lambda + (1 + 2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} \le 0 \tag{5}$$

holds for all λ such that $\beta\alpha/6 \le \lambda \le 2$. (As we shall see, $\alpha \le \min\{(\beta/24)^{1/\delta}, \delta/16\}$ suffices.) There are three cases to consider.

First, suppose that $\lambda \le \alpha$. In this case, we have

$$[(1 + \alpha)\lambda + (1 + 2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} \le (1 + 4\alpha)^{1-\delta} - 1 - (\beta\alpha/6)^{1-\delta} \tag{6}$$

$$\le 4\alpha - \left(\frac{(\beta/6)^{1-\delta}}{\alpha^{\delta}} \right) \alpha \tag{7}$$

$$\le \left(4 - \frac{\beta}{6\alpha^{\delta}} \right) \alpha. \tag{8}$$

Equation (6) is derived using the inequalities $(1 + \alpha) \le 2$, $\lambda \le \alpha$, and $\beta\alpha/6 \le \lambda$. Equation (7) is derived from (6) using the inequality $(1 + 4\alpha)^{1-\delta} \le (1 + 4\alpha)$. Equation (8) is derived from (7) using the inequality $\beta/6 \le (\beta/6)^{1-\delta}$. For $\alpha \le (\beta/24)^{1/\delta}$, the right-hand side of (8) is at most 0.

Second, suppose that $\alpha \le \lambda \le 1$. In this case we have

$$[(1 + \alpha)\lambda + (1 + 2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} \le [1 + \lambda + 3\alpha]^{1-\delta} - 1 - \lambda^{1-\delta} \tag{9}$$

$$\le (1 - \delta)(\lambda + 3\alpha) - \lambda^{1-\delta} \tag{10}$$

$$\le 3\alpha + \lambda - \delta\lambda - \lambda^{1-\delta}. \tag{11}$$

Equation (9) is derived using the inequality $\lambda \leq 1$. Equation (10) is derived from (9) using the fact that $(1+a)^b \leq 1+ab$ for any $a \geq 0$ and $0 \leq b \leq 1$. Equation (11) is derived from (10) using the inequality $3\alpha\delta > 0$. Now let $h(\lambda) = 3\alpha + \lambda - \delta\lambda - \lambda^{1-\delta}$. Differentiating with respect to λ, we have $h'(\lambda) = 1 - \delta - (1-\delta)\lambda^{-\delta}$. For $0 < \lambda < 1$, $h'(\lambda) < 0$, and for $\lambda = 1$, $h'(\lambda) = 0$. Thus for $\alpha \leq \lambda \leq 1$, $h(\lambda)$ takes on its maximum value when $\lambda = \alpha$. Returning to (9)–(11), we have

$$[(1+\alpha)\lambda + (1+2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} \leq 4\alpha - \delta\alpha - \alpha^{1-\delta} \tag{12}$$

$$\leq (4 - \alpha^{-\delta})\alpha. \tag{13}$$

Equation (13) is derived from (12) using the inequality $\delta > 0$. The right-hand side of (13) is at most 0, provided that $\alpha \leq (\beta/24)^{1/\delta}$, as in the first case.

Finally, suppose that $1 < \lambda \leq 2$. In this case, we have

$$[(1+\alpha)\lambda + (1+2\alpha)]^{1-\delta} - 1 - \lambda^{1-\delta} \tag{14}$$

$$\leq [(1+\lambda) + 4\alpha]^{1-\delta} - 1 - \lambda^{1-\delta}$$

$$\leq (1+\lambda)^{1-\delta}(1+4\alpha)^{1-\delta} - 1 - \lambda^{1-\delta} \tag{15}$$

$$\leq \lambda^{1-\delta}\left(1 + \frac{1-\delta}{\lambda}\right)(1 + 4\alpha(1-\delta)) - 1 - \lambda^{1-\delta} \tag{16}$$

$$= \frac{1-\delta}{\lambda^{\delta}} + \lambda^{1-\delta}\left(1 + \frac{1-\delta}{\lambda}\right)(4\alpha(1-\delta)) - 1 \tag{17}$$

$$\leq 16\alpha - \delta. \tag{18}$$

Equation (14) is derived using the inequality $\lambda \leq 2$. Equation (15) is derived from (14) using the inequality $1/(1+\lambda) \leq 1$. Equation (16) is derived from (15) using the fact that $(1+a)^b \leq 1+ab$ for any $a \geq 0$ and $0 \leq b \leq 1$ (in two places). Equation (18) is derived from (17) using the inequalities $1/\lambda^{\delta} < 1$, $\lambda^{1-\delta} < 2$, $(1-\delta)/\lambda < 1$, and $1 - \delta < 1$. For $\alpha \leq \delta/16$ the right-hand side of (18) is at most 0.

The growth process fails only if a core of side length k is created where $k + 2\lceil \alpha k \rceil > N$. In this case $k > (N-2)/(2\alpha+1) > N/(2\alpha+3)$, for $N > 2\alpha+3$. Such a core must contain at least $F(k) \geq Ak^{1-\delta} \geq A(N/(2\alpha+3))^{1-\delta}$ faults. Recall that $\delta = \epsilon/2$. Thus, for sufficiently large N, $A(N/(2\alpha+3))^{1-\delta} > N^{1-\epsilon}$. So, if there are fewer than $N^{1-\epsilon}$ faults in the mesh, then no such core is created, and the growth process produces an (α-β)-ensemble of finished boxes. □

3.2. Mapping the computation. In this section, we show how to map the computation of the fault free $N \times N$ mesh G onto an $N \times N$ mesh H with $N^{1-\epsilon}$ faults. The mapping requires that an (α-β)-ensemble of finished boxes be grown in H, for some constants α and β, where $0 < \alpha < 1$ and $0 < \beta < 1$. As it turns out, β can be chosen independently of α and ϵ. So we choose β first. Next, we choose α such that for fixed β and ϵ and any set of $N^{1-\epsilon}$ faults an (α-β)-ensemble of boxes can be grown using the growth process outlined in section 3.1.

We will use the pebbling terminology introduced in section 2 to describe the mapping. The mapping is produced by a *mapping process* that progresses iteratively in rounds. Initially, an s-pebble for each node of G is mapped to the corresponding node of H. Like a regular mesh computation, each s-pebble gets c-pebbles from the s-pebbles mapped to the neighboring nodes in H. The mapping process selects a finished box in the ensemble at the beginning of each round in the decreasing order of their round numbers. In each round, the mapping process changes the mapping inside the selected finished box so that no s-pebbles are mapped to the core of that

finished box. This is done by removing s-pebbles from the core, duplicating some s-pebbles, and setting up constant-congestion communication paths between duplicated s-pebbles that avoid routing through any finished boxes with smaller round numbers. The mapping process terminates when all finished boxes in the ensemble have been selected.

A mapping of the computation of G to H is said to be *valid* if no s-pebble of G is mapped to a faulty node in H and no communication path between two s-pebbles passes through a faulty node in H. The initial mapping is *not* valid since it maps s-pebbles to faulty nodes of H. After all the rounds are completed, no s-pebble will be mapped to a faulty node and no communication will pass though a faulty node. Thus, the final mapping will be valid.

The computation mapped onto a finished box B is said to be *meshlike* if each node in B has exactly one s-pebble mapped to it and each s-pebble that is mapped to a node m in B receives a c-pebble from each of the s-pebbles mapped to neighboring nodes of m in B. Our mapping process will ensure that the following invariant will hold true at the beginning of every round.

INVARIANT 3.3. *Suppose that B is a finished box with round number l that is selected at some round. At the beginning of the round, for every finished box B' with round number k, $k \leq l$, either no computation is mapped to B', or a meshlike computation is mapped to B'. Furthermore, no communication path passes through any node in B'.*

The invariant is true at the beginning of the first round since every finished box has a meshlike computation mapped to it and there are no communication paths. At the end of each round, this invariant will hold true inductively. Later in this section, we will outline the steps involved in a specific round of the mapping process in which the computation within the chosen finished box is remapped. We now show that the invariant guarantees that upon termination the mapping process produces a mapping that does not map computation or communication to faulty processors.

THEOREM 3.4. *The mapping process produces a valid mapping of the computation of G into H.*

Proof. We must show that every faulty node of H has neither an s-pebble mapped to it nor a communication path passing through it in the final mapping produced by the iterative mapping process. A node v of H is said to be *active* at a particular round of the mapping process if it either has an s-pebble mapped to it or has a communication path passing through it in the beginning of this round. A node is said to be *inactive* if it is not active. In the first round, every node in H is active.

A key property of the iterative mapping process is that if in some round a node v becomes inactive it remains inactive through the remaining rounds. For a contradiction, suppose that an inactive node v becomes active. Let B be the finished box selected in the last round in which v was inactive. Since only nodes inside B are affected by the remapping, v must be in B. From Invariant 3.3 and the fact that v is inactive, it must be the case that no computation was mapped to B. This means that no computation was remapped in this round, which is a contradiction.

We now show that no faulty node remains active at the end of the mapping process. From Property 1 of an (α-β)-ensemble of finished boxes, every fault in H is contained in some core of some finished box. Let v be a faulty node in H and let B be the finished box whose core contains this fault. If v is already inactive in some round before B is selected, it will remain inactive through the rest of the rounds. Otherwise, if v is active in the round that B is selected, it follows from Invariant 3.3 that there

must be an s-pebble mapped to v but no communication paths passing through v at the beginning of this round. The remapping of computation inside B will remove the computation from v and no new communication path will pass through v. Therefore, v becomes inactive and remains that way through the rest of the mapping process. Thus, no faulty node remains active at the end of the mapping process. □

3.2.1. Remapping the computation within a finished box. In this section, we show how to remap the computation within a finished box chosen in some round of the iterative mapping process. Let B be a box with round number l and side length $(2\alpha + 1)k$ that is selected at some round of the mapping process. (In the remainder of this paper we ignore the issue of whether quantities such as $2\alpha k$ are integral.) We assume that Invariant 3.3 is true at the beginning of this round. Later we show that this invariant is true at the end of the round after the remapping. If there is no computation mapped to B at the beginning of the round, no remapping needs to be done and the invariant holds at the end of the round.

The other possibility is that a meshlike computation is mapped to the nodes of B at the beginning of this round. In this case, the computation is partitioned into two overlapping pieces, the patch and the outerskirt. The precise sizes of the patch and the outerskirt will be specified later. The set of nodes in the finished box to which the border of the patch is mapped forms a ring in the finished box, as does the set of nodes to which the inner border of the outerskirt is mapped. We call these rings the border rings, or b-rings for short. Because the patch and the outerskirt overlap, the nodes on the border of the patch have duplicates in the interior of the outerskirt that perform the same computation. Similarly, the nodes in the inner border of the outerskirt have duplicates in the interior of the patch. These duplicate nodes in the interior of the outskirt and the interior of the patch are also mapped to rings in the finished box. We call these rings the interior rings, or i-rings for short.

The region consisting of nodes in B not more than distance $\alpha k/5$ from the outer border of B is called the *outer region*. The outerskirt will be embedded in this region. Similarly, a square box of side length $3\alpha k/5$ is called the *patch region*. As shown in Figure 4, the patch region is located below and to the left of the core, and does not intersect the outer region. The patch will be embedded in this region.

The first step in remapping the computation within B is to place a b-ring and an i-ring in each of the two regions (see Figure 4). A *free ring* in the patch region or outer region is defined to be a ring that does not pass through any finished boxes B' with smaller round numbers than B. The b- and i-rings satisfy the following *ring properties*.

(1) The i-ring and the b-ring of the outer region must be free rings. Furthermore, between the i-ring and the b-ring of the outer region there must be $\Theta(k)$ free rings. The same condition must hold for the i-ring and the b-ring of the patch region. Further, the i-ring of the patch region must have side length $\Theta(k)$.
(2) For *any* constant-load embedding of s-pebbles into a side of the i-ring of one region and *any* constant-load embedding of the duplicates of these s-pebbles into the corresponding side of the b-ring of the other region, there must be paths of length $\Theta(k)$ from every s-pebble to its duplicate. These paths must have constant congestion, must be completely contained in B, and must not pass through any finished boxes B' with smaller round numbers than B.

The procedure for finding rings with these properties is outlined in section 3.4.

Having determined the i-rings and the b-rings, the next step is to determine the size and layout of the patch and the outerskirt. Recall that a meshlike computation

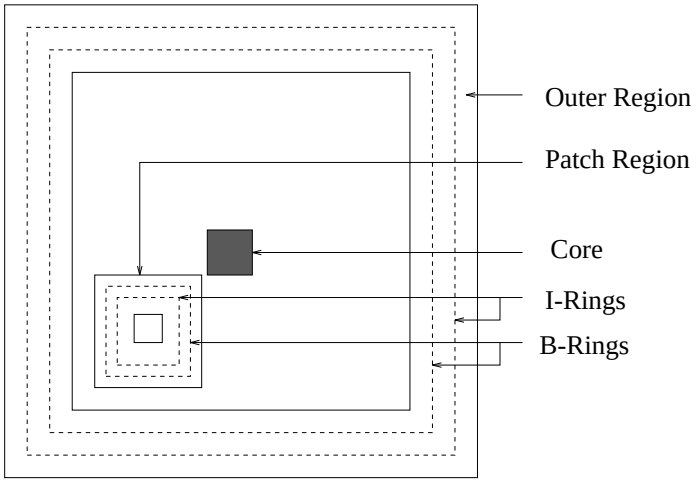


FIG. 4. *Layout of finished box B in the host.*

was mapped into box B at the beginning of this round. The size of the outerskirt and patch will depend on the choice of the i- and b-rings. The computation mapped between the b-ring of the outer region and the border of B at the beginning of this round forms the outerskirt. The computation mapped within the i-ring of the outer region at the beginning of this round forms the patch. Remapping the outerskirt and the patch to nodes within B must be done with care so that Invariant 3.3 is true at the end of this round.

The outerskirt is embedded in the region of B between the b-ring in the outer region and the border of B. Since the size and shape of the outerskirt are the same as the region in which it is embedded, we simply map each s-pebble in the outerskirt to the corresponding node in that region of B.

The patch must be embedded into the square region enclosed by the b-ring in the patch region. This is trickier since the patch is a constant factor larger in size than the square region in which it is embedded. In particular, we must ensure that each finished box B' with a smaller round number than B that intersects this square region receives a meshlike computation or receives no computation at all. A column or row in this square region that intersects such a finished box B' will be called a *bad column* or a *bad row*. The remaining rows and columns are said to be *good rows* and *good columns* respectively. Since the sum of the side lengths of such boxes B' is at most $\beta\alpha k$, if we choose $\beta < 1/10$, then a majority of the $\alpha k/5$ columns and rows will be good. Our embedding is described by two functions ρ and κ such that a node in the ith row and the jth column of the patch is mapped to the $\rho(i)$th row and the $\kappa(j)$th column of the square region. The function ρ is selected so that for all i, $\rho(i) \leq \rho(i+1) \leq \rho(i)+1$. Further, for any value of j there are at most a constant number of values of i with $\rho(i) = j$ and if the jth row is bad there is exactly one value of i with $\rho(i) = j$. The function κ is chosen with similar properties for the columns. That such functions ρ and κ exist follows from the fact that the patch is at most a constant factor larger than the square region and that a majority of the rows and columns of the square region are good. This completes the embedding of the s-pebbles to nodes within B.

Finally, the constant-congestion paths between the s-pebbles in the i-ring and

their duplicates in the b-ring are set up. These paths must not pass through any finished boxes with round numbers smaller than that of B. These paths can be set up since the i-ring and b-ring satisfy the second ring property. Using these paths, each s-pebble on the b-ring receives c-pebbles from its duplicate on the i-ring. The procedure for finding these paths is given in section 3.4.

Now we show inductively that Invariant 3.3 holds at the end of the round in which B was selected.

THEOREM 3.5. *Invariant* 3.3 *is true after the computation within* B *has been remapped.*

Proof. We must show that each finished box B' with a smaller round number than B has a meshlike computation mapped to it or no computation mapped to it at all. All such boxes B' that do not intersect B are not affected by the remapping at all. From ring property 1, we know that no box B' with a smaller round number than B can intersect the b-ring in the outer region. Therefore, any intersecting box B' not entirely contained in B must intersect the region only where the outerskirt is embedded. Since the embedding of this region does not change in the course of the remapping, all such boxes B' still have a meshlike computation mapped to them.

We will now look at boxes B' contained entirely within B. Since the b-rings and i-rings do not pass through B', either B' is contained entirely in the region where the outerskirt is embedded or entirely in the region between the b-ring of the outer region and the core of the box or entirely inside the square region where the patch is embedded. In the first case, the embedding inside B' does not change by the remapping and it continues to have a meshlike computation mapped to it. In the second case, no computation is mapped to B' and no communication path passes through the nodes in it. In the third case, observe that every row or column of B' is in a bad row or bad column of the square region. Thus ρ and κ map exactly one row and one column respectively of the patch to these rows and columns. Thus a meshlike computation is mapped to B'. Finally, none of the newly formed communication paths pass through any of the finished boxes B'. □

3.3. The emulation. In this section, we show that if the growth process terminates with a set of nonintersecting finished boxes, then the host H can emulate the guest G with constant slowdown.

In order for a node in H to create an s-pebble for a node v of G, it must receive c-pebbles for each of the edges into v in G. If this s-pebble is in the interior of a patch or an outerskirt, the s-pebbles for the neighbors of v are created either by the same node in H or by the neighbors of that node in H. Thus, the required c-pebbles can be obtained in constant time. However, the s-pebbles for the neighbors of an s-pebble on the border of the patch or on the inner border of the outerskirt may not be created nearby in H. But, since the patch and the outerskirt overlap, for every s-pebble on the border of the patch or the inner border of the outerskirt there is a duplicate in the interior of the outerskirt or the patch, respectively. In our emulation, every s-pebble on the border will receive the c-pebbles for all of its incoming edges from its duplicate that is mapped to one of the i-rings in the finished box.

The emulation consists of a series of macrosteps as in section 2.2.

LEMMA 3.6. *Each macrostep takes only a constant number of time steps to execute.*

Proof. We will prove that the maximum number of s-pebbles mapped to any node of H and the maximum number of communication paths passing through any node of H is a constant when the mapping process terminates. We prove this by

induction on the rounds of the mapping process. At the beginning of the first round of the mapping process there is exactly one s-pebble mapped to every node of H and no communication paths pass through any node, so the hypothesis is true at the beginning of the first round. Suppose that the hypothesis is true at the beginning of some round. Let the finished box selected at this round be B. There are two possibilities. If there is no computation mapped to B and no communication passing through it, no remapping is done and the hypothesis remains true at the beginning of the next round. Otherwise, from Invariant 3.3, there is a meshlike computation mapped to B and no communication path passes through any of its nodes. Remapping the computation within each finished box B causes at most a constant number of s-pebbles to be mapped to any node within it. Furthermore, the maximum number of paths created in this round that pass through a node in B is a constant. Since no communication path created before this round uses a node in B, the inductive hypothesis is true at the beginning of the next round.

Since there are only a constant number of s-pebbles mapped to any node of H, the computation step takes only constant time. Since each s-pebble can produce at most four c-pebbles in the communication step, there are at most a constant number of c-pebbles created at each step by each node. Thus, the communication step takes only constant time. Since there are only a constant number of paths passing through every node and since every c-pebble moves in every macrostep and only a constant number of c-pebbles enter a particular path at any macrostep, there can be only a constant number of c-pebbles on a particular path resident at a particular node at a particular time. Thus the routing step also takes only a constant number of time steps. □

THEOREM 3.7. *For sufficiently large N, any computation on an $N \times N$ fault-free mesh G that takes time T can be emulated by an $N \times N$ faulty mesh H with $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$) in time $O(T + N)$.*

Proof. We show that only $O(T + N)$ macrosteps are required to emulate any T-step computation of G. The final result then follows from Lemma 3.6.

Let $B_1, B_2, \ldots, B_m$ be the finished boxes in the *descending* order of their round numbers and let their side lengths be $k_1, k_2, \ldots, k_m$. The iterative mapping process produces a series of mappings, $\phi_0, \phi_1, \ldots, \phi_m$, where ϕ_i is the mapping of s-pebbles to H at the end of the ith round. The mapping ϕ_i is obtained from the mapping ϕ_{i-1} by remapping the computation within box B_i. The final mapping generated by the mapping process is ϕ_m. Note that because the guest network can be redundant, it is possible for two distinct s-pebbles s and s' to have the same label $\langle v, t\rangle$; i.e., v is the node of G whose state after t steps of computation is represented by both s-pebbles s and s'. Furthermore, it is possible for different mappings ϕ_i and ϕ_j, $i \neq j$, to map different numbers of s-pebbles to H. For example, for each node v in G and each time step t, ϕ_0 maps only one s-pebble to H. For $i > 0$, however, unless there are no faults in H, ϕ_i maps at least two s-pebbles s and s' with the same label to different nodes in H.

The dependency tree and critical path for the final mapping ϕ_m are defined as in section 2.2. In general, a sequence of s-pebbles $s_{i,T}, s_{i,T-1}, \ldots, s_{i,0}$ is called a *T-sequence* with respect to a mapping ϕ_i if, for $0 \leq j < T-1$, $t_{i,j+1} = t_{i,j}+1$, and either $v_{i,j}$ and $v_{i,j+1}$ are the same node of G and $\phi_i(s_{i,j}) = \phi_i(s_{i,j+1})$, or $v_{i,j}$ and $v_{i,j+1}$ are neighbors in G and under ϕ_i, $s_{i,j}$ sends a c-pebble to $s_{i,j+1}$. For a given mapping ϕ_i and a T-sequence $s_{i,T}, s_{i,T-1}, \ldots, s_{i,0}$, $l_{i,j}$ is 1 if nodes $\phi_i(s_{i,j})$ and $\phi_i(s_{i,j-1})$ are the same node or neighbors in H. Otherwise, $l_{i,j}$ is the length of the communication path

by which a c-pebble generated by $s_{i,j-1}$ is sent to $s_{i,j}$. With each mapping ϕ_i we associate one T-sequence. The T-sequence for ϕ_m is the critical path. For $i < m$, the T-sequence for ϕ_i is derived from the T-sequence for ϕ_{i+1}. Let $\langle v_{i+1,j}, t_{i+1,j}\rangle$ be the label of $s_{i+1,j}$. If ϕ_{i+1} does not map $s_{i+1,j}$ to the border of the patch or the outerskirt of box B_{i+1}, then both ϕ_{i+1} and ϕ_i map a single pebble labeled $\langle v_{i+1,j}, t_{i+1,j}\rangle$ to H. In this case, $s_{i,j}$ is the pebble with label $\langle v_{i+1,j}, t_{i+1,j}\rangle$ that ϕ_i maps to H. Otherwise, ϕ_{i+1} maps two duplicate s-pebbles, both labeled $\langle v_{i+1,j}, t_{i+1,j}\rangle$, to box B_{i+1}, but ϕ_i maps only one pebble with that label to B_{i+1}. In this case, $s_{i,j}$ is the one s-pebble with label $\langle v_{i+1,j}, t_{i+1,j}\rangle$ that ϕ_i maps to B_{i+1}.

We now show that for any T-sequence $s_{m,T}, s_{m,T-1}, \ldots, s_{m,0}$ with respect to ϕ_m, $\sum_{0<j\le T} l_{m,j}$ is $O(T+N)$. For each mapping ϕ_i, we define a series of weights c_i, d_i, and $w_{i,j}$, where $0 \le i \le m$ and $0 < j \le T$. The weights are chosen such that for any value i the following two properties are satisfied:

(1) $\sum_j l_{i,j} \le \sum_j w_{i,j} + \sum_{r\le i}(c_r + d_r)$,
(2) For all j, $w_{i,j}$ is less than some fixed constant.

We will find an upper bound on $\sum_j l_{m,j}$, and hence on T', by finding upper bounds on the $w_{m,j}$ (they will be constant) and on $\sum_{r\le m}(c_r + d_r)$ (which will be $O(N)$).

Initially, we define $c_0 = d_0 = 0$ and $w_{0,j} = l_{0,j}$ for all values of j. Since ϕ_0 simply maps the s-pebbles of G to the corresponding nodes of H, every $l_{0,j}$ and hence every $w_{0,j}$ is 1. Thus $\sum_j l_{0,j} = \sum_j w_{0,j} + c_0 + d_0$. Furthermore, for all values of j, $w_{0,j}$ can be bounded from above by a fixed constant.

We choose c_i, d_i, and $w_{i,j}$, $0 < j \le T$ as follows. Inductively assume that we have determined the weights c_r, d_r, $0 \le r \le i-1$, and $w_{i-1,j}$, $0 < j \le T$, such that the two properties listed above are satisfied. Suppose that ϕ_{i-1} maps no computation onto box B_i. Then no remapping is necessary and so $l_{i,j} = l_{i-1,j}$ for all values of j. In this case we set $w_{i,j} = w_{i-1,j}$ for all j and set $c_i = d_i = 0$. Otherwise, from Invariant 3.3, ϕ_{i-1} maps a meshlike computation onto box B_i. The mapping ϕ_i differs from ϕ_{i-1} in that s-pebbles inside box B_i are remapped. Since s-pebbles outside B_i are not affected, $l_{i,j} = l_{i-1,j}$ for all j such that $\phi_{i-1}(s_{i-1,j})$ is not in B_i. We define $w_{i,j} = w_{i-1,j}$ for all such values of j. (Note that if ϕ_{i-1} maps an s-pebble $s_{i-1,j}$ outside B_i but adjacent to its border and maps $s_{i-1,j-1}$ to the border of B_i, then $l_{i,j} = l_{i-1,j}$. This is because the remapping inside B_i will not the change the location of the pebble with the same label as $s_{i-1,j-1}$.)

We now look at s-pebbles mapped inside B_i by ϕ_{i-1}. The new mapping ϕ_i introduces communication paths for s-pebbles $s_{i,j}$ such that $\phi_{i-1}(s_{i-1,j})$ lies on a b-ring of B_i. For all such s-pebbles $s_{i,j}$, $l_{i,j}$ equals the length of the communication path in B_i which is $\Theta(k_i)$, where k_i is the side length of B_i. For all other s-pebbles $s_{i,j}$, $l_{i,j} = l_{i-1,j}$. We determine the weights $w_{i,j}$ for each s-pebble $s_{i,j}$ such that $\phi_{i-1}(s_{i-1,j})$ is in B_i as follows. We will consider every maximal subsequence, $s_{i-1,h+p}, s_{i-1,h+p-1}, \ldots, s_{i-1,h}$, of the T-sequence such that $\phi_{i-1}(s_{i-1,h+q})$ is in B_i for $0 \le q \le p$. Let I be a set of integers q such that $l_{i,h+q} > 1$. There are three cases depending on the value of $|I|$.

If $|I| = 0$, there are no communication paths and for every q such that $0 \le q \le p$, $l_{i,h+q} = l_{i-1,h+q} = 1$. Therefore, we will define $w_{i,h+q} = w_{i-1,h+q}$ for every $0 \le q \le p$ and set $c_i = d_i = 0$.

If $|I| \ne 0$, let $L = \sum_{q\in I}(l_{i,h+q} - l_{i-1,h+q})$, which equals the net increase in the values of $l_{i,h+q}$ in the subsequence.

If $|I| = 1$, there is exactly one communication path. Note that this can happen only if either $\phi_{i-1}(s_{i-1,0})$ or $\phi_{i-1}(s_{i-1,T})$ is in box B_i. This is so because a maximal

subsequence of s-pebbles mapped to B_i that contains neither $s_{i-1,0}$ nor $s_{i-1,T}$ must necessarily begin and terminate with s-pebbles mapped to the border of B_i. Therefore such a subsequence must necessarily use communication paths an even number of times. If $\phi_{i-1}(s_{i-1,0})$ is in B_i, make $c_i = L$, where $L = \Theta(k_i)$ (since there is only one path). Otherwise set $c_i = 0$. Similarly, if $\phi_{i-1}(s_{i-1,T})$ is in B_i, make $d_i = L$. Otherwise set $d_i = 0$. For every $0 \le q \le p$, set $w_{i,h+q} = w_{i-1,h+q}$.

If $|I| > 1$, let J be the set of integers q such that $\phi_i(s_{i-1,h+q})$ is in some free ring either in the patch region or in the outer region. Recall that nodes in the free rings are not contained in any finished box B_l, $l > i$. The value of L is $|I|\Theta(k_i)$ since each communication path in B_i is $\Theta(k_i)$ in length. This increase must be distributed evenly among the weights of the s-pebbles $s_{i,h+q}, q \in J$. Thus for all $q \in J$, $w_{i,h+q} = w_{i-1,h+q} + L/|J|$. For any two s-pebbles $s_{i-1,h+q_1}$ and $s_{i-1,h+q_2}$ such that $q_1 < q_2$ and $q_1, q_2 \in I$, the subsequence $s_{i-1,h+q_1}, \ldots, s_{i-1,h+q_2}$ contains at least $\Theta(k_i)$ s-pebbles $s_{i-1,h+q'}$, such that $q' \in J$. This is so because the i-ring and the b-ring of the patch region or the outer region were chosen such that there are $\Theta(k_i)$ free rings between them. This implies that $|J| = \Theta(k_i|I|)$ and thus $L/|J|$ is a constant. For all other $q \notin |J|$, $l_{i,h+q} = l_{i-1,h+q}$ and $w_{i,h+q} = w_{i-1,h+q}$. We also set $c_i = d_i = 0$. After all such subsequences have been dealt with we go to the next iteration.

The weight assignments in all three cases maintain the condition that $\sum_j l_{i,j} \le \sum_j w_{i,j} + \sum_{r \le i}(c_r + d_r)$. Further, for all i and j, $w_{i,j}$ is at most a constant. This is so because if the weight of some $s_{i-1,j}$ increases at the ith iteration, i.e., $w_{i,j} > w_{i-1,j}$, then it will never increase again since $\phi_{i-1}(s_{i-1,j})$ is in a free ring of the finished box selected in the ith iteration and hence is not contained in any of the finished boxes with smaller round numbers that will be considered in future rounds. Thus its weight will never change after this iteration. Further, as we saw earlier, the increment $w_{i,j} - w_{i-1,j}$ is also a constant.

We bound $\sum_j l_{m,j}$ by bounding $\sum_j w_{m,j}$ and $\sum_{i \le m}(c_i + d_i)$. The fact that $w_{m,j}$ is a constant for all j implies that $\sum_j w_{m,j}$ is $O(T)$. We bound the summation $\sum_i (c_i + d_i)$ as follows. The value of c_i or d_i is either zero or $\Theta(k_i)$. Thus $\sum_i c_i$ can be no more than the sum of the side lengths of all the boxes in the (α-β)-ensemble, i.e., $\sum_i k_i$. We show that this quantity is $O(N)$. By the proof of Theorem 3.2 we know that the core of each B_i has at least $Ak_i^{1-\epsilon/2} > Ak_i^{1-\epsilon}$ faults, where A is a constant. Since each fault is contained in a unique core and there are at most $N^{1-\epsilon}$ faults in the mesh,

$$\sum_i Ak_i^{1-\epsilon} \le N^{1-\epsilon}.$$

The maximum value of $\sum_i k_i$ that satisfies the above constraint occurs when all but one of the values of k_i equal zero, i.e., when one value of k_i is $\Theta(N)$ and the rest are zero. Thus $\sum_i k_i$ and hence $\sum_i c_i$ is $O(N)$. Similarly, $\sum_i d_i$ can be shown to be $O(N)$. Therefore,

$$\sum_j l_{m,j} \le \sum_j w_{m,j} + \sum_{i \le m}(c_i + d_i) = O(T + N). \qquad \square$$

3.4. Finding the i- and b-rings. In this section, we show how to find i- and b-rings in a finished box B in H with side length $(2\alpha + 1)k$ satisfying the two ring properties listed in section 3.2.1.

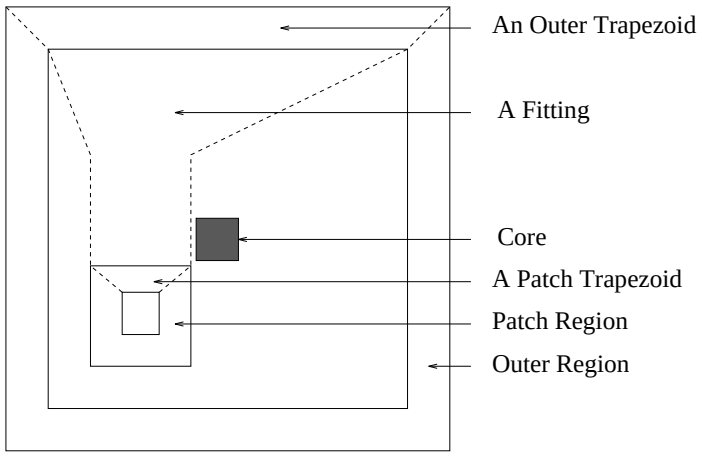


FIG. 5. *Finding i- and b-rings in H.*

Recall that the patch region is a square region in B of side length $3\alpha k/5$ and the outer region is an annular region of width $\alpha k/5$. In the center of the patch region, we place a square of size $\alpha k/5$ (see Figure 5). The i-ring of the patch is required to enclose this square. This guarantees that the i-ring of the patch has size $\Theta(k)$. A *trapezoid* is a four-sided figure consisting of two parallel sides and two nonparallel sides. We define the ith column of a trapezoid to be the set of nodes in the trapezoid at a distance i from the longer parallel side of the trapezoid. By joining the corners of the square in the patch region to the respective corners of the patch region, we partition the patch region, excluding the area enclosed by the square, into four trapezoidal regions (one of these regions is shown in Figure 5). Similarly, the outer region is also partitioned into four trapezoidal regions by joining each corner of the box B to the corresponding corner of the square forming the inner boundary of the outer region. Each ring in the outer or patch region consists of four sides, and each side is a column of one of the trapezoids.

We will define four distinct *zones*, each of which is made up of three parts. (A zone is marked with dotted lines in Figure 5.) The first part of a zone consists of one of the four trapezoids in the outer region (called the *outer trapezoid*) and the last part consists of the corresponding trapezoid in the patch region (called the *patch trapezoid*). The middle part is called the *fitting* and joins the outer trapezoid to the patch trapezoid (see Figure 5). The fitting is either a trapezoidal region or a rectangular region adjoining a trapezoidal region. (The patch region is positioned below and to the left of the core so that this is true.) Each of the four sides of a ring in the patch region or the outer region is a trapezoidal column in one of the four zones. Choosing b- and i-rings is equivalent to finding two trapezoidal columns in the patch trapezoid and two trapezoidal columns in the outer trapezoid of each of the four zones. Further, the four trapezoidal columns, one in each zone, that correspond to a particular ring must be chosen so as to have the same column number.

Since it is easier to work with rectangular grids than trapezoids, we will embed each of the four zones into a single rectangular grid R with $\alpha k/5$ rows and $3\alpha k/5$ columns. Note that R is not part of the guest or the host. It is a tool of the construction only. The grid formed by the first $\alpha k/5$ columns of R is called the *outer grid*, the next $\alpha k/5$ columns the *fitting grid*, and the last $\alpha k/5$ columns the *patch grid*.

The outer trapezoid, the fitting, and the patch trapezoid of each zone are embedded into the outer grid, fitting grid, and patch grid, respectively.

Since the outer trapezoid and the patch trapezoid each have $\alpha k/5$ columns, embedding them into their respective grids can be done by embedding the nodes in the ith trapezoidal column of the outer or the patch trapezoid to nodes in the ith column of the respective grid. Any constant-load and constant-dilation embedding will do for our purposes. We describe such an embedding below. The nodes in each trapezoidal column are grouped into $\alpha k/5$ groups such that each group contains some constant number of consecutive nodes of the column. Further, the cardinality of any two groups in a column differ by at most one and for all $i < j$ the cardinality of the ith group is at most the cardinality of the jth group of the same column. For every trapezoidal column, the ith such group is mapped to the ith node of the corresponding column of the grid. The dilation of this embedding is at most two and the load is constant.

Note that the four sides that form a ring either in the outer region or in the patch region are embedded into the same column in the outer grid or the patch grid, respectively. Therefore, a column in the outer or patch grid corresponds to the ring in the outer or patch region that gets mapped to it.

We can use the above technique to embed the fitting as well. The only difference is that since the fitting may have more than $\alpha k/5$ columns, we may have to embed a constant number of columns of the fitting in each column of the fitting grid.

We define regions in R called *obstacles* as follows. The region of R to which a finished box B' (or a portion of it) with a smaller round number than B is embedded is defined to be an obstacle. Note that the total perimeter of the obstacles in R is at most some constant times the total perimeter of the intersecting region of B.

A *free column* of R is defined to be one that does not pass through any obstacles. From the correspondences between columns in R and rings in the finished box B, in order to find i- and b-rings in B with the required ring properties, it is sufficient to choose i- and b-columns in R with the following *column properties*.

(1) The i-column and the b-column of the outer grid must be free columns. Further, between the i-column and the b-column of the outer grid there must be $\Theta(k)$ free columns. A similar condition must hold for the i-column and the b-column of the patch grid.
(2) The nodes of the i-column in one grid can be connected to the b-column of the other grid in any permutation using constant-congestion paths of length $\Theta(k)$ that do not pass through any of the obstacles.

Note that from the definition of the obstacles, if path p in R avoids all obstacles, then the paths in the four zones that are mapped to p also avoid all the finished boxes with smaller round numbers than B.

For technical reasons, we would like the placement of the obstacles in the outer grid, the patch grid, and the entire rectangular grid R itself to be symmetric about the columns in the centers of these respective grids, i.e., the obstacles in the first half of the columns of the grid are a mirror image of the obstacles in the second half of the columns of each of these three grids. To satisfy this condition we first copy every obstacle in one half of the outer grid to the other half by reflecting this obstacle about its center column. We do the same for the patch grid and then finally for the entire rectangular grid R. This copying can increase the perimeter of the obstacles by at most a constant factor.

We define a square box in R to be *flowless* as follows.

DEFINITION 3.8. *A square box F of side length q is said to be flowless if and*

only if either more than $q/4$ rows or more than $q/4$ columns pass through obstacles.

A column of R that does not intersect any of the flowless boxes is called a *live column*. Note that a live column is also a free column, since a box of size 1 that is not flowless can contain no obstacles.

THEOREM 3.9. *For a small enough value of β, a majority of the columns in the outer grid, fitting grid, and patch grid are live columns.*

Proof. Let f denote the number of columns in R that are not live. We can bound f in terms of the total perimeter of the obstacles in the grid by the following counting argument. Initially, let each nonlive column have one unit of credit associated with it. The total amount of credit in the system is f. For each nonlive column h let the largest flowless box that intersects h be box H. This nonlive column distributes its unit of credit evenly to nodes on the perimeters of the obstacles contained entirely in H. After every nonlive column has redistributed its unit of credit, the total number of credits in the nodes on the perimeters of the obstacles is still f.

Now we will determine the maximum credit received by any node s on the perimeter of an obstacle. Node s may receive credits from many different nonlive columns. First we look at nonlive columns with smaller column numbers than the column of s. Let the farthest such column from s that contributes to s be at a distance q from s. The flowless box F that intersects this nonlive column and contains s must have side length at least q. The total perimeter of the obstacles of a flowless box of size at least q must be at least $q/4$, since such a flowless box has at least $q/4$ rows or $q/4$ columns that pass through obstacles. Therefore the contribution of this nonlive column is at most $4/q$. Further, note that every nonlive column between this nonlive column and the column of s can contribute at most $4/q$. This is because box F intersects all these columns and hence the size of the largest flowless box intersecting these columns is at least q. Thus the total contribution to s from nonlive columns with smaller column numbers than its own column is at most $q \cdot 4/q$, which equals 4. Similarly the total contribution to s from nonlive columns with greater column numbers than its own column can also be bounded by 4. Therefore s receives at most eight credits.

We will now bound f, the number of columns that are not live. The perimeter of the obstacles is at most some constant c (independent of α and β) times the sum of the side lengths of the finished boxes in the intersecting region of B, i.e., at most $c\beta\alpha k$. Thus the total number of credits in the nodes on the perimeters of the obstacles is at most $8c\beta\alpha k$, so $f \leq 8c\beta\alpha k$. For $\beta < 1/80c$, the majority of the $\alpha k/5$ columns in each of the outer, fitting, and patch grids are live columns. □

3.4.1. Permuting grids. We define a *permuting grid* as follows. An $l \times m$ rectangular grid (i.e., a grid with l rows and m columns) is said to be a permuting grid if

(1) Each node on the left side of the grid is connected by a path to a distinct node on the right side of the grid. The paths have constant congestion, do not pass through any obstacles, and each path has length $\Theta(m)$. These paths are called the *horizontal paths*.
(2) There are at least $m/4$ paths from nodes on the top side of the grid to nodes on the bottom side of the grid such that the congestion of these paths is also a constant. These paths do not pass through any of the obstacles and each path has length $\Theta(l)$. These paths are called the *vertical paths*.

Note that the horizontal and vertical paths in a permuting grid are required to satisfy different properties.

LEMMA 3.10. *The l nodes on the left side of an $l \times m$ permuting grid can be connected in any permutation to the nodes on the right side of the grid using constant-congestion paths of length $\Theta(m)$ that do not pass through any obstacles, provided that $l = O(m)$.*

Proof. The idea is to use the horizontal and vertical paths in the grid as a crossbar. Each node on the left side of the permuting grid is assigned a vertical path such that no vertical path is assigned more than $4l/m$ nodes. Let π denote the permutation to be routed. A path from node v in the left side to $\pi(v)$ in the right side is routed in three stages. In the first stage, a path is routed from v along the horizontal path originating at v to the node where this horizontal path first meets the vertical path assigned to v. In the next stage, the path goes along this vertical path to the node where this vertical path first meets the horizontal path ending at $\pi(v)$. In the last stage, the path goes along this horizontal path to the destination node $\pi(v)$. It is easy to see that this path has length $\Theta(l + m) = \Theta(m)$.

The total congestion on any node in the grid can be split into a sum of three parts. The congestion of a node due to paths in the first (last) stage is at most the congestion of the horizontal paths and hence is constant. The congestion due to paths in the middle stage is constant since it is at most $4l/m$ times the congestion of the vertical paths. Hence for $l = O(m)$ the net congestion is a constant. □

We choose the i- and b-columns from the live columns in the outer and patch grids. Recall that live columns do not pass through flowless boxes. We choose β as small as is required by Theorem 3.9 so that the majority of the columns in the outer, fitting, and patch grids are live columns. Note that since the obstacles are symmetric about the middle column of the outer grid, the live columns of the outer grid are symmetric about the middle column as well. Similarly, the live columns in the patch grid and the entire rectangular grid R are symmetric about their middle columns. The live columns with the smallest column number in the outer grid and the patch grid are chosen to be the i-column of the outer grid and the b-column of the patch grid, respectively. The live columns with the largest column number in the outer grid and the patch grid are chosen to be the b-column of the outer grid and the i-column of the patch grid, respectively. The i- and b-rings in the finished box B are the rings that correspond to the chosen i- and b-columns. Let the grid between the i-column and b-column in the outer grid be O, the grid between the b-column of the outer grid and b-column of the patch grid be F, and the grid between the b-column and i-column of the patch be P.

THEOREM 3.11. *The grids* O, F, *and* P *are permuting grids.*

Proof. First we show that O is a permuting grid. O is an $\alpha k/5 \times m$ grid, for some $m \leq \alpha k/5$. Since i- and b-rings in the outergrid are the leftmost and rightmost live columns, respectively, O contains at least $\alpha k/10 \geq m/2 \geq m/4$ live columns. These live columns can serve as the vertical paths in the grid. Now we grow constant congestion paths that do not hit obstacles from every node in the i-column that forms the left side of O to the corresponding node in the b-column that forms the right side of O. These will serve as the horizontal paths of the permuting grid.

The first step is to grow constant-congestion paths that do not hit obstacles from every node in the i-column to nodes in the middle column of O. Every node in the middle column need not have a path ending in it but every node in the i-column must have a path originating in it. We define a series of square boxes of side length $2^i, 0 \leq i \leq \log_2(\alpha k/5)$. (For simplicity, we assume that $\alpha k/5$ is a power of 2.) The left side of every box consists of a set of consecutive nodes in the i-column. All the boxes

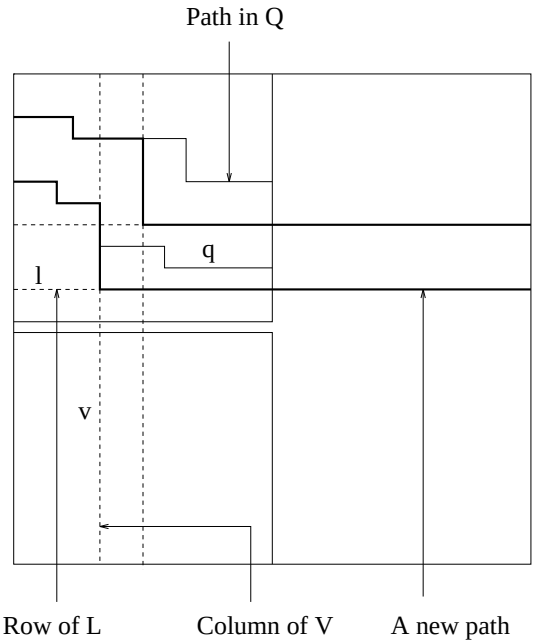


FIG. 6. *A new path constructed from path q, column v, and row l.*

of a particular size are numbered consecutively starting from the top and ending at the bottom of the i-column. Boxes of size 1 are the individual nodes in the i-column itself. Given boxes of size 2^i, we obtain boxes of size 2^{i+1} by enclosing every odd numbered box of size 2^i and the succeeding even numbered box with a box of size 2^{i+1}. There will be exactly one box of the largest size and this box encloses O. It is important to note that none of these boxes are flowless since the i-column is a live column.

We grow paths iteratively from smaller sized boxes to larger sized boxes. At the beginning of the ith iteration, we assume that each box of size 2^{i-1} has paths with maximum length $4 \cdot 2^{i-1}$ originating from every node on its left side and ending at some node on its right side. The paths are grouped into sets of 8, such that two paths share an edge only if they belong to the same group. Hence, the congestion of the paths is at most 8. We show how to construct the groups of 8 paths with maximum length $4 \cdot 2^i$ for every box of size 2^i.

For $i = 0$, it suffices to observe that each box of size 1 has no obstacles in it, since these boxes are not flowless. For $i > 0$, each box of size 2^i encloses two smaller boxes of size 2^{i-1} (see Figure 6). Let D denote the rectangular box formed by the first half of the columns of the big box. D encloses both of the smaller boxes of size 2^{i-1}. Since the big box is not flowless, there are at least $(3/4)2^i$ rows that do not hit obstacles (call this set of rows L) and at least $(3/4)2^i$ columns that do not hit obstacles. Of these $(3/4)2^i$ columns, at least $(1/4)2^i$ columns lie within D (call this set of columns V). Let Q denote the set of paths that are inductively assumed to exist inside both the smaller boxes. We use L and V to extend the paths in Q to nodes on the right side of the big box (see Figure 6). The paths in Q can be ordered sequentially from top to bottom. Likewise, the columns in V are ordered sequentially from left to right and the rows in L are ordered sequentially from top to bottom.

First we group the nodes on the left side of the big box into consecutive groups of size 8 each. To the nodes in the ith such group we assign the ith row in L. The paths from the left side of the big box to the right side of the big box are grown sequentially starting from the first group of nodes.

The first group of nodes uses paths in Q until they hit the centermost column in V. Then each of these eight nodes uses this column in V to reach its assigned row in L. Then it takes this assigned row to reach the right side of the big box (see Figure 6). When routing the next group we must make sure that we do not overlap these paths with the paths already routed since this would increase the congestion. Let w be the column in V that was used by the previous group of nodes. Further suppose that the last node in this group turned upward into column w to reach its row in L. In this case, we use the column in V that succeeds w for the current group of nodes. Similarly, if the last node of the previous group turned downward into column w, we use the column in V that precedes w for the current group. As before, the paths in the current group follow paths in Q until they hit the chosen column in V and then use this column until their assigned row in L. These paths do not share edges with any of the previous paths. We use this procedure to route paths from all the groups of nodes. Since we have $2^i/8$ columns to the right and to the left of the centermost column in V and since there are at most $2^i/8$ groups of nodes, we will never run out of columns in V. Since the paths outside of a group do not overlap, the congestion is at most 8. The maximum length of any path is at most the maximum length of any path in Q ($4 \cdot 2^{i-1}$ by the inductive hypothesis) added to the maximum length of the newly added portion (at most $2 \cdot 2^i$) which is $4 \cdot 2^i$.

After constructing paths in progressively larger boxes, we will have constructed a path from every node of the i-column to the right side of a square box of size $\alpha k/5$. These paths can be truncated at the middle column of O. Note that the obstacles in O are symmetric about its middle column. From this symmetry, exactly the same paths reflected about the middle column connect every node in the b-column to the same set of nodes in the middle column. Concatenating these two sets of paths, we obtain paths from every node in the b-column to a corresponding node in the i-column of congestion at most 8 and length at most $8\alpha k/5 = \Theta(m)$. Thus O is a permuting grid.

The proof that F and P are permuting grids is similar. □

THEOREM 3.12. *The b- and i-columns satisfy both of the column properties.*

Proof. The first property is true since the i- and b-column of the outer grid or the patch grid are chosen so that there are $\Theta(k)$ live columns between them. The second property follows from Lemma 3.10 and Theorem 3.11. To connect the nodes of the i-column of the patch grid to the b-column of the outer grid in some arbitrary permutation, we use the permuting grid P followed by the permuting grid F. One of the grids will be used to route the required permutation and the other will route the identity permutation. From Lemma 3.10, the paths obtained have constant congestion and do not pass through obstacles. Furthermore, each path is $\Theta(k)$ in length. To connect the nodes from the b-column of the patch grid to the i-column of the outer grid in an arbitrary permutation, we use grid F followed by grid O. □

4. A limit on the fault-tolerance of linear arrays. Unlike two-dimensional arrays, one-dimensional arrays are not very fault tolerant. For example, placing $f(N)$ evenly spaced faults in an N-node linear array splits the array into disjoint pieces of size $N/f(N)$, for any function $f(N)$. Emulating the entire linear array on one of these pieces entails a slowdown of at least $f(N)$. Thus if $f(N)$ grows as a function of N, the slowdown is not constant. However, if we assume a weaker model of faults

in which a faulty node cannot perform any computation but can communicate with its neighbors, then the linear array becomes more fault tolerant. In particular, the following theorem shows that an N-node linear array can tolerate $\log^k N$ worst-case faults, for any constant $k > 0$, and still emulate a fault-free N-node linear array with constant slowdown.

THEOREM 4.1. *For any constant $k > 0$, an N-node linear array with $\log^k N$ worst-case faults can emulate T steps of any computation of a fault-free N-node linear array in $O(T + N)$ steps, provided that faulty nodes can communicate with their neighbors.*

Proof. It is straightforward to apply the emulation scheme from section 2 to the linear array. □

In the remainder of this section we show that an N-node linear array with more than $\log^{O(1)} N$ worst-case faults cannot perform a static emulation of a fault-free N-node array with constant slowdown.

4.1. Bounding the load, congestion, and dilation. For the sake of convenience, we repeat the definition of a static emulation here. In a *static* emulation, a *redundant* guest network $G' = (V', E')$ is embedded in the host H. The redundant network is defined as follows. For every node v in the guest network $G = (V, E)$, there is a set of nodes $\pi(v)$ in V'. Each set $\pi(v)$ contains at least one node, and for $u \neq v$, $\pi(v)$ and $\pi(u)$ are disjoint. We call the nodes in $\pi(v)$ the *instances* of v in G'. The network G' is called redundant because it may contain several instances of each guest node. For every node $v' \in \pi(v)$ and every edge (u, v) in E, the redundant network contains a directed edge (u', v') for some $u' \in \pi(u)$. The embedding maps nodes of G' to nonfaulty nodes in the host, and edges of G' to paths in the host. In this section we allow the paths to pass through faulty host nodes.

The host emulates T steps of the guest network's computation as follows. The embedding of G' into H maps a set $\psi(a)$ of nodes of G' to each host node a. Node a emulates each node $v' \in \psi(a)$ by creating an s-pebble $\langle v', t\rangle$ for $1 \leq t \leq T$. An s-pebble $\langle v', t\rangle$ represents the state of node v' at time t. Initially, each node a of H contains s-pebbles $\langle v', 0\rangle$ for $v' \in \psi(a)$. Node a can create an s-pebble $\langle v', t\rangle$ only if it is not faulty, has already created an s-pebble $\langle v', t-1\rangle$, and has received all of the c-pebbles of the form $[e, t-1]$, where e is an edge (u', v') into v'. A c-pebble $[e, t-1]$ represents the communication that v' receives from its neighbor u' in step $t-1$. After creating an s-pebble $\langle v', t\rangle$, a nonfaulty node a can create all of the c-pebbles of the form $[g, t]$ for each edge g out of v'. At each host time step a nonfaulty host node a can create a single s-pebble (and the corresponding c-pebbles). In this section, and this section only, we assume that any node, faulty or nonfaulty, can send and receive one c-pebble on each of its edges at each step. A c-pebble for an edge (u', v') is sent along the path from u' to v' that is specified by the embedding. Note that a node u' may send c-pebbles to a neighbor v' but receive c-pebbles from a different instance v'' of guest node v.

The following three lemmas show that if a static emulation has slowdown s, then the load and congestion of the embedding of G' into H cannot exceed s, and the average dilation of the edges on any cycle in G' cannot exceed s.

LEMMA 4.2. *Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in Ts steps. Then the maximum load on any host node is at most s.*

Proof. Let l be the load of the embedding. Then some node a in H must emulate l nodes of G'. For each of these nodes, a must create T s-pebbles. Since a can create

at most one s-pebble at each step, the total time is at least lT. Thus, if the slowdown is s, the load can be at most s. □

LEMMA 4.3. *Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in Ts steps. Then the maximum congestion on any host edge is at most s.*

Proof. Let c be the congestion of the embedding. Then there is some host edge e through which c paths pass. For each of these paths, T c-pebbles must pass through e. Since e can transmit at most one c-pebble at each step, the total time is at least cT. Thus, if the slowdown is s, the congestion can be at most s. □

LEMMA 4.4. *Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in at most Ts steps. Then the average dilation of the edges on any cycle in G' is at most s.*

Proof. Suppose that there is a cycle of length L in G' with dilation D (the dilation of a cycle is the sum of the dilations of its edges). Let $v'_{L-1}, v'_{L-2}, \ldots, v'_0$ denote the nodes on the cycle. For any t, the s-pebble $\langle v'_0, t\rangle$ cannot be created until a c-pebble $[(v'_1, v'_0), t-1]$ arrives at the host node that emulates v'_0. Since a c-pebble can traverse at most one host edge at each time step, the time for the c-pebble to travel from the node that emulates v'_1 to the node that emulates v'_0 is at least the dilation of the edge (v'_1, v'_0). The dilation is also a lower bound on the time between the creation of s-pebbles $\langle v'_1, t-1\rangle$ and $\langle v'_0, t\rangle$. Working our way around the cycle, we see that the time between the creation of s-pebbles $\langle v'_0, t-L\rangle$ and $\langle v'_0, t\rangle$ is at least the dilation of the cycle, D. Thus, for any T that is a multiple of L, the time between the start of the emulation and the creation of s-pebble $\langle v'_0, T\rangle$ is at least TD/L. For $D/L > s$, this pebble is not created until after step Ts, a contradiction. □

4.2. Bounding the number of faults.

THEOREM 4.5. *For any s, there is a pattern of $h(s)(\log N)^{2s}$ worst-case faults, for any $h(s) > 2^{6s+4}s^{6s+5}$, such that it is not possible for an N-node host linear array with these faults to perform a static emulation of an N-node guest linear array with slowdown s.*

Proof. We begin by placing a layer of $g(s)$ blocks of $f(s)$ consecutive faults in an N-node array so that the number of nonfaulty nodes in the gap between each pair of blocks is at most $N/g(s)$. Formulas for $g(s)$ and $f(s)$ will be determined later.

Next we find a block of faults B that some edges of the redundant network must cross. Because the slowdown is s, and at most $N/g(s)$ host nodes lie between any pair of blocks for $g(s) > s$, it is not possible for the entire emulation to take place in one gap. (If it did, then the load in the gap would be greater than s, which is forbidden by Lemma 4.2.) Since the emulation uses host nodes in at least two gaps, there must be some block B such that some, but not all, of the the guest nodes are emulated on its left, and some, but not all, of the guest nodes are emulated on its right.

Now we find a cycle C in the redundant network G' that crosses B. Let u be a node in the guest network G such that every instance of u in G' on the right side of B receives its left input from the left of B. If there is no such u, then let u be a node in the guest network such that every instance of u on the right side of B receives its right input from the left of B; in the latter case interchange the role of left and right inputs in what follows. Note that since we have chosen B so that the host does not emulate the entire guest on the right side of B, there must be such a node u. Select one of the instances, u', of u and follow the left input edge into u' (i.e., the input edge coming from the node in G' that corresponds to the left neighbor of u in the guest) back to where it came from. It must lead across B to some node v' in G' on the left side of B.

Now follow the left input edge into v' to some other node w' in G' (node w' may be on either side of B). Continue to follow left input edges until reaching a node x' that corresponds to the left endpoint of G. Then follow right input edges until reaching the right endpoint of G, and reverse direction again. Repeat this process until some edge of G' is used twice. When this happens, a cycle C is formed. Furthermore, the cycle C must cross B because it visits every node of G and we know that H does not emulate all of G on one side of B.

The next thing to show is that on one side of B or the other, cycle C visits at least l consecutive nodes of the guest network, where $l > f(s)/2s$, and these nodes are emulated within distance $2sl$ of B in the host. If the slowdown of the emulation is s, then by Lemma 4.4 the dilation of any cycle is at most s times the number of redundant network nodes on the cycle. (The dilation of a cycle or path is equal to the sum of the dilations of the edges on the cycle or path.) Let us define a *segment* to be a maximal subpath of C that begins with an edge that crosses block B, but does not cross B again. Note that every segment either consists of a sequence of right input edges followed by a (possibly empty) sequence of left input edges, or vice versa. Suppose that cycle C crosses block B a total of $2h$ times. Then there are $2h$ segments. Associate with each segment the dilation of the edges on the segment. Note that the average ratio of the dilation of a segment to the number of nodes on the segment must be at most s (since the ratio for the entire cycle C is at most s). Now classify segments into two types: long and short. A short segment is one containing fewer than $f(s)/s$ edges. Since every segment has dilation at least $f(s)$ (due to the first edge on the segment), the ratio of a short segment's dilation to length (number of nodes) is more than s. Since the average ratio over all of the segments is at most s, there must be some long segment whose ratio of dilation to length is at most s. If this segment has more left input edges than right input edges, then discard the right input edges and the nodes that they visit. Otherwise, discard the left input edges. We are left with some set of $l \geq f(s)/2s$ nodes emulated within distance $2sl$ of B. Suppose that there are more left input edges, and let $v'_1, v'_2, \ldots, v'_l$ denote the nodes that were visited on (say) the right side of B, where v_1 is the leftmost node in the guest network. We will call the $2sl$ host nodes on the right of B the *emulation region*. (Note that in the construction of the cycle, we visited v'_l first and v'_1 last.)

Now we show that some communication must pass over the emulation region. Although nodes $v_1, v_2, \ldots, v_l$ are consecutive in the guest network, their instances are not necessarily embedded in the host in consecutive order. Suppose that v'_i is the node embedded the farthest to the right. If $i > l/2$, then the path in the cycle from the left side of B to v'_l to v'_{l-1} and on to v'_i overlaps all of nodes v'_1, $v'_2, \ldots, v'_{l/2}$. On the other hand, if $i \leq l/2$, then the path from v'_i to v'_{i-1} to v'_1 and back across to the left side of B overlaps all of nodes $v'_{l/2+1}, v'_{l/2+2}, \ldots, v'_l$. In either case, we have a set of $l/2$ consecutive nodes in the guest network that the host emulates, and some other edges of G' overlap their emulation with congestion 1.

We now proceed recursively within the emulation region. One last issue that must be dealt with is that some of the $l/2$ nodes that the host is emulating within the emulation region may receive some of their right inputs from outside the emulation region. However, since the embedding has congestion at most s (by Lemma 4.3), at most $2s$ right inputs can enter the emulation region from outside. Thus, there must be a set of at least $(l/2)/2s = l/4s$ redundant network nodes that the host emulates within the emulation region that are consecutive in the guest and receive all of their inputs from within the emulation region. At this point we have placed $g(s)$ blocks of

$f(s)$ faults in the network and we have proved that on one side of one of the blocks, there is an emulation region of size $2sl$ in which at least $l/4s$ consecutive nodes of the guest are emulated, for some $l \geq f(s)/2s$, and some other edges of G' cause congestion 1 in the emulation region. For recursion on sets of $l/4s$ guest nodes, where $l \geq f(s)/2s$, we need $f(s) > 8s^2$.

We are now going to place an additional layer of faults in the network. Because we do not know where the emulation region is, we will place faults immediately adjacent to both sides of each of the $g(s)$ blocks of faults in the first layer. Also, because we do not know how large the emulation region is, we will place the faults in patterns of size $2, 4, 8, \ldots, N$ on top of each other. (Note that N is the size of the entire array.) In a pattern of size 2^k, we will place $g(s)$ blocks of $f(s)$ consecutive faults at spacings of $2^k/g(s)$. Thus, in each pattern there are at most $g(s)f(s)$ faults, and there are at most $\log N$ patterns on each side of the blocks in the first layer. The total number of faults in the second layer is $2g^2(s)f(s)\log N$.

The entire emulation region must lie under some pattern P of faults of size 2^k, where $2^k \leq 4sl$. The blocks of faults in this pattern are spaced at a distance of $2^k/g(s)$, which is at most $4sl/g(s)$. In this region, at least $l/4s$ guest nodes are emulated. If the slowdown is at most s, and $(l/4s)/(2sl/g(s)) > s$, then by Lemma 4.2 it is not possible for the entire emulation to be performed entirely between two blocks of faults in this pattern. (Thus, we need $g(s) > 8s^3$.) Arguing as we did for the first layer, we can show that, for some l', on one side of one of the blocks of P, there is an emulation region of size $2sl'$ and a set of least $l'/4s$ nodes that are consecutive in the guest network that receive their inputs from within the emulation region. But now two units of congestion pass over the new emulation region (possibly in opposite directions).

A third layer of faults is now placed in the network. As before, a set of patterns of faults is placed around each block in the second layer. There are $2g(s)^2 \log N$ blocks in the second layer. Thus, there are $4g(s)^3(\log N)^2 f(s)$ faults in the third layer.

By applying $2s+1$ layers of faults, we find an emulation region over which at least $s+1$ units of congestion (in one direction) pass, which is a contradiction by Lemma 4.3. The $(2s+1)$st layer contains $2^{2s}g(s)^{2s+1}(\log N)^{2s}f(s)$ faults. The total number of faults contained in all the $2s+1$ layers is at most twice the number of faults contained in the $(2s+1)$st layer alone, since the number of faults in the ith layer is at least double the number of faults in the $(i-1)$st layer. Thus the total number of faults is at most $2^{2s+1}g(s)^{2s+1}(\log N)^{2s}f(s) = h(s)\log^{2s} N$, where $h(s) = 2^{2s+1}g(s)^{2s+1}f(s)$, $g(s) > 8s^3$, and $f(s) > 8s^2$. □

5. Remarks. The scheme described in section 2 for tolerating $\log^k N$ faults in an $N \times N$ mesh can be easily generalized to tolerate $\log^k N$ faults in a d-dimensional array with side length N for any fixed $d > 2$. It seems plausible that the techniques described in section 3 can also be generalized to arrays of higher dimension.

REFERENCES

[1] M. Ajtai, N. Alon, J. Bruck, R. Cypher, C. T. Ho, M. Naor, and E. Szemerédi, *Fault tolerant graphs, perfect hash functions and disjoint paths*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 693–702.

[2] Y. Aumann and M. Ben-Or, *Computing with faulty arrays*, in Proc. 24th Annual ACM Symposium on the Theory of Computing, Victoria, BC, Canada, 1992, pp. 162–169.

[3] T. Blank, *The MasPar MP-1 architecture*, in compcon90, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 20–24.

[4] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb, iWarp, *an integrated solution to high-speed parallel computing*, in Proc. Supercomputing '88, Orlando, FL, 1988, IEEE Computer Society Press, Washington, DC, pp. 330–339.

[5] J. Bruck, R. Cypher, and C.–T. Ho, *Fault-tolerant meshes with small degree*, in Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures, Velen, Germany, 1993, pp. 1–10.

[6] M. R. Fellows, *Encoding Graphs in Graphs*, Ph.D. thesis, Department of Computer Science, University of California, San Diego, CA, 1985.

[7] J. W. Greene and A. El Gamal, *Configuration of VLSI arrays in the presence of defects*, J. ACM, 31 (1984), pp. 694–717.

[8] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, and A. Tsantilas, *Asymptotically tight bounds for computing with faulty arrays of processors*, in Proc. 31st Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 285–296.

[9] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg, *Work-preserving emulations of fixed-connection networks*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Seattle, WA, 1989, pp. 227–240.

[10] R. K. Koeninger, M. Furtney, and M. Walker, *A shared MPP from Cray research*, Digital Tech. J., 6 (1994), pp. 8–21.

[11] F. T. Leighton, B. M. Maggs, and S. B. Rao, *Packet routing and job-shop scheduling in O(congestion + dilation) steps*, Combinatorica, 14 (1994), pp. 167–180.

[12] T. Leighton and C. E. Leiserson, *Wafer-scale integration of systolic arrays*, IEEE Trans. Comput., C–34 (1985), pp. 448–461.

[13] T. Leighton, B. Maggs, and R. Sitaraman, *On the fault tolerance of some popular bounded-degree networks*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 542–552.

[14] T. R. Mathies, *Percolation theory and computing with faulty arrays of processors*, in Proc. 3rd Annual ACM–SIAM Symposium on Discrete Algorithms, Orlando, FL, 1992, SIAM, Philadelphia, pp. 100–103.

[15] F. Meyer auf der Heide, *Efficiency of universal parallel computers*, Acta Inform., 19 (1983), pp. 269–296.

[16] F. Meyer auf der Heide, *Efficient simulations among several models of parallel computers*, SIAM J. Comput., 15 (1986), pp. 106–119.

[17] F. Meyer auf der Heide and R. Wanka, *Time-optimal simulations of networks by universal parallel computers*, in Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 349, Springer-Verlag, Heidelberg, 1989, pp. 120–131.

[18] M. D. Noakes, D. A. Wallach, and W. J. Dally, *The J-machine multicomputer: An architectural evaluation*, in Proc. 20th Annual International Symposium on Computer Architecture, San Diego, CA, 1993, ACM, New York, pp. 224–235.

[19] M. O. Rabin, *Efficient dispersal of information for security, load balancing, and fault tolerance*, J. Assoc. Comput. Mach., 36 (1989), pp. 335–348.

[20] P. Raghavan, *Robust algorithms for packet routing in a mesh*, in Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures, Sante Fe, NM, 1989, pp. 344–350.

[21] E. J. Schwabe, *On the computational equivalence of hypercube-derived networks*, in Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Crete, Greece, 1990, pp. 388–397.

[22] H. Tamaki, *Efficient self-embedding of butterfly networks with random faults*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 533–541.

[23] H. Tamaki, *Robust bounded-degree networks with small diameters*, in Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA, 1992, pp. 247–256.

[24] H. Tamaki, *Construction of the mesh and the torus tolerating a large number of faults*, in Proc. 6th Annual ACM Symposium on Parallel Algorithms and Architectures, Cape May, NJ, 1994, pp. 268–277.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1612–1634, December 1997

003

MATRIX SEARCHING WITH THE SHORTEST-PATH METRIC*

JOHN HERSHBERGER† AND SUBHASH SURI‡

Abstract. We present an $O(n)$ time algorithm for computing row-wise maxima or minima of an implicit, totally monotone $n \times n$ matrix whose entries represent shortest-path distances between pairs of vertices in a simple polygon. We apply this result to derive improved algorithms for several well-known problems in computational geometry. Most prominently, we obtain linear-time algorithms for computing the geodesic diameter, all farthest neighbors, and external farthest neighbors of a simple polygon, improving the previous best result by a factor of $O(\log n)$ in each case.

Key words. shortest paths, matrix searching, geodesic diameter, farthest neighbors, geometric matching

AMS subject classifications. 68Q25, 68P05, 68PL0

PII. S0097539793253577

1. Introduction.

Matrix-searching is the popular term for a technique introduced by Aggarwal et al. [2] for computing row-wise maxima in a totally monotone matrix. A matrix M is called *totally monotone* if

$$M(i,k) < M(i,l) \implies M(j,k) < M(j,l)$$

for any $i < j$ and $k < l$. Aggarwal et al. [2] discovered the importance of totally monotone matrices and showed that many problems in computational geometry can be formulated as finding maxima in a totally monotone matrix. The problem of finding farthest neighbors in a convex polygon is a prototypical application of this technique; we repeat it here for the benefit of the uninitiated.

Consider a convex polygon P and partition its boundary into two disjoint chains $U = \{u_1, u_2, \ldots, u_p\}$ and $V = \{v_1, v_2, \ldots, v_m\}$. The vertices in each chain are ordered counterclockwise, as shown in Figure 2.1. Define a $p \times m$ matrix M that encodes pairwise distances between the vertices of U and V: $M(i,j) = d(u_i, v_j)$ for $1 \le i \le p$ and $1 \le j \le m$. One may easily check that M is totally monotone; the proof follows from the triangle inequality. We observe that a maximum entry in row i of M corresponds to the maximum distance between u_i and any vertex in V. Thus, we can find for all vertices of U a corresponding farthest neighbor in V by computing the row-wise maxima in M.

The main result of Aggarwal et al. [2] says that the row-wise maxima of a totally monotone $n \times n$ matrix can be found with only $O(n)$ comparisons and evaluations of the matrix entries. The matrix is defined implicitly—an entry is evaluated only when needed by the algorithm. If evaluating a matrix entry takes $f(n)$ time, then the row-wise maxima can be computed in time $O(nf(n))$. This result leads to linear-time algorithms for several computational geometry problems set in convex polygons, where the matrix entries correspond to Euclidean distances.

*Received by the editors August 12, 1993; accepted for publication (in revised form) October 24, 1995. A preliminary version of this paper appeared in the Proceedings of the 25th ACM Symposium on Theory of Computing, ACM, New York, 1993, pp. 485–494. The research reported here was performed while the first author was at the DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, and the second author was at Bellcore, 445 South Street, Morristown, NJ 07960.

http://www.siam.org/journals/sicomp/26-6/25357.html

†Mentor Graphics Corp., 1001 Ridder Park Drive, San Jose, CA 95131 (johnh@icx.com).

‡Department of Computer Science, Washington University, St. Louis, MO 63130 (suri@cs.wustl.edu).

In this paper, we propose a powerful extension of the basic matrix-searching algorithm. We give a linear-time algorithm for computing row-wise maxima in a totally monotone matrix whose entries represent *shortest path* distances between pairs of vertices in a simple polygon. The shortest path distance, also called the *geodesic distance*, is the natural distance measure in a simple nonconvex polygon. Observe, however, that unlike the Euclidean distance, the geodesic distance in general cannot be computed in constant time. Unless one precomputes all the distances, which requires too much preprocessing, the best data structure for computing shortest-path distances is one due to Guibas and Hershberger [6, 10], which achieves $f(n) = O(\log n)$. (After an $O(n)$ time and space preprocessing step, this data structure computes the geodesic distance between two arbitrary points of a simple polygon in $O(\log n)$ time; n denotes the number of vertices in the polygon.) A straightforward combination of the shortest-path query data structure and the basic matrix-searching gives an $O(n \log n)$ time algorithm for computing row-wise maxima of a totally monotone matrix with shortest path distances. Our result in this paper improves the time complexity to optimal $O(n)$.

Our result has applications in computational geometry: we achieve linear-time algorithms for several outstanding open problems. The most straightforward application concerns geodesic diameter and all-farthest neighbors. We obtain $O(n)$ time algorithms for them, improving the previous $O(n \log n)$ bound. A similar improvement also follows for the problem of computing the external farthest neighbors in a simple polygon.

A somewhat less obvious application concerns finding a closest visible vertex pair between two simple polygons. Although this problem does not involve the shortest path metric in its formulation, we apply matrix-searching with shortest paths to get a conceptually simple algorithm for the problem that runs in optimal linear time. Previous linear-time algorithms were quite complicated [3, 4]. In a separate paper [12] we apply this result, along with some additional techniques, to obtain a linear-time algorithm for computing a shortest diagonal of a simple polygon, improving the previous $O(n \log n)$ algorithm.

We can also compute weighted-nearest neighbors between two disjoint chains of a simple polygon in linear time; the weighted distance is defined as $d(u_i, v_j) - w(u_i) - w(v_j)$, where $d(u_i, v_j)$ is the shortest path distance, and w is a real-valued weight function. This improves the complexity of computing an optimal matching on the vertices of a polygon from $O(n \log^2 n)$ to $O(n \log n)$ [13], matching the best bound for a convex polygon.

Our paper is organized in seven main sections. In section 2, we formulate the key problem of our paper and provide a brief summary of the matrix-searching technique. In section 3, we introduce the basic geometric infrastructure required by our algorithm. The algorithm itself is described in section 4, and its analysis is presented in section 5. The applications of our main result are presented in section 6, and we conclude with some remarks in section 7.

2. Matrix-searching and the shortest-path metric. The centerpiece of our paper is an algorithm for computing farthest neighbors between two chains of a simple polygon under the shortest-path metric. In this section, we introduce the problem, define terminology, and provide a brief review of the matrix-searching framework of Aggarwal et al. [2]. Throughout, we assume that P is a simple polygon with n vertices; $\pi(x, y)$ is the shortest path between the points x and y in P; $d(x, y)$ is the length of the path $\pi(x, y)$; and y is a (geodesic) *farthest neighbor* of x if $d(x, y) =$

$\max\{d(x, x') \mid x' \in P\}$.

2.1. Farthest neighbors and the totally monotone matrix. Let $U = (u_1, u_2, \ldots, u_p)$ and $V = (v_1, v_2, \ldots, v_m)$ be two polygonal chains that together partition the vertices of P; the vertices are ordered counterclockwise in each chain, and the total number of vertices is $n = p+m$. We want to compute a farthest neighbor for each vertex of U in the chain V. This problem can be reformulated as a maxima-finding problem in the associated distance matrix.

Let M denote the $p \times m$ matrix that encodes the pairwise distances between the vertices of U and V:

$$M(i, j) \;=\; d(u_i, v_j). \tag{2.1}$$

See Figure 2.1. Our problem is to compute a maximum entry in each row of M. (The equivalence is clear: $M(i, j)$ is a maximum entry in row i if and only if v_j is a geodesic farthest neighbor of u_i.) We break ties by choosing the leftmost maximum in each row.

We use the matrix-searching framework of Aggarwal et al. [2], which depends crucially on a matrix property called *total monotonicity*. The matrix M is *totally monotone* if

$$M(i, k) \;<\; M(i, l) \quad\Longrightarrow\quad M(j, k) \;<\; M(j, l) \tag{2.2}$$

for any $1 \le i < j \le p$ and $1 \le k < l \le m$. The total monotonicity of M follows from the triangle inequality of the geodesic distance function. Total monotonicity also implies that the leftmost maxima form a "staircase" in the matrix: as we move down the rows, the maximum can move only right, but never left.

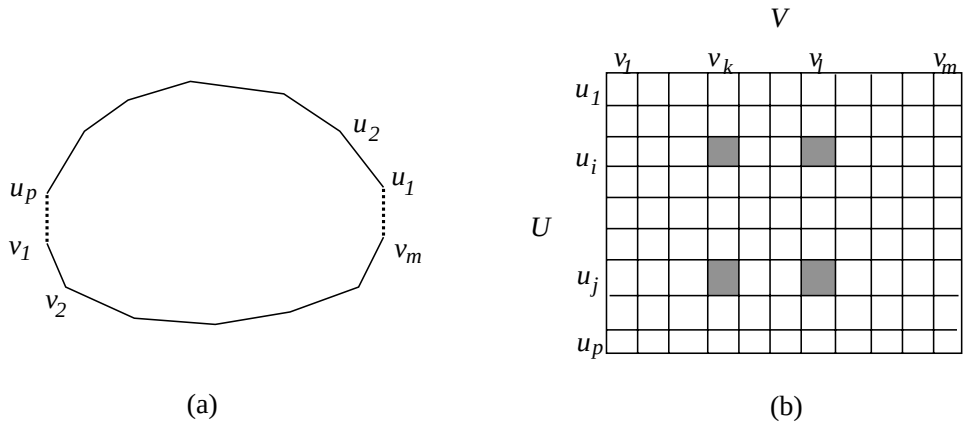


FIG. 2.1. *A polygon, two chains, and the corresponding matrix. The triangle inequality implies that* $M(i, k) + M(j, l) \ge M(i, l) + M(j, k)$, *which in turn shows that* M *is totally monotone.*

The matrix M is defined implicitly: an entry is computed only when needed. The matrix-searching algorithm performs two kinds of operations: evaluation of a matrix entry $M(i, j)$ and comparison between two matrix entries. We use the term *matrix operation* to describe either of these two steps. In general, the comparison step can always be performed in constant time, but the evaluation step can be more complicated. The main result of Aggarwal et al. [2] is the following lemma.

LEMMA 2.1 (see [2]). *The leftmost maximum in each row of a totally monotone $n \times n$ matrix can be determined using $O(n)$ matrix operations. If each matrix operation takes $f(n)$ time, then the row-maxima problem can be solved in time $O(nf(n))$.*

If we use shortest path queries [6, 10] to compute the entries of M, then $f(n) = O(\log n)$, and we get an $O(n \log n)$ time algorithm. Our contribution in this paper is an algorithm that reduces $f(n)$ to amortized $O(1)$, giving an optimal linear-time algorithm.

2.2. The matrix-searching algorithm. Our algorithm assumes some familiarity with the matrix-searching algorithm of [2]. This section gives a brief overview of the matrix-searching technique; however, the reader is encouraged to refer to the original paper for details.

The matrix-searching algorithm of [2] finds the leftmost maximum in each row of a totally monotone matrix using a linear number of matrix operations. The algorithm is recursive; it proceeds in $O(\log n)$ phases. Each phase cuts down each dimension of the matrix by half, after evaluating a linear number of matrix entries. The following pseudocode gives a top-level view of the algorithm.

MSEARCH(M)

(∗ *Compute the position and value of the leftmost maximum in each row of* M ∗)

1. $A \leftarrow$ *REDUCE*(M)
2. **if** $|A| = 1$ **then return** the value of $A[1, 1]$ and its position in M
3. $B \leftarrow$ the matrix consisting of *even* rows of A
4. *MSEARCH*(B)
5. *MFILL*(A, B) (∗ Finds maxima in the odd rows of A ∗)
6. Compute the positions and values of the maxima in M from their positions in A.

REDUCE takes a $p \times m$ matrix M, where $m \geq p$, and returns a $p \times p$ submatrix A. (If $m < p$, *REDUCE* returns just M.) The leftmost maxima in the rows of A and M are identical. *REDUCE* achieves this size reduction using only $O(m)$ matrix operations. In the pseudocode below, A^k refers to the kth column of A.

REDUCE (M)
  $A \leftarrow M$; $k \leftarrow 1$;
  while A has more than p columns **do**
    case
    $A(k,k) \geq A(k,k+1)$ and $k < p$: $k \leftarrow k + 1$.
    $A(k,k) \geq A(k,k+1)$ and $k = p$: Delete column A^{k+1}.
    $A(k,k) < A(k,k+1)$: Delete column A^k; **if** $k > 1$ **then** $k \leftarrow k - 1$.
    endcase
return A.

Figure 2.2 illustrates the pattern of comparisons performed by *REDUCE*. No entry in the shaded portion of the matrix can be a candidate for the leftmost maxima.

The code above, which is essentially in the form given by Aggarwal et al. [2], operates on a dynamic matrix: when a column is deleted, all columns to its right are renumbered. This property makes it somewhat difficult to identify which entries of the original matrix are examined. Because our algorithm depends crucially on the order in which matrix entries are computed, we rewrite the code for *REDUCE* to

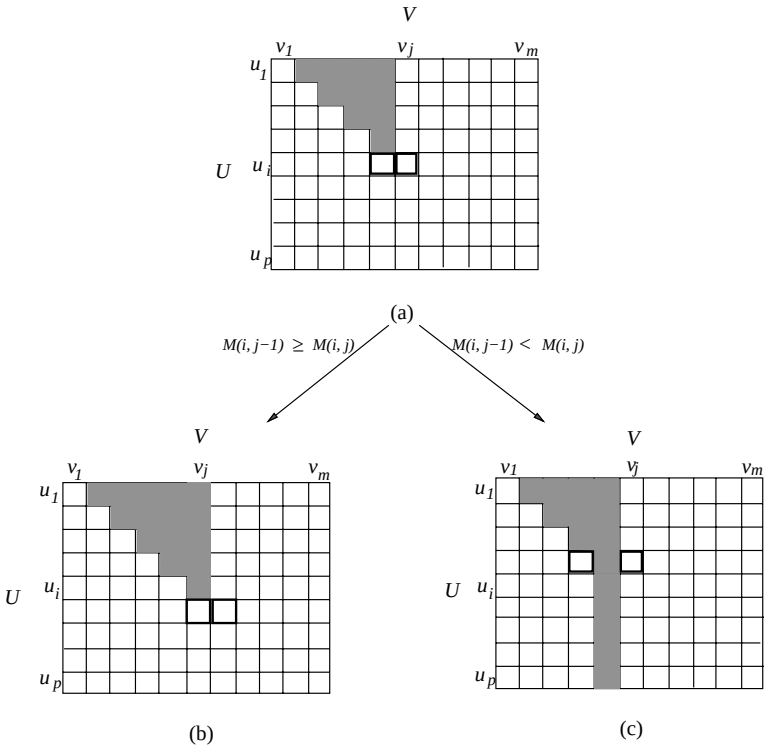


FIG. 2.2. *Illustrating REDUCE. The current comparison is between* $M(i, j-1)$ *and* $M(i, j)$*, shown by the highlighted cells in* (a)*. If* $M(i, j-1) \geq M(i, j)$*, then we arrive at Figure* 2.2(b)*: the first* i *rows of the* j*th column are eliminated, and the next comparison is between* $M(i+1, j)$ *and* $M(i+1, j+1)$*. Otherwise* $M(i, j-1) < M(i, j)$ *and we arrive at Figure* 2.2(c)*: column* $j-1$ *is eliminated entirely. In this case, the search backs up and next compares* $M(i-1, j-2)$ *and* $M(i-1, j)$*.*

clarify the matrix access pattern. The revised version below uses a *predecessor* array (essentially a linked list) to delete columns. Initially $pred[i]$ is set to $i-1$ for all i. To delete a column j such that $pred[k] = j$, the algorithm resets $pred[k] \leftarrow pred[j]$, thereby splicing j out of the list. The following code avoids some of the matrix probes in the version of *REDUCE* above by reusing previously computed values.

```
NEW-REDUCE (M)
    for j ← 1 to m do pred[j] ← j − 1;
    ncols ← m;    j ← 2;    k ← 1;
    value[1] ← M(1, 1);
    while ncols > p do
        (* Note that value[pred[j]] = M(k, pred[j]) *)
        if value[pred[j]] ≥ M(k, j) then
            if k < p then
                k ← k + 1;    value[j] ← M(k, j);
            else
                pred[j + 1] ← pred[j];    ncols ← ncols − 1;
```

```
            endif
            j ← j + 1;
        else
            pred[j] ← pred[pred[j]];    ncols ← ncols − 1;
            if k ≠ 1 then k ← k − 1 else j ← j + 1 endif;
        endif
    endwhile
return the pred[∗] linked list.
```

LEMMA 2.2. *The two algorithms REDUCE and NEW-REDUCE are equivalent.*

Proof. We need to prove the correspondence between the chain of $pred[*]$ pointers in *NEW-REDUCE* and the dynamic matrix in *REDUCE*. To this end, define the function $length(j)$ to be the length of the chain of $pred[*]$ pointers starting at j. That is, $length(j) = h$ iff $pred^h[j] = 0$, but $pred^{h-1}[j] \neq 0$.

We maintain the invariant that at the top of each loop, if column k in *REDUCE* is actually column j of the original matrix M, then $length(j) = k$ in *NEW-REDUCE*. This is established by the **for** loop of *NEW-REDUCE*. The main loop of *REDUCE* corresponds to that of *NEW-REDUCE*. At the top of each **while** loop, column j of *NEW-REDUCE* corresponds to column $k + 1$ in *REDUCE*, and column $pred[j]$ corresponds to column k.

We must prove that the matrix comparisons in *REDUCE* and *NEW-REDUCE* are equivalent. The main loop of *NEW-REDUCE* explicitly computes the value needed on the right-hand side of the comparison in that loop. The value needed on the left-hand side of the comparison is referred to as $A(k, k)$ in *REDUCE*; that is, it is the matrix element $M(k, l)$ such that $length(l) = k$ in *NEW-REDUCE* and $l = pred[j]$. So, we must show that $value[l] = M(k, l)$. If *NEW-REDUCE* does not delete column j, it sets $value[j] \leftarrow M(length(j), j)$ just before it advances to column $j + 1$. Because *NEW-REDUCE* does not change $length(l)$ for any $l < j$, it follows that $value[l] = M(k, l) = A(k, k)$. We leave it as an exercise to the reader to prove the exact correspondence between the statements of *REDUCE* and *NEW-REDUCE*. □

In a high-level view of the algorithm, *REDUCE* and *NEW-REDUCE* are completely equivalent; the latter only fleshes out more details. The $pred[*]$ list that *NEW-REDUCE* returns encodes the reduced matrix A returned by *REDUCE*. In an implementation, *MSEARCH* and *MFILL* would also use *pred* pointers; however, in order to preserve a close parallel with the basic algorithm of [2], we will use *NEW-REDUCE* only in the next subsection, and use *REDUCE* in the remainder of the paper.

Finally, the procedure $MFILL(A, B)$ computes the leftmost maxima for the odd rows of A, given the locations of the maxima for its even rows; recall that the matrix B consists of precisely the even rows of A. This phase also requires $O(m)$ matrix operations. Let C_{2j} denote the column index of the leftmost maximum in the even rows of A, for $j = 1, 2, \ldots, \lfloor p/2 \rfloor$. Then total monotonicity implies that C_{2j+1} lies between C_{2j} and C_{2j+2}. The following pseudocode implements *MFILL*. For convenience we set $C_0 = 1$ and $C_{2\lfloor p/2 \rfloor + 2} = m$.

```
MFILL (A, B)
    for i ← 1 to ⌈p/2⌉ do
        max ← A(2i − 1, C_{2i−2});
        C_{2i−1} ← C_{2i−2};
```

```
        for j ← C_{2i-2} + 1 to C_{2i} do
            if A(2i − 1, j) > max then
                max ← A(2i − 1, j);
                C_{2i-1} ← j;
            endif
        Evaluate A(2i, C_{2i});   (* Extra work, for accounting purposes *)
    endfor;
```

Our algorithm has exactly the same form as that of Aggarwal et al. [2]; only the details of evaluating a matrix entry are different. In [2], each matrix evaluation takes constant time. In our case, the evaluations are trickier and require a careful implementation of a shortest-path algorithm. To this end, we need to understand better the path followed by the matrix evaluations in the subroutine *REDUCE*.

2.3. Summary of matrix searching. Each phase of the recursive algorithm *MSEARCH* halves the matrix dimensions. The core of the algorithm is *REDUCE*, which eliminates columns of the matrix. In the kth phase of the algorithm, *REDUCE* operates on a submatrix A_k of size at most $n_k \times 2n_k$, where $n_k = \lfloor p2^{-k} \rfloor$. From here on we will assume that A_k is exactly $n_k \times 2n_k$. Matrix-searching theory tells us that *REDUCE* performs at most $3n_k$ matrix operations and returns an $n_k \times n_k$ submatrix.

We call the vertices corresponding to rows and columns of A_k *active* vertices; the remaining vertices of U and V are called *dead*. Thus, there are n_k active vertices in the U-chain and $2n_k$ active vertices in the V-chain. Because *MSEARCH* deletes even rows, the active vertices of U are uniformly spaced in U; more precisely, two adjacent active vertices are separated by $2^k - 1$ dead vertices. However, no such regularity necessarily exists among the active vertices of V.

Our algorithm makes critical use of the order in which the matrix-searching algorithm evaluates entries. This order is easier to analyze in *NEW-REDUCE* than in the equivalent routine *REDUCE*. The first matrix entry evaluated by *REDUCE* (or *NEW-REDUCE*) in the kth phase is $A_k[1, 1]$. In each new evaluation, either the row or the column index is changed by one. In particular, the movement in the matrix consists of steps that belong to $\{up,\ down,\ right\}$, where *up* and *down* decrement and increment the row-index and *right* increments the column-index. The column-index is never decremented, so there is no *left* movement in the matrix. We can break the sequence of moves into blocks that start with a *right* move. Each *right* move is followed by zero or more *up* moves and at most one *down* move. Thus the sequence of moves satisfies the following regular expression:

$$(\ right\ \ up^*\ \ (down \mid \epsilon)\ )^*, \tag{2.3}$$

where ϵ is the null move. The sequence of evaluations in *MFILL* also satisfies the same regular expression.

3. Geometric structures. In this section, we introduce the geometric infrastructure necessary for computing the shortest-path distances. We describe both the geometric properties of these structures and their implementations. We start with some notation. The shortest path between two points x and y in P is denoted $\pi(x, y)$. The two extreme vertices of the chain V play a distinguished role in our algorithm. We refer to them with the shorthand notation $a = v_1$ and $b = v_m$.

3.1. The shortest-path tree. Let T_a and T_b denote the shortest-path trees of the polygon P rooted at the vertices $a = v_1$ and $b = v_m$, respectively. (A shortest-path

tree T_p is the union of shortest paths from p to all the vertices of P.) We precompute an array that stores, for every vertex x of P, the parents of x in T_a and T_b, along with the distances from x to a and b. We first triangulate the polygon in linear time using Chazelle's algorithm [5] and then build the shortest-path trees T_a and T_b also in linear time using an algorithm of Guibas et al. [7] or Hershberger and Snoeyink [11].

LEMMA 3.1 (see [7, 11]). *After linear-time preprocessing, the distances $d(x,a)$ and $d(x,b)$, for any vertex $x \in P$, can be computed in $O(1)$ time.*

Given two arbitrary vertices $x, y \in P$, we let $\alpha(x,y)$ denote the *lowest common ancestor* of x and y in T_a. Similarly, $\beta(x,y)$ denotes the lowest common ancestor of x and y in the tree T_b. In our discussion, we so frequently use lowest common ancestors where one of the vertices is a or b that we also introduce the following abbreviated notation:

$$\alpha(x) = \alpha(x,b),$$
$$\beta(x) = \beta(x,a).$$

In Figure 3.1, $\alpha(u_{19}, v_{10}) = v_4$, $\beta(u_{19}, v_{10}) = u_{12}$, $\alpha(v_{10}) = v_4$, and $\beta(v_{10}) = v_{17}$.

We use the linear-time algorithm of Harel and Tarjan [9] or that of Schieber and Vishkin [15] to preprocess our trees for lowest common ancestor queries.

LEMMA 3.2 (see [9, 15]). *After linear-time preprocessing of T_a and T_b, we can compute $\alpha(x,y)$ and $\beta(x,y)$ in constant time for any two vertices $x, y \in P$.*

3.2. The funnel and the funnel-difference. Consider a vertex $v \in V$ and the two shortest paths $\pi(v,a)$ and $\pi(v,b)$. These two paths may coincide for some distance before they diverge at a vertex z, never to meet again. The *funnel* of v, denoted $F(v)$, is the set of edges in the shortest paths from z, called the *apex* of the funnel, to the lowest common ancestor vertices $\alpha(v)$ and $\beta(v)$:

$$F(v) \;=\; \pi(z, \alpha(v)) \;\cup\; \pi(z, \beta(v)). \tag{3.1}$$

Both $\pi(z,\alpha(v))$ and $\pi(z,\beta(v))$ are concave. See Figure 3.1 for an example of funnels.

In our analysis, the difference of two funnels turns out to be important. The idea is applicable to both U and V chains, but we define it here just for the V-chain. Given two vertices $v_i, v_j \in V$, where $i > j$, their funnel-difference is defined as

$$\Delta F(v_i, v_j) \;=\; F(v_i) \setminus F(v_j). \tag{3.2}$$

That is, $\Delta F(v_i, v_j)$ is the set of edges that are in $F(v_i)$ but not in $F(v_j)$. The edges of $\Delta F(v_i, v_j)$ form a contiguous portion of $F(v_i)$ surrounding v_i. The following property of the funnel-difference will be most important to us.

LEMMA 3.3. *$\Delta F(v_i, v_j)$ is edge-disjoint from $F(v_k)$, for any $k < j < i$.*

Proof. The proof uses the following elementary properties of the lowest common ancestor in an ordered tree: (i) the lowest common ancestor relation is commutative and associative, and (ii) the lowest common ancestor of two nodes x and y is above or equal to the lowest common ancestor of any nodes lying between x and y (in symmetric order). These properties imply that $\alpha(v_i, v_k)$ is above or equal to $\alpha(v_i, v_j)$ in T_a and $\beta(v_i, v_k)$ is above or equal to $\beta(v_i, v_j)$, for any $k < j < i$. Thus, no edge of $F(v_k)$ lies in $\Delta F(v_i, v_j)$. This completes the proof. □

3.3. Data structures for computing tangents on funnels. Our main operation on funnels is finding a common tangent between two funnels. Consider a

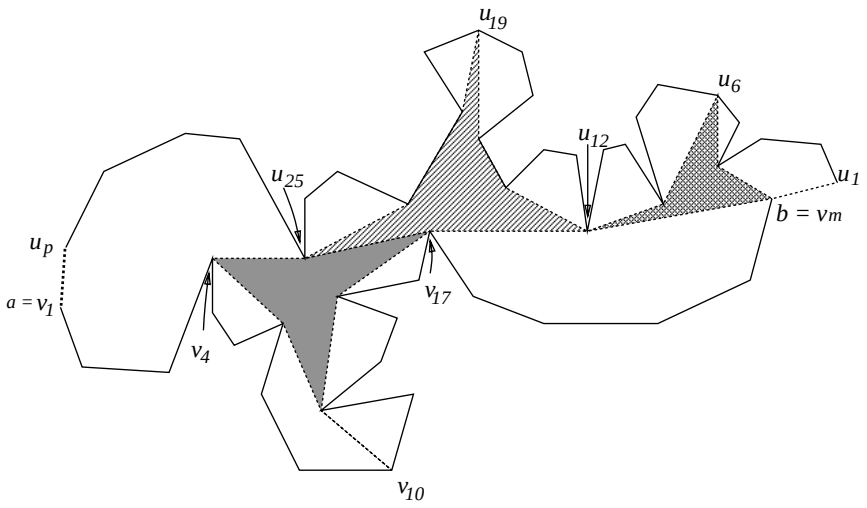


FIG. 3.1. *The example shows funnels for three vertices: v_{10}, u_6, and u_{19}. Lowest common ancestors bounding $F(v_{10})$ are v_4 and v_{17}. The apex of $F(v_{10})$ is v_{12}. Similarly, $F(u_6)$ is bounded by u_6, u_{12}, and $b = v_m$; in this case, u_6 itself is the apex.*

vertex-pair $u \in U$ and $v \in V$. The funnel-pair $(F(u), F(v))$ is *closed* if $\pi(\alpha(u), \beta(u))$ and $\pi(\alpha(v), \beta(v))$ are disjoint except perhaps at endpoints, and *open* otherwise.

An open funnel-pair always admits a unique common tangent. If $(F(u), F(v))$ is an open funnel-pair, then the shortest path $\pi(u, v)$ necessarily uses the tangent between their funnels. We denote the tangent for the funnel-pair $(F(u), F(v))$ by $\ell(u, v)$. The endpoints of $\ell(u, v)$ are denoted u_ℓ and v_ℓ, where $u_\ell \in U$ and $v_\ell \in V$. (Throughout, we adopt the convention that the subscript ℓ will denote the endpoint of a tangent.)

In Figure 3.1, funnels $F(v_{10})$ and $F(u_6)$ form a closed pair, while $F(v_{10})$ and $F(u_{19})$ form an open pair. The tangent between $F(v_{10})$ and $F(u_{19})$ is (u_{22}, v_{15}).

During each execution of *REDUCE*, we maintain funnels $F(u)$ and $F(v)$ for the current vertices u and v. As observed in section 2.3, u moves backward and forward among the active vertices of U, while v moves only forward through the active vertices of V. After each movement of u or v, we compute the tangent for $(F(u), F(v))$, if it is an open funnel-pair.

Let us focus on one of the shortest paths in $F(u)$, say, $\pi(z(u), a)$. We assume that the apex $z(u)$ of $F(u)$ is precomputed; precomputation of all funnel-apexes takes $O(n)$ time altogether using the two shortest-path trees T_a and T_b.

A natural representation of $\pi(z(u), a)$ is a doubly linked list of vertices, but unfortunately this data structure is too inefficient for searching in our application. We speed up the search by breaking the path into subpaths, each represented by a *supernode* that supports fast searches. In this representation, a shortest path is a list of supernodes, linked by *superedges*. In phase k of the algorithm, the subpath within a single supernode has at most 2^k vertices. It is represented by a binary tree of height at most k. The leaves store the subpath vertices. Each internal node represents the subpath of its leaf descendants; it stores the edge linking its left and right subtrees'

paths.

Our tangent-finding algorithm starts its search from two start vertices $u_s \in F(u)$ and $v_s \in F(v)$. The algorithm is a standard binary search [8, 14], with a modified search sequence. Instead of starting from the middle edge of the funnel and then recursively cutting the funnel in half, the search on $F(u)$ walks up from u_s to the root of the supernode binary tree containing u_s, follows the superedges between supernodes, and descends into the supernode binary tree containing the tangent endpoint u_ℓ. The search uses funnel edges in its comparisons, namely, the superedges it follows between supernodes and the edges above u_s and u_ℓ in the supernode binary trees containing those vertices. See Figure 3.2. The following lemma bounds the time of the search.

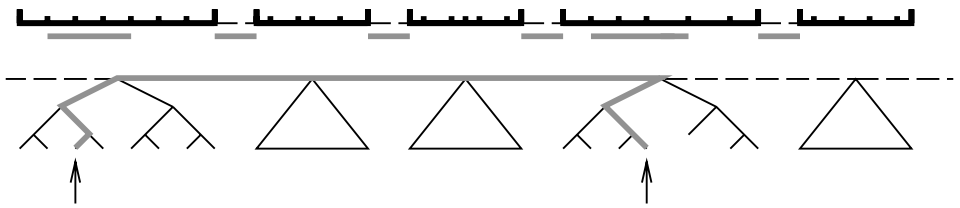


FIG. 3.2. *The supernode representation. The funnel path is shown schematically on top, and its supernode representation on the bottom. The positions of u_s and u_ℓ are shown by arrows, u_s on the left. The search path through the supernode representation is shaded gray, as are the funnel edges examined by the search.*

LEMMA 3.4. *Let $(F(u), F(v))$ be an open funnel-pair with a common tangent $\ell(u, v) = (u_\ell, v_\ell)$, with each funnel represented by two paths of supernodes. Let n_u be the number of supernodes on $F(u)$ between the start vertex u_s and u_ℓ for a particular tangent search; define n_v analogously. If the search occurs during the kth phase of MSEARCH, then computing the tangent $\ell(u, v)$ takes $O(n_u + n_v + k)$ time.*

We need to show how to maintain this representation of $F(u)$ (actually of the path $\pi(z(u), a)$) as u moves among the active vertices of U, and how to initialize the data structure at the start of each phase. To accomplish both tasks, we consider the shortest path tree T_a. In the kth phase we will retain only the vertices of T_a that are ancestors of at least one apex $z(u)$ for some active $u \in U$. This gives a tree with at most n_k leaves, and hence with at most $n_k - 1$ nodes of degree more than two. If we cut T_a below each node of degree greater than two and below each active apex, we get a decomposition of T_a into at most $2n_k - 1$ paths. We partition each of these paths into supernodes, each with at most 2^k vertices. Linking the paths back together gives a shortest path tree of supernodes. We will ensure that the total number of supernodes in the tree is $O(kn_k) = O(kn2^{-k})$.

Given this tree of supernodes and the path of supernodes representing $\pi(z(u), a)$, it is relatively easy to produce the path representing $\pi(z(u'), a)$ for u' an active neighbor of u in U. We compute $\alpha(u, u')$ and map it to the supernode X containing it. The descendant of X is Y, a supernode ancestor of u but not of u'. One of the (at most two) immediate siblings of Y in the shortest path tree is Z, a supernode ancestor of u' but not of u. We unlink X from Y, and link X to Z instead. (If u' is an ancestor or descendant of u, the work required is different but still easy.) All this takes only constant time.

In phase 0 the shortest-path tree of supernodes is just the original tree T_a: all

vertices of U are active and all supernodes contain exactly one vertex. To produce the tree of supernodes for phase $k+1$, given the tree for phase k, we delete dead vertices and combine neighboring supernodes. More precisely, we identify the supernodes that contain active apexes in phase $k+1$ and delete all supernodes that are not (improper) ancestors of at least one such supernode. We cut the tree below every supernode of degree greater than two and every supernode containing an active apex. This gives a decomposition into at most $2n_{k+1} - 1$ paths. On each path with at least two supernodes, we merge adjacent pairs of supernodes. "Merging" two supernodes means that we build a new binary tree whose left and right subtrees are the old supernode trees and whose root contains the superedge linking the old supernodes. It is not hard to show by induction on k that the number of supernodes is at most $(\frac{k+1}{2})n_k$ for $k > 0$. We summarize our data structure's properties in the following lemma.

LEMMA 3.5. *During phase k, the shortest-path tree of supernodes contains $O(kn_k)$ supernodes. The overhead to prepare the data structure for phase k is $O(kn_k)$. Given the supernode representation of $F(u)$, we can modify it to obtain the representation of $F(u')$ for a neighboring active vertex u' in constant time.*

Remark. The bound on the number of supernodes during the kth phase can be improved to $O(n_k)$, at the expense of increasing the tree height to $2k$. The $O(kn_k)$ bound results from merging each supernode tree with at most one other at each phase; this allows the tree height to remain at most k in phase k. By performing two rounds of merging per phase, we can ensure that the number of supernodes is $O(n_k)$, but the two rounds of merging may increase the tree height to $2k$. We use one round of merging because the inferior supernode bound of $O(kn_k)$ is sufficient for our purpose.

3.4. Distance computation. In this section, we show how the primitives of the previous two subsections are used to perform a matrix evaluation. Suppose we want to compute the distance $d(u, v)$, where $u \in U$ and $v \in V$. We start by computing the four lowest common ancestors $\alpha(u), \beta(u), \alpha(v), \beta(v)$. By performing three ancestor queries, we can determine the exact order of these four vertices along the path $\pi(a, b)$. The funnel-pair $(F(u), F(v))$ is closed if and only if the order is either $(\alpha(u), \beta(u), \alpha(v), \beta(v))$ or $(\alpha(v), \beta(v), \alpha(u), \beta(u))$. Otherwise, the pair is an open funnel-pair. Since each ancestor query takes $O(1)$ time, the time to determine whether the funnel-pair is open or not is only a constant. The following two lemmas detail how to compute the distance $d(u, v)$, depending upon whether the funnel-pair is open or closed.

LEMMA 3.6. *If the funnel-pair $(F(u), F(v))$ is closed, $d(u, v)$ can be computed in $O(1)$ time.*

Proof. Without loss of generality, assume that the lowest common ancestor vertices $\alpha(v), \beta(v)$ precede $\alpha(u), \beta(u)$ on $\pi(a, b)$. See Figure 3.1 for an illustration; the funnel-pair $(F(u_6), F(v_{10}))$ is closed. Then it is easy to see that

$$d(u, v) = d(u, \alpha(u)) + d(\alpha(u), \beta(v)) + d(v, \beta(v)). \tag{3.3}$$

These distances can be deduced from shortest-path trees T_a and T_b in constant time each. For instance, $d(u, \alpha(u)) = d(a, u) - d(a, \alpha(u))$, and both the distances on the right-hand side are precomputed in T_a. This completes the proof. □

LEMMA 3.7. *If the funnel-pair $(F(u), F(v))$ is open, $d(u, v)$ can be computed in $O(\tau(u, v))$ time, where $\tau(u, v)$ is the time to compute the tangent $\ell(u, v)$.*

Proof. We compute the tangent $\ell(u, v)$ for the funnel-pair $(F(u), F(v))$. Refer to Figure 3.1, where the funnel-pair $(F(u_{19}), F(v_{10}))$ is open. Then it is easy to see that

$$d(u, v) = d(u, u_\ell) + d(u_\ell, v_\ell) + d(v_\ell, v). \tag{3.4}$$

The distance $d(u_\ell, v_\ell)$ is the Euclidean distance between u_ℓ and v_ℓ. The remaining two terms are deduced from the shortest path trees T_a and T_b in constant time. For example, consider the distance $d(u, u_\ell)$. If u_ℓ is an ancestor of u in T_a, then $d(u, u_\ell) = d(a, u) - d(a, u_\ell)$; otherwise u_ℓ is an ancestor of u in T_b, and $d(u, u_\ell) = d(b, u) - d(b, u_\ell)$. In either case, the distances to a and b are precomputed and accessible in constant time apiece. □

The preceding two lemmas have established that the dominant term in matrix operations is the cost of finding tangents between open funnel-pairs.The next two sections contain the main result of our paper. Section 4 presents our algorithm for computing tangents, and section 5 analyzes the algorithm, showing that it takes $O(k)$ amortized time to compute a tangent in the kth phase of the matrix-searching.

4. The algorithm. We describe how to perform the matrix operations during one phase of the subroutine *REDUCE*. Suppose that we are in the kth phase of the algorithm: the submatrix A_k has size $n_k \times 2n_k$, where $n_k \leq n2^{-k}$. Matrix-searching theory tells us that *REDUCE* performs at most $3n_k$ matrix evaluations. We label the matrix evaluations in time-order, where time runs in integer steps from 1 to $3n_k$. By convention, the matrix entry being evaluated at time t is $d(u(t), v(t))$, where $1 \leq t \leq 3n_k$ and $u(t) \in U$ and $v(t) \in V$ are active vertices. The preceding section shows that the dominant cost in computing $d(u(t), v(t))$ is finding the tangent $\ell(u(t), v(t))$. Therefore, in the remainder of this section, we focus on the tangent-finding component of the matrix operations.

We use the shorthand notation $\ell(t)$ to denote the tangent computed at time t:

$$\ell(t) \;=\; \ell(u(t), v(t)).$$

The vertices $u_\ell(t)$ and $v_\ell(t)$, as before, denote the endpoints of $\ell(t)$. Our main subroutine for implementing *REDUCE* is called *ADVANCE*; it computes the tangent $\ell(t+1) = \ell(u(t+1), v(t+1))$ and advances the current time from t to $t+1$. We maintain the following two auxiliary data fields to support the operations in the subroutine at the current time t:

1. $\ell(t)$: This is the tangent computed at time t; that is, $\ell(t) = \ell(u(t), v(t))$. We store $u_\ell(t)$ and $v_\ell(t)$, the two endpoints of $\ell(t)$, at vertex $v(t)$.
2. $\tilde{\ell}(u)$: This is the most recent tangent computed at the vertex u prior to the current time t; the time t is implicit in this definition. Each vertex $u \in U$ maintains its $\tilde{\ell}(u)$, which is initially set to *NIL*.

The pseudocode for *ADVANCE* finds the tangent $\ell(t+1)$ between the funnels $F(u(t+1))$ and $F(v(t+1))$. The algorithm has two nearly identical parts, one executed when the V-vertex advances, and the other when the U-vertex advances. The two parts differ in one important detail: the choice of the vertex where the search for new tangent begins. When the V-vertex advances (Step 1), the search starts at the lowest common ancestor $\alpha(v(t), v(t+1))$ in V, and at the previous tangent endpoint $u_\ell(t)$ in U. However, when the U-vertex advances (Step 2), a more involved choice of the start vertex is needed. We introduce these vertices, $u^*(t+1)$ and $v^*(t+1)$, in the following; see Figure 4.1.

Let t' denote the time when $\tilde{\ell}(u(t+1))$ was computed; it is undefined if $\tilde{\ell}(u(t+1))$ is *NIL*. Thus $\ell(t') = \tilde{\ell}(u(t+1))$. The two endpoints of $\ell(t')$ are denoted $u_\ell(t')$ and $v_\ell(t')$. Let $u_\alpha = \alpha(u(t), u(t+1))$. Then the start vertex $u^*(t+1)$ is defined as follows.

$$u^*(t+1) \;=\; \begin{cases} u_\alpha & \text{if } t' \text{ is undefined or if } u_\alpha \text{ is a descendant of } u_\ell(t') \text{ in } T_a, \\ u_\ell(t') & \text{otherwise.} \end{cases}$$

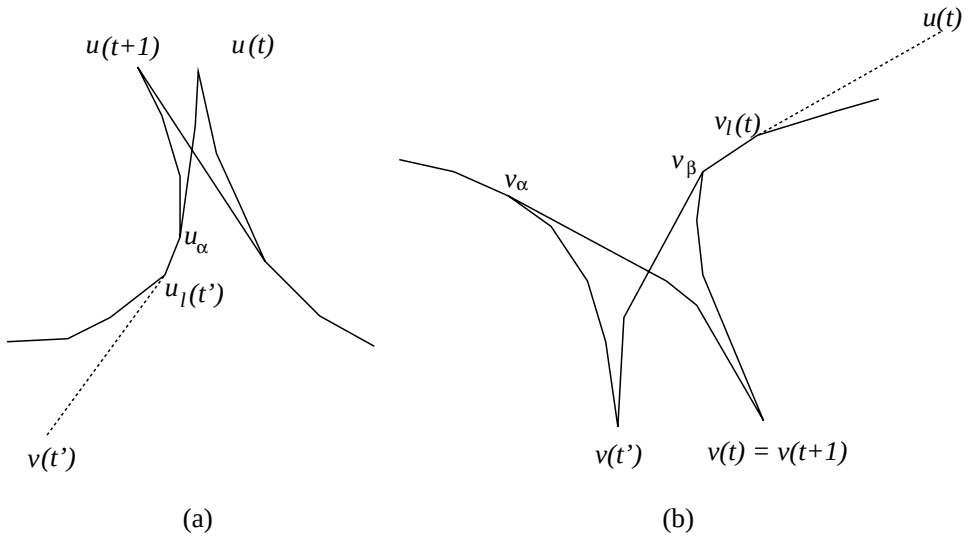


FIG. 4.1. *The definition of the start vertices* $u^*(t+1)$ *and* $v^*(t+1)$. *In* (a), $u^*(t+1) = u_\alpha$, *and in* (b), $v^*(t+1) = v_\beta$.

We observe that $u^*(t+1)$ is a vertex in the funnel-difference $\Delta F(u(t+1), u(t))$.

Next, we define $v^*(t+1)$, which is a vertex in the funnel-difference $\Delta F(v(t), v(t'))$; recall that $v(t')$ is the vertex on the V-chain that was current at time t'. We define v_α and v_β as the vertices bounding the funnel-difference $\Delta F(v(t), v(t'))$:

$$v_\alpha \;=\; \alpha(v(t), v(t')), \qquad v_\beta \;=\; \beta(v(t), v(t')).$$

Finally, recall that $v_\ell(t)$ is the V-chain endpoint of the tangent $\ell(t)$. The start vertex $v^*(t+1)$ is defined as follows.

$$v^*(t+1) \;=\; \begin{cases} v_\alpha & \text{if } t' \text{ is defined and } v_\alpha \text{ is a descendant of } v_\ell(t) \text{ in } T_a, \\ v_\beta & \text{if } t' \text{ is defined and } v_\beta \text{ is a descendant of } v_\ell(t) \text{ in } T_b, \\ v_\ell(t) & \text{otherwise.} \end{cases}$$

With these definitions, we are ready to present the pseudocode for *ADVANCE*. The pseudocode talks about "painting" edges, which is merely a device used by our analysis to prove the linearity of the algorithm. To perform the pseudocode test whether $\ell(t+1)$ is the same as a previously computed tangent ℓ', the algorithm checks whether ℓ' is still tangent to $F(u(t+1))$ and $F(v(t+1))$ at its endpoints. This takes constant time using lowest common ancestor queries.

ADVANCE(t)

(∗ *By matrix-searching theory, either* $v(t+1) \neq v(t)$ *or* $u(t+1) \neq u(t)$, *but not both.* ∗)

1. **if** $v(t+1) \neq v(t)$ **then**
   (a) **if** $\ell(t+1) = \ell(t)$ **then**
   Set $\ell(t+1) = \ell(t)$;
   (b) **else**
   Simultaneously search for $v_\ell(t+1)$ on $F(v(t+1))$ starting from

$\alpha(v(t), v(t+1))$, and for $u_\ell(t+1)$ on $F(u(t+1))$ starting from $u_\ell(t)$.
Paint the edges of $F(u(t+1))$ between $u_\ell(t)$ and $u_\ell(t+1)$;

2. **else** ($* u(t+1) \neq u(t) *$)
   (a) **if** $\ell(t+1) \in \{\ell(t), \tilde{\ell}(u(t+1))\}$ **then**
   Set $\ell(t+1)$ accordingly;
   (b) **else**
   Simultaneously search for $v_\ell(t+1)$ on $F(v(t+1))$ starting from $v^*(t+1)$, and for $u_\ell(t+1)$ on $F(u(t+1))$ starting from $u^*(t+1)$.
   Paint the edges of $F(v(t+1))$ between $v^*(t+1)$ and $v_\ell(t+1)$, and paint the edges of $F(u(t+1))$ between $u^*(t+1)$ and $u_\ell(t+1)$;

end *ADVANCE*

ADVANCE correctly computes the new tangent $\ell(t+1)$: Steps 1a and 2a set $\ell(t+1)$ to a previously computed tangent if that tangent is still valid at time $t+1$. The remaining steps search for the tangent endpoints on funnels $F(u(t+1))$ and $F(v(t+1))$. Since $u_\ell(t+1) \in F(u(t+1))$ and $v_\ell(t+1) \in F(v(t+1))$, these searches are also correct. (The choice of the vertices where the search starts is relevant only for analyzing the running time.)

5. The analysis. This section establishes the linearity of our matrix-searching algorithm. In the kth phase of *MSEARCH*, the submatrix input to *REDUCE* has size $n_k \times 2n_k$, where $n_k \leq n2^{-k}$, and the total number of matrix evaluations performed by *REDUCE* is $O(n_k)$. The discussion in subsection 3.4 shows that the only part of a matrix operation requiring nonconstant work is the tangent computation. Thus, the running time of *ADVANCE* dominates the running time of *REDUCE* in a phase.

5.1. The analysis of *ADVANCE*. The running time of *ADVANCE* is dominated by the searches performed in the tangent-finding steps. The remaining steps take only constant time apiece per step: the tests of tangent equivalence in Steps 1a and 2a can be made in constant time, and the start vertices $u^*(t+1)$ and $v^*(t+1)$ can also be computed in constant time.

We bound the cost of computing tangents in *ADVANCE* during the kth phase by arguing that each funnel edge is searched over at most twice. Using the terminology of section 3.3, if u_s, v_s are the start vertices for the tangent search, then the edges of $F(u)$ between u_s and u_ℓ and the edges of $F(v)$ between v_s and v_ℓ are considered to be *searched over* when $\ell(u, v)$ is computed. (The key point is that the actual cost of computing the tangent is $O(n_u + n_v + k)$, where $n_u + n_v - 2$ is the number of superedges searched over during this tangent-finding operation; see Lemma 3.4.)

Except for the search on the V-chain in Step 1b, all other searches in *ADVANCE* are accounted for by a "painting" paradigm: we paint the edges that are searched over during a tangent computation, and each edge is painted at most once (Lemmas 5.3 and 5.4). We use a separate accounting scheme (Lemma 5.1) to count the edges of V searched over in Step 1b. Together, these lemmas prove that no edge of either U or V is searched over more than twice during a complete phase of *REDUCE*.

LEMMA 5.1. *During a phase of REDUCE, each edge in the V-chain is searched over in Step* 1b *of ADVANCE at most once.*

Proof. Step 1 is executed when $u(t+1) = u(t)$, and $v(t+1) \neq v(t)$. The search in Step 1b is carried out if and only if $\ell(t+1) \neq \ell(t)$. These two facts together

imply that $v_\ell(t+1)$ necessarily lies in the funnel-difference $\Delta F(v(t+1), v(t))$. Only edges of the funnel-difference $\Delta F(v(t+1), v(t))$ are searched over. By Lemma 3.3 the funnel-differences are disjoint at any two distinct times: $\Delta F(v(t'+1), v(t')) \cap \Delta F(v(t''+1), v(t'')) = \emptyset$ for any $0 < t' < t'' < 3n_k$. This completes the proof. □

The proofs of the two remaining lemmas (Lemmas 5.3 and 5.4) are more complicated, and they rely on the following technical lemma.

LEMMA 5.2. *Whenever edges of the V-chain are painted at time $t+1$, their extensions hit the U-chain between $u(t)$ and $u(t+1)$. Similarly, whenever edges of the U-chain are painted at time $t+1$, their extensions hit the V-chain before $v(t+1)$. If the edges of the U-chain are painted because $v(t+1) \neq v(t)$, then their extensions hit the V-chain between $v(t)$ and $v(t+1)$.*

Proof. The proof of the lemma depends on the fact that the painted edges lie on a funnel, and the edges of a funnel have either monotonically decreasing or monotonically increasing slopes from one end to the other. The edges of V painted at time $t+1$ are in $F(v(t+1))$, and they lie between $v^*(t+1)$ and $v_\ell(t+1)$. Because $v_\ell(t+1)$ belongs to $\Delta F(v(t+1), v(t'))$, $v^*(t+1)$ lies on $F(v(t+1))$ between $v_\ell(t)$ and $v_\ell(t+1)$. By the slope property, the slopes of all the painted edges are between the slopes of $\ell(t)$ and $\ell(t+1)$. By the regular expression of section 2.3, $u(t)$ and $u(t+1)$ are two adjacent active vertices in U; notice that $u(t+1)$ can be either the successor or the predecessor of $u(t)$, but the two vertices must be adjacent. Because the extensions of all the painted edges lie between $\pi(v(t), u(t))$ and $\pi(v(t), u(t+1))$, they must hit the U-chain between $u(t)$ and $u(t+1)$. Figure 5.1(a) illustrates this case: the edges painted are shown with heavy lines.

The edges of U can be painted either in Step 1b or in Step 2b. If the edges are painted in Step 1b, then $v(t+1) \neq v(t)$, but $u(t) = u(t+1)$. In this case, the painted edges lie between $u_\ell(t)$ and $u_\ell(t+1)$, the U-endpoints of the tangents at times t and $t+1$. Again, by the slope property, all painted edges have slopes in the range determined by the slopes of $\ell(t)$ and $\ell(t+1)$. Since $v(t)$ and $v(t+1)$ are adjacent active vertices of V, the extensions of all painted edges hit the V-chain between $v(t)$ and $v(t+1)$.

Finally, if the U-edges are painted in Step 2b, then $v(t) = v(t+1)$, but $u(t) \neq u(t+1)$. In this case, the painted edges lie between $u^*(t+1)$ and $u_\ell(t+1)$. Since the matrix-searching algorithm never backs up on the V-chain, the tangent endpoint on $F(u(t+1))$ moves monotonically clockwise with time. In particular, if the old tangent $\tilde{\ell}(u(t+1))$ is not *NIL*, it lies on the shortest path to some vertex $v(t')$ that is before $v(t+1)$ in the V-chain order. Because $u_\ell(t+1)$ belongs to $\Delta F(u(t+1), u(t))$, $u^*(t+1)$ lies on $F(u(t+1))$ between $u_\ell(t')$ and $u_\ell(t+1)$. If $\tilde{\ell}(u(t+1))$ is *NIL*, then $u^*(t+1) = \alpha(u(t), u(t+1))$ is counterclockwise of $u_\ell(t+1)$. In either case, the slope property ensures that the extensions of the U-edges painted in Step 2b hit the V-chain before $v(t+1)$. Figure 5.1(b) illustrates this case: the edges painted are shown with heavy lines. This completes the proof. □

The next two lemmas show that no edge of U or V is painted more than once in a phase. Both lemmas rely on the following observations about Step 2b of *ADVANCE*, in which $v(t+1) = v(t)$ and $u(t+1) \neq u(t)$, and the tangent $\ell(t+1)$ equals neither $\ell(t)$ nor $\tilde{\ell}(u(t+1))$. Let t' denote the time when $\tilde{\ell}(u(t+1))$ is computed, assuming it is not *NIL*; that is, t' is the most recent time before t when a tangent is computed for $F(u(t+1))$. We observe the following vertex-relations:

$$u(t') = u(t+1), \qquad u(t'+1) = u(t), \qquad v(t') = v(t'+1). \tag{5.1}$$

The first relation follows from the definition of t'. The third follows from the first: at

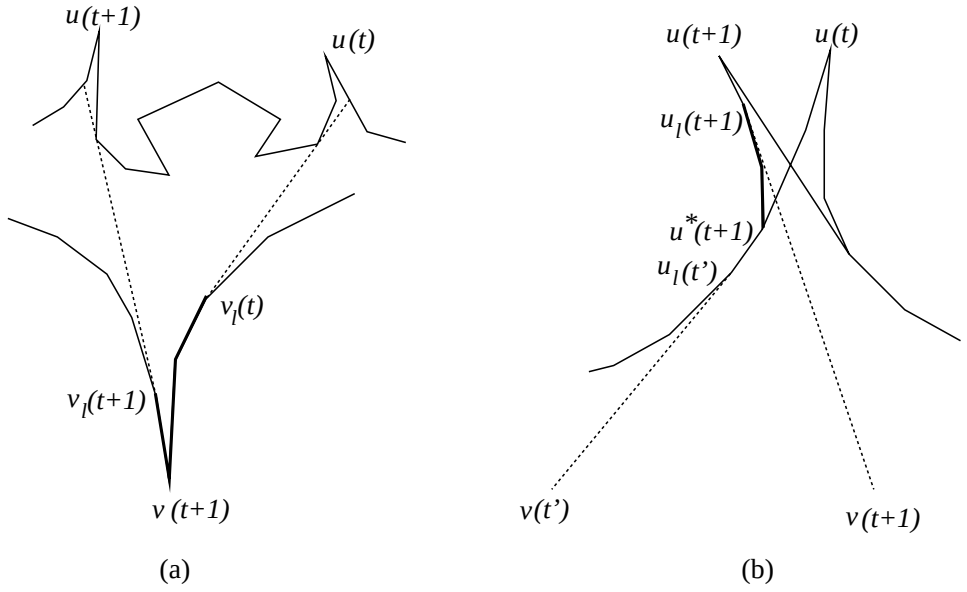


FIG. 5.1. *Illustration for the proof of Lemma* 5.2. *Part* (a) *shows the painting of* V*-edges; a symmetric case arises when* $u(t)$ *is after* $u(t+1)$. *Part* (b) *shows the painting of* U*-edges (in Step* 2b*); in this example, the start vertex is* $u^*(t+1) = \alpha(u(t), u(t+1))$, *and* t' *is the time when the old tangent* $\tilde{\ell}(u(t+1))$ *was computed.*

time $t'+1$, the algorithm must have advanced the U-vertex from $u(t')$ to $u(t'+1)$, keeping the V-vertex the same, since assuming otherwise contradicts the choice of time t'. Finally, the second relation follows from the important fact that *ADVANCE* visits the vertices of U in chain order (either forward or backward), never skipping an active vertex. The direction of movement in the U-chain at time t is opposite to the direction at time t'. A key consequence of the vertex relations in Eq. (5.1) is that between times t' and t, the algorithm visits vertices that are all separated from $u(t+1)$ by $u(t)$. That is, $u(t)$ separates $u(t+1)$ from $u(t'')$, for $t'+1 < t'' < t$.

LEMMA 5.3. *An edge of the* V*-chain is painted at most once per phase.*

Proof. The edges of V are painted only in Step 2b. Our analysis makes crucial use of the funnel difference $\Delta F(v(t), v(t'))$. Since $\ell(t+1) \neq \ell(t')$, it follows that $v_\ell(t+1) \in \Delta F(v(t), v(t'))$. We break up our analysis in two cases, depending upon whether or not $v_\ell(t) \in \Delta F(v(t), v(t'))$.

Case 1. $v_\ell(t) \in \Delta F(v(t), v(t'))$, or t' is undefined. See Figure 5.2.

In this case, we observe that $v^*(t+1) = v_\ell(t)$. Since $v_\ell(t+1)$ lies in the funnel-difference $\Delta F(v(t), v(t'))$, all edges of V painted at time $t+1$ belong to $\Delta F(v(t), v(t'))$. By Lemma 5.2, the extensions of the newly painted edges hit the U-chain between $u(t)$ and $u(t+1)$. If t' is undefined, no painted edge's extension has hit here before, so the newly painted edges have never been painted before. If t' is defined, the most recent time *ADVANCE* traversed this portion of U was at time $t'+1$, when the U-vertex moved from $u(t')$ to $u(t'+1)$. Since the V-funnel at time $t'+1$, namely, $F(v(t'))$, is edge-disjoint from $\Delta F(v(t), v(t'))$, the newly painted edges were not painted at time step $t'+1$. Because $u(t)$ separates $u(t+1)$ from $u(t'')$ for any $t'+1 < t'' < t$, it follows

from Lemma 5.2 that the edges were not painted between time $t'+1$ and time t. By Lemma 3.3, these edges were not painted at any time $t'' \leq t'$, since $\Delta F(v(t), v(t'))$ is disjoint from $F(v(t''))$. Thus, the edges of V painted at time $t+1$ have never been painted before.

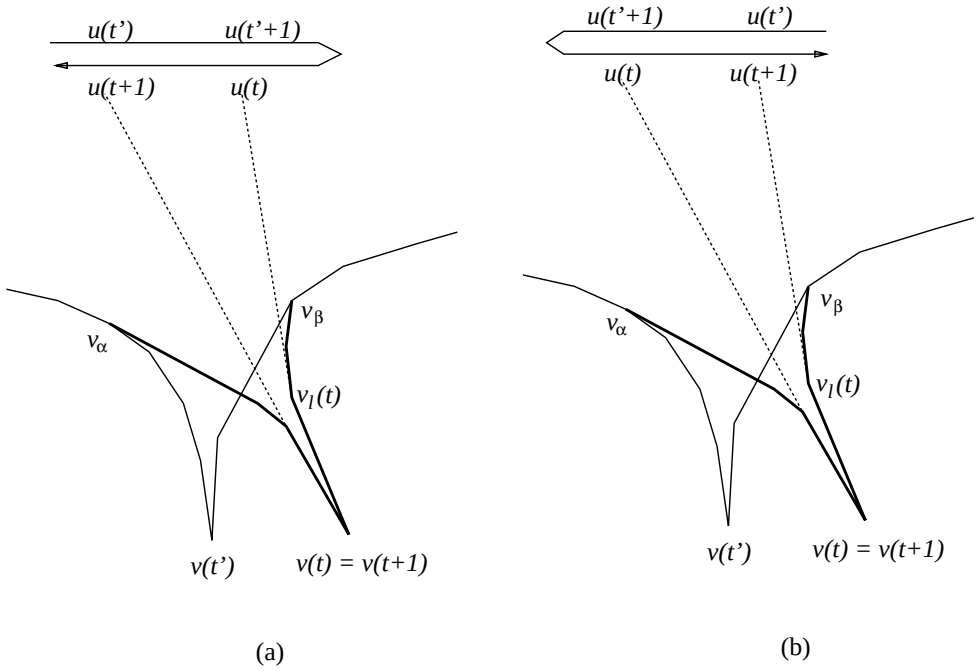


FIG. 5.2. *Illustration for Case* 1 *of Lemma* 5.3. *Parts* (a) *and* (b) *show the two symmetric cases: in* (a)*, the algorithm is moving forward on the U-chain at time t, and in* (b)*, it is moving backward at time t. The motion is suggested by the arrow on top. The funnel-difference* $\Delta F(v(t), v(t'))$ *is shown in heavy lines.*

Case 2. $v_\ell(t) \notin \Delta F(v(t), v(t'))$. See Figure 5.3.

Because $v_\ell(t) \in F(v(t'))$, we have $v_\ell(t) = v_\ell(t'+1)$. Furthermore, $v^*(t+1)$ equals either v_α or v_β, where $v_\alpha = \alpha(v(t), v(t'))$ and $v_\beta = \beta(v(t), v(t'))$. As noted above, $u(t)$ separates $u(t+1)$ from $u(t'')$, for $t' < t'' < t$. Since the V-vertex is confined between $v(t')$ and $v(t)$ during this time interval, we can be sure that tangent endpoints on V lie outside the funnel-difference $\Delta F(v(t), v(t'))$ between times t' and t. In particular, if $v^*(t+1) = v_\beta$, then $v_\ell(t'')$ lies at or above $v_\ell(t)$ in T_b, and if $v^*(t+1) = v_\alpha$, then $v_\ell(t'')$ lies at or above $v_\ell(t)$ in T_a, for any $t' < t'' < t$. Thus, any edges painted on V between times $t'+1$ and t are disjoint from $\Delta F(v(t), v(t'))$. Finally, no edge of $\Delta F(v(t), v(t'))$ is painted at or before t', since Lemma 3.3 ensures that $\Delta F(v(t), v(t'))$ is disjoint from $F(v(t''))$, for any $t'' \leq t'$. This completes the proof. □

LEMMA 5.4. *An edge of the U-chain is painted at most once per phase.*

Proof. The edges of U are painted either in Step 1b or in Step 2b. If the edges are painted in Step 1b, then $v(t+1) \neq v(t)$ and the painted edges lie between $u_\ell(t)$ and $u_\ell(t+1)$. By Lemma 5.2, the extensions of the newly painted edges hit the V chain between $v(t)$ and $v(t+1)$. Because $v(t)$ separates $v(t+1)$ from $v(t')$, for any $t' < t$, the newly painted edges cannot have been painted at an earlier time.

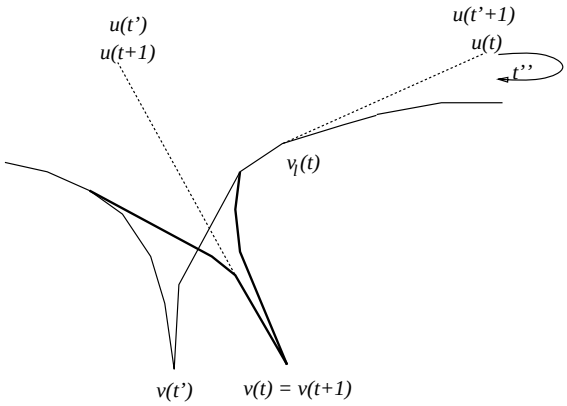


FIG. 5.3. *Illustration for Case* 2 *of Lemma* 5.3. *The symmetric case, in which the algorithm is moving backward on the* U*-chain at time* t*, is not shown.*

If the edges are painted in Step 2b, then $u(t+1) \neq u(t)$, and the painted edges lie between the start vertex $u^*(t+1)$ and $u_\ell(t+1)$. Let t' denote the time when $\tilde{\ell}(u(t+1))$ is computed (possibly undefined). Since $u_\ell(t+1)$ equals neither $u_\ell(t)$ nor $u_\ell(t')$, we know that $u_\ell(t+1)$ lies in the funnel-difference $\Delta F(u(t+1), u(t))$. Since the start vertex $u^*(t+1)$ is guaranteed to be in $\Delta F(u(t+1), u(t))$, all the newly painted edges belong to $\Delta F(u(t+1), u(t))$. If t' is undefined, then by Lemma 3.3 no edge of $\Delta F(u(t+1), u(t))$ has appeared in $F(u(t''))$, for $t'' \leq t$, so none has been painted before. If t' is defined, the edges lying between $u_\ell(t')$ and $u_\ell(t+1)$ have not been painted at or before time t', since their extensions hit the V-chain after $v(t')$ (cf. Lemma 5.2). No edge of the funnel-difference $\Delta F(u(t+1), u(t))$ is painted between times $t'+1$ and t, since the lowest common ancestor property implies that $\Delta F(u(t+1), u(t))$ is disjoint from $F(u(t''))$, for any $t' < t'' \leq t$. Thus, none of the newly painted edges has been painted at an earlier time. This completes the proof. □

5.2. The analysis of *REDUCE* and *MFILL*. We show that the kth phase of *MSEARCH* takes $O(kn_k)$ time, where $n_k \leq n2^{-k}$. The analysis of the preceding section has shown that no edge of U or V is searched over more than twice. There are at most $3n_k$ matrix evaluations during *REDUCE* in the kth phase, and by Lemma 3.4 the time for one evaluation is $O(k + n_u + n_v)$, where $n_u + n_v - 2$ is the number of superedges searched over during the evaluation. The total number of superedges and the total overhead cost of maintaining the funnel data structure during the kth phase is $O((k+1)n_k)$. Thus, the total time spent on matrix evaluations in the kth phase of *REDUCE* is $O((k+1)n_k)$.

The running time of *MFILL* has the same bound. Section 2.3 shows that the matrix evaluations in *MFILL* satisfy the same regular expression as those in *REDUCE*. The same analysis applies, therefore, proving that the running time of *MFILL* at the kth phase is also $O((k+1)n_k)$.

5.3. The time complexity of *MSEARCH*. Finally, we are ready to show that our matrix-searching algorithm runs in $O(n)$ time overall. The algorithm consists of $O(\log n)$ phases, and the matrix in the kth phase has size at most $n_k \times 2n_k$,

where $n_k \leq n2^{-k}$. In each phase, the running time is dominated by the subroutines *REDUCE* and *MFILL*; other steps take $O(n_k)$ time. The cost of executing *REDUCE* and *MFILL* in the kth phase is $O((k+1)n_k)$. Thus, the total cost of the matrix-searching algorithm *MSEARCH* is

$$T(n) = O\left(\sum_{k=0}^{\lceil \log n \rceil} (k+1)n_k\right) = O\left(\sum_{k=0}^{\infty} k\frac{n}{2^k}\right) = O(n).$$

We have established our main theorem.

THEOREM 5.5. *Let P be a simple polygon with n vertices, and let U and V be subchains of its boundary that together partition the vertices of P. Let M denote a $p \times m$ matrix with $M(i,j)$ equal to the shortest path distance between the vertices $u_i \in U$ and $v_j \in V$, where $p = |U|$ and $m = |V|$. Then we can compute the maximum entry in each row of M in $O(n)$ total time.*

5.4. Computing row-wise minima. The algorithm of section 4 also works for computing row-wise minima of M. In fact, by changing the ordering of columns and negating all distances, we can directly apply our maxima-finding algorithm to compute the minima. Let us define an auxiliary $p \times m$ matrix M', where

$$M'(i,j) \;=\; -M(i, m-j+1). \tag{5.2}$$

Notice that the column-ordering of M' is reverse of the column-ordering of M. The triangle inequality again implies that M' is totally monotone.

LEMMA 5.6. *The matrix M' defined in Eq.* (5.2) *is totally monotone.*

We can find the row-wise maxima of M' (which correspond to the row-wise minima in M) using the matrix-searching algorithm of section 4. The direction of travel in the V-chain is now reversed, due to the reversal of columns. To account for this change, we set $a = v_m$ and $b = v_1$. The algorithm and its analysis remain the same. We have the following theorem.

THEOREM 5.7. *Let P be a simple polygon with n vertices, and let U and V be subchains of its boundary that together partition the vertices of P. Let M denote a $p \times m$ matrix with $M(i,j)$ equal to the shortest path distance between the vertices $u_i \in U$ and $v_j \in V$, where $p = |U|$ and $m = |V|$. Then we can compute the minimum entry in each row of M in $O(n)$ total time.*

6. Applications. In this section, we give several applications of Theorems 5.5 and 5.7. Our applications concern problems in simple polygons, where the shortest-path metric is the most natural distance function. In addition, in several problems, we can use the shortest-path metric to mask visibility constraints. Throughout this section, P denotes a simple polygon, whose vertices are labeled $p_1, p_2, \ldots, p_n$ in counterclockwise order around the boundary.

6.1. The geodesic diameter and farthest neighbors. This section describes a linear-time algorithm for computing a farthest neighbor for each vertex of P under the shortest-path metric; the previous best time bound for this problem was $O(n \log n)$ [6, 16]. The *geodesic diameter* of P, denoted $D(P)$, is the maximum length of a shortest path in P:

$$D(P) \;=\; \max\{d(x,y) \mid x, y \in P\}.$$

The diameter is always realized by two vertices of P [16], and so finding it is a special case of the problem of computing farthest neighbors for all vertices of P. By

computing a farthest neighbor of a vertex $p \in P$, we mean two things: identifying a vertex $q \in P$ such that $d(p,q) = \max\{d(p,p') \mid p' \in P\}$, and computing the distance $d(p,q)$. We use a lemma from [16] to decompose the all-farthest neighbor problem into at most three instances of maxima-finding in totally monotone matrices.

Let p_i, p_j and p_k be three vertices of P such that p_j is a farthest neighbor of p_i, and p_k is a farthest neighbor of p_j: $d(p_i,p_j) = \max\{d(p_i,p_l) \mid p_l \in P\}$, and $d(p_j,p_k) = \max\{d(p_j,p_l) \mid p_l \in P\}$. Without loss of generality, assume that p_i, p_j, p_k are in counterclockwise order. The triple p_i, p_j, p_k divides the polygon into three disjoint chains: $P_1 = (p_{i+1}, p_{i+2}, \ldots, p_{j-1})$, $P_2 = (p_{j+1}, p_{j+2}, \ldots, p_{k-1})$, and $P_3 = (p_{k+1}, p_{k+2}, \ldots, p_{i-1})$. Corresponding to these chains are the three complementary chains: $Q_1 = (p_j, p_{j+1}, \ldots, p_i)$, $Q_2 = (p_k, p_{k+1}, \ldots, p_j)$, and $Q_3 = (p_i, p_{i+1}, \ldots, p_k)$. Figure 6.1 shows a schematic diagram of this construction. The following lemma is established in [16].

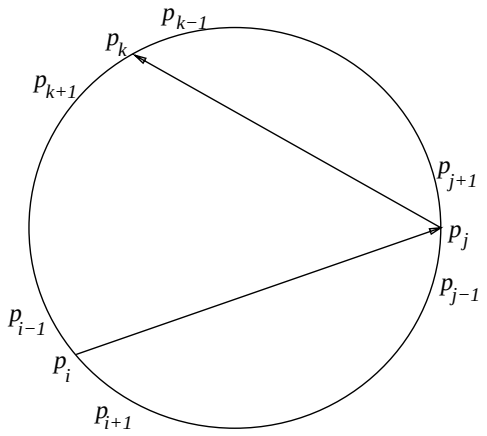


FIG. 6.1. *A schematic diagram showing chains P_i, for $i = 1, 2, 3$, in Lemma 6.1.*

LEMMA 6.1 (see [16]). *Every vertex of the chain P_i has a farthest neighbor in chain Q_i, for $i = 1, 2, 3$.*

Since the farthest neighbors p_j and p_k can be computed in linear time, using shortest path trees [7, 11], Lemma 6.1 gives a linear-time reduction from the farthest-neighbors problem of P to three instances of the farthest-neighbors problem between disjoint chains. The chains P_i, for $i = 1, 2, 3$, cover all vertices of P except p_i, p_j and p_k. Since the farthest neighbors of p_i and p_j are already computed, only p_k remains, and we can compute its farthest neighbor in linear additional time. The farthest-neighbors problem for the chains P_i and Q_i falls within the framework of section 4 and is therefore equivalent to the problem of computing row-wise maxima in a totally monotone matrix. By Theorem 5.5, this can be done in $O(n)$ time, and we have the following theorem.

THEOREM 6.2. *Given a simple polygon P on n vertices, we can compute a geodesic farthest neighbor for each of its vertices in $O(n)$ total time. Consequently, the geodesic diameter of P can also be computed in linear time.*

6.2. The external farthest neighbors. Agarwal et al. [1] gave an $O(n \log n)$ time algorithm for computing an external farthest neighbor for all vertices of a simple polygon. (The external farthest neighbors are computed using the *external* shortest

path metric: the distance between two points is the length of a shortest path in the plane minus the interior of the polygon.) The bottleneck in their algorithm is the computation of *internal* farthest neighbors; the (internal) farthest neighbors algorithm is needed to compute the external farthest neighbors within a "pocket" of the polygon. The remaining steps of the external farthest neighbors algorithm take only $O(n)$ time. Thus, the following result is a direct corollary of Theorem 6.2.

THEOREM 6.3. *Given a simple polygon P on n vertices, we can compute a geodesic external farthest neighbor for each of its vertices in $O(n)$ time. Consequently, the geodesic external diameter of P can also be computed in linear time.*

6.3. Weighted nearest neighbors and Euclidean matching. An internal diagonal of P divides the polygon into two chains. Let U and V be two polygonal chains obtained in this way, where $p = |U|$ and $m = |V|$. Let $w : P \to \Re$ be a real-valued weight function on the vertices of P, and define a $p \times m$ matrix M of weighted distances:

$$M(i,j) \;=\; d(u_i, v_j) - w(u_i) - w(v_j); \tag{6.1}$$

It is easily seen that M is totally monotone, by the triangle inequality. The problem of computing a *weighted nearest neighbor* for each vertex of U in V is equivalent to computing row-wise minima of M. This problem arises in an algorithm of Marcotte and Suri [13] for solving the Euclidean matching problem for points on a polygon. That algorithm finds an optimal matching for the vertices of a simple polygon, under the constraint that each matching edge lies inside the polygon. The algorithm is based on divide-and-conquer, and the dominating term in the conquer step is the complexity of solving the weighted nearest-neighbors problem as defined above. Our algorithm of section 4 solves the row-minima problem for M in linear time, thus improving the result in [13] from $O(n \log^2 n)$ to $O(n \log n)$.

THEOREM 6.4. *The algorithm of Marcotte and Suri* [13] *for computing an optimal matching of the vertices of a simple polygon can be implemented in $O(n \log n)$ time, where n denotes the number of vertices of the polygon.*

6.4. Closest visible vertex pair between two chains. Given two disjoint polygonal chains U and V, the closest visible vertex problem for U and V is to find a closest pair of vertices $u \in U$ and $v \in V$ that are visible to each other. The visibility constraint implies that some vertex-pairs are not legal candidates, and thus the associated distance matrix is not fully defined. Aggarwal et al. [3] gave a complicated linear-time algorithm for this problem that exploits the special structure of the legal entries in the matrix. A linear-time algorithm due to Amato [4] corrects minor problems with the algorithm of Aggarwal et al. Amato's algorithm can also be implemented on a Concurrent Read Concurrent Write Parallel Random Access Machine (CREW PRAM) in time $O(\log n)$ using $O(n)$ processors. However, like the algorithm of Aggarwal et al., Amato's algorithm is quite complicated. In this section, we show that a simpler and considerably more direct linear-time algorithm follows from our paradigm of matrix-searching using the shortest path metric.

Let M denote the $p \times m$ matrix whose (i, j) entry denotes the shortest path distance between u_i and v_j:

$$M(i,j) = d(u_i, v_j).$$

The triangle inequality implies that M is totally monotone, and so by Theorem 5.7, we can compute its row-minima in $O(n)$ time. Suppose that $M(i^*, j^*)$ is the overall

TABLE 7.1

Problem	Previous result	New result
Geodesic diameter	$O(n \log n)$	$O(n)$
All farthest neighbors	$O(n \log n)$	$O(n)$
External farthest neighbors	$O(n \log n)$	$O(n)$
Euclidean matching	$O(n \log^2 n)$	$O(n \log n)$
2-chain closest visible pair	$O(n)$	$O(n)$

minimum entry of M; clearly, this entry is also minimum in row i^*. We claim that the pair (u_{i^*}, v_{j^*}) is a closest visible pair in $U \times V$. By assumption, the distance between u_{i^*} and v_{j^*} is minimum over all u, v pairs, where $u \in U$ and $v \in V$; we need to show only that u_{i^*} and v_{j^*} are mutually visible. We assume otherwise and arrive at a contradiction. If u_{i^*} and v_{j^*} are not visible, then the shortest path $\pi(u_{i^*}, v_{j^*})$ has at least one intermediate vertex, say, z. If $z \in U$, then $d(z, v_{j^*}) < d(u_{i^*}, v_{j^*})$, and if $z \in V$, then $d(u_{i^*}, z) < d(u_{i^*}, v_{j^*})$. In either case, the minimality of $M(i^*, j^*) = d(u_{i^*}, v_{j^*})$ is contradicted. Thus we have the following theorem.

THEOREM 6.5. *Let U and V be two vertex-disjoint chains that partition the vertices of a simple polygon P. Then a closest visible vertex pair of U and V can be found in linear time.*

Theorem 6.5 applies equally well to computing the closest visible pair of vertices in two polygons. In linear time, we can easily transform the problem to two chains U and V that satisfy the conditions of the theorem.

7. Closing remarks. We have presented a linear-time algorithm for computing the row-wise maxima or minima of a totally monotone matrix whose entries correspond to shortest path distances between two chains of a simple polygon. The result appears to be quite powerful and has several applications in computational geometry. The most direct applications concern distance-related problems in simple polygons, since the shortest path metric is the natural metric for nonconvex simple polygons. However, as Theorem 6.5 demonstrates, the result also has applications when the distances are Euclidean but there is an additional visibility constraint. The generalization to the shortest path metric helps mask the visibility constraint. Table 7.1 summarizes the main applications of our result.

Using some additional techniques in combination with the matrix-searching result of this paper, we have also obtained a linear-time algorithm for computing a shortest diagonal of a simple polygon [12].

REFERENCES

[1] P. K. AGARWAL, A. AGGARWAL, B. ARONOV, S. R. KOSARAJU, B. SCHIEBER, AND S. SURI, *Computing external-furthest neighbors for a simple polygon*, Discrete Appl. Math., 31 (1991), pp. 97–111.

[2] A. AGGARWAL, M. KLAWE, S. MORAN, P. SHOR, AND R. WILBER, *Geometric applications of a matrix searching algorithm*, Algorithmica, 2 (1987), pp. 195–208.

[3] A. AGGARWAL, S. MORAN, P. SHOR, AND S. SURI, *Computing the minimum visible vertex distance between two polygons*, in Proc. of 1st Workshop on Algorithms and Data Structures, Springer-Verlag, New York, 1989, pp. 115–134.

[4] N. M. AMATO, *Finding a closest visible vertex pair between two polygons*, Algorithmica, 14 (1995), pp. 183–201.

[5] B. CHAZELLE, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.

[6] L. Guibas and J. Hershberger, *Optimal shortest path queries in a simple polygon*, J. Comput. System Sci., 39 (1989), pp. 126–152. Special issue of selected papers from the 3rd Annual ACM Symposium on Computational Geometry, 1987.

[7] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan, *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, 2 (1987), pp. 209–233.

[8] L. Guibas, J. Hershberger, and J. Snoeyink, *Compact interval trees: A data structure for convex hulls*, Internat. J. Comput. Geom. Appl., 1 (1991), pp. 1–22.

[9] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[10] J. Hershberger, *A new data structure for shortest path queries in a simple polygon*, Inform. Process. Lett., 38 (1991), pp. 231–235.

[11] J. Hershberger and J. Snoeyink, *Computing minimum length paths of a given homotopy class*, Comput. Geom., 4 (1994), pp. 63–97.

[12] J. Hershberger and S. Suri, *Finding a shortest diagonal of a simple polygon in linear time*, Comput. Geom., 7 (1997), pp. 149–204.

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

[14] M. Overmars and J. van Leeuwen, *Maintenance of configurations in the plane*, J. Comput. System Sci., 23 (1981), pp. 166–204.

[15] B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[16] S. Suri, *Computing geodesic furthest neighbors in simple polygons*, J. Comput. System Sci., 39 (1989), pp. 220–235.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1635–1655, December 1997

004

RANDOMIZED $\widetilde{O}(M(|V|))$ ALGORITHMS FOR PROBLEMS IN MATCHING THEORY*

JOSEPH CHERIYAN†

Abstract. A randomized (Las Vegas) algorithm is given for finding the Gallai–Edmonds decomposition of a graph. Let n denote the number of vertices, and let $M(n)$ denote the number of arithmetic operations for multiplying two $n \times n$ matrices. The sequential running time (i.e., number of bit operations) is within a poly-logarithmic factor of $M(n)$. The parallel complexity is $O((\log n)^2)$ parallel time using a number of processors within a poly-logarithmic factor of $M(n)$. The same complexity bounds suffice for solving several other problems:

(i) finding a minimum vertex cover in a bipartite graph,
(ii) finding a minimum $X \rightarrow Y$ vertex separator in a directed graph, where X and Y are specified sets of vertices,
(iii) finding the allowed edges (i.e., edges that occur in some maximum matching) of a graph, and
(iv) finding the canonical partition of the vertex set of an elementary graph.

The sequential algorithms for problems (i), (ii), and (iv) are Las Vegas, and the algorithm for problem (iii) is Monte Carlo. The new complexity bounds are significantly better than the best previous ones, e.g., using the best value of $M(n)$ currently known, the new sequential running time is $O(n^{2.38})$ versus the previous best $O(n^{2.5}/(\log n))$ or more.

Key words. randomized algorithms, matching theory, Gallai–Edmonds decomposition, allowed edges, canonical partition, bipartite minimum vertex covers, digraph minimum vertex separators

AMS subject classifications. 68R10, 05C85, 05C50, 05C40, 05C70, 90C27

PII. S0097539793256223

1. Introduction.

A *matching* of an undirected, possibly nonbipartite, graph $G = (V, E)$ is a subset E' of the edges such that no two of the edges in E' have a vertex in common. A *perfect matching* is one with cardinality $|V|/2$. Tutte [T 47] gave a good characterization of graphs that have perfect matchings, i.e., he showed that the perfect matching decision problem (deciding whether or not a given graph has a perfect matching) is in NP ∩ co-NP. One of Tutte's innovations was introducing the skew symmetric adjacency matrix $\widetilde{B}$ of the graph G, defined as follows: Associate each edge ij of G with a distinct variable x_{ij}. Then $\widetilde{B} = \widetilde{B}(x_{ij})$ is a $|V| \times |V|$ matrix whose entries are given by

$$\widetilde{B}_{ij} = \begin{cases} x_{ij} & \text{if } i > j \text{ and } ij \in E, \\ -x_{ij} & \text{if } i < j \text{ and } ij \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Tutte observed that G has a perfect matching iff the determinant of $\widetilde{B}(x_{ij})$, $\det(\widetilde{B}(x_{ij}))$, is not identically zero; here, $\det(\widetilde{B}(x_{ij}))$ is a polynomial in the variables x_{ij}. Lovász [Lo 79] used this observation to give an efficient randomized algorithm for the perfect matching decision problem: Choose a prime number $q = |V|^{O(1)}$, and substitute each

*Received by the editors September 6, 1993; accepted for publication (in revised form) October 30, 1995.

http://www.siam.org/journals/sicomp/26-6/25622.html

†Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (jcheriyan@watdragon.uwaterloo.ca). This research was supported by NSERC grant OGP0138432 (NSERC code OGPIN 007), by a University of Waterloo faculty research grant, and by the Lucille and David Packard Fellowship of Éva Tardos.

variable x_{ij} in $\widetilde{B}$ by an independent random number drawn from $\{1, 2, \ldots, q-1\}$. Compute the determinant of the resulting random matrix B over the field of integers modulo q. With high probability (i.e., probability $\geq 1 - 1/\Omega(|V|)$; see Lemma 2.1), $\det(B) \neq 0 \mod q$ iff $\det(\widetilde{B}(x_{ij}))$ is not identically zero iff G has a perfect matching. This algorithm has two especially attractive features: it is simple, solving the decision problem by executing one "matrix operation," and it is efficient, running in sequential time $\widetilde{O}(M(|V|)) = O(|V|^{2.38})$ and in parallel time $O((\log |V|)^2)$ using $\widetilde{O}(M(|V|))$ processors. Here, $M(n)$ denotes the number of arithmetic operations for multiplying two $n \times n$ matrices and is currently known to be $O(n^{2.376})$; see Coppersmith and Winograd [CW 90]. Throughout, the bounds on the sequential running time or on the number of parallel processors are stated for the arithmetic complexity model (uniform-cost RAM or PRAM), but they apply also to the bit complexity model because, for each arithmetic operation, comparison, or data transfer, each operand has $O(\log |V|)$ bits, hence the number of bit operations is at most $O((\log |V|)^2)$ times the number of arithmetic operations; see the last paragraph of section 2.

The problem of *finding* a perfect matching of a graph G in time polynomial in $|V(G)|$ remained open until Edmonds [E 65] gave the first algorithm. Edmonds' algorithm solves the following more general problem: For every graph G, the algorithm finds a matching of maximum cardinality in time $|V(G)|^{O(1)}$. One consequence of the algorithm is a theorem that was discovered independently by Gallai [Ga 64], which is the so-called Gallai–Edmonds theorem. According to this theorem, for every graph G the vertex set can be partitioned into three sets $A(G), C(G), D(G)$ in a unique way such that certain properties hold (see Theorem 3.1). The partition gives much useful information, e.g., the cardinality of a maximum matching, the vertices that are incident to every maximum matching, etc. Several algorithms for constructing the partition are known. Edmonds' matching algorithm implicitly constructs the partition. Lovász (see [Kf 86, section 2]) developed a randomized algorithm for finding the Gallai–Edmonds decomposition that runs in time $\widetilde{O}(|V|\, M(|V|))$; though there are faster algorithms for finding the decomposition, the algorithm of [Kf 86] is interesting for its simplicity.

This paper (see Figure 1) presents a simple and efficient randomized algorithm for finding the Gallai–Edmonds decomposition. Lemma 3.3 shows that, with high probability, the partition $A(G) \cup C(G), D(G)$ for a given graph G can be found by computing a basis for the null space of a random skew symmetric adjacency matrix B, i.e., executing one "matrix operation" on B yields this partition. Obtaining the partition $A(G), C(G), D(G)$ from $A(G) \cup C(G), D(G)$ is trivial. The sequential running time is $\widetilde{O}(M(|V|))$ and the parallel time is $O((\log |V|)^2)$ using $\widetilde{O}(M(|V|))$ processors. Our algorithm is closely related to Lovász's algorithm (in [Kf 86]) for the Gallai–Edmonds decomposition; also, the algorithm uses a technique due to Eberly [E 91]. The algorithm, its proof, and running time analysis are all quite simple. Due to the information provided by the Gallai–Edmonds decomposition, our algorithm can be used to find a minimum cardinality vertex cover of a bipartite graph within the same complexity bounds. The minimum cardinality bipartite vertex cover problem is equivalent to the problem of finding a minimum vertex separator for two given vertex sets X and Y in a directed graph (see Proposition 2.4); hence the directed graph problem can be solved within the same complexity bounds.

An edge of a graph G is called *allowed* if it occurs in at least one maximum cardinality matching. Consider the problem of finding the allowed edges. If G has a perfect matching, then the Gallai–Edmonds decomposition gives *no* information about the al-

lowed edges because the partition $A(G), C(G), D(G)$ is trivial with $A(G) = \emptyset = D(G)$. Rabin and Vazirani [RV 89], in an elegant study of the random skew symmetric adjacency matrix B, observed that, if $\det(B) \neq 0$ and the (i, j) minor of B (i.e., the determinant of the submatrix obtained from B by removing row i and column j) is nonzero, then the edge ij (if present) must be allowed. Moreover, all of the (i, j) minors of B can be computed simultaneously by computing the inverse B^{-1}; the (j, i) entry of B^{-1} equals $(-1)^{i+j}/\det(B)$ times the (i, j) minor of B. Combining Rabin and Vazirani's method with our algorithm for the Gallai–Edmonds decomposition gives a randomized algorithm for finding the allowed edges of arbitrary graphs (see section 3.3); the sequential running time is $\widetilde{O}(M(|V|))$ and the parallel time is $O((\log |V|)^2)$ using $\widetilde{O}(M(|V|))$ processors. We also give a randomized algorithm, with the same complexity bounds for finding the canonical partition of an elementary graph, where a

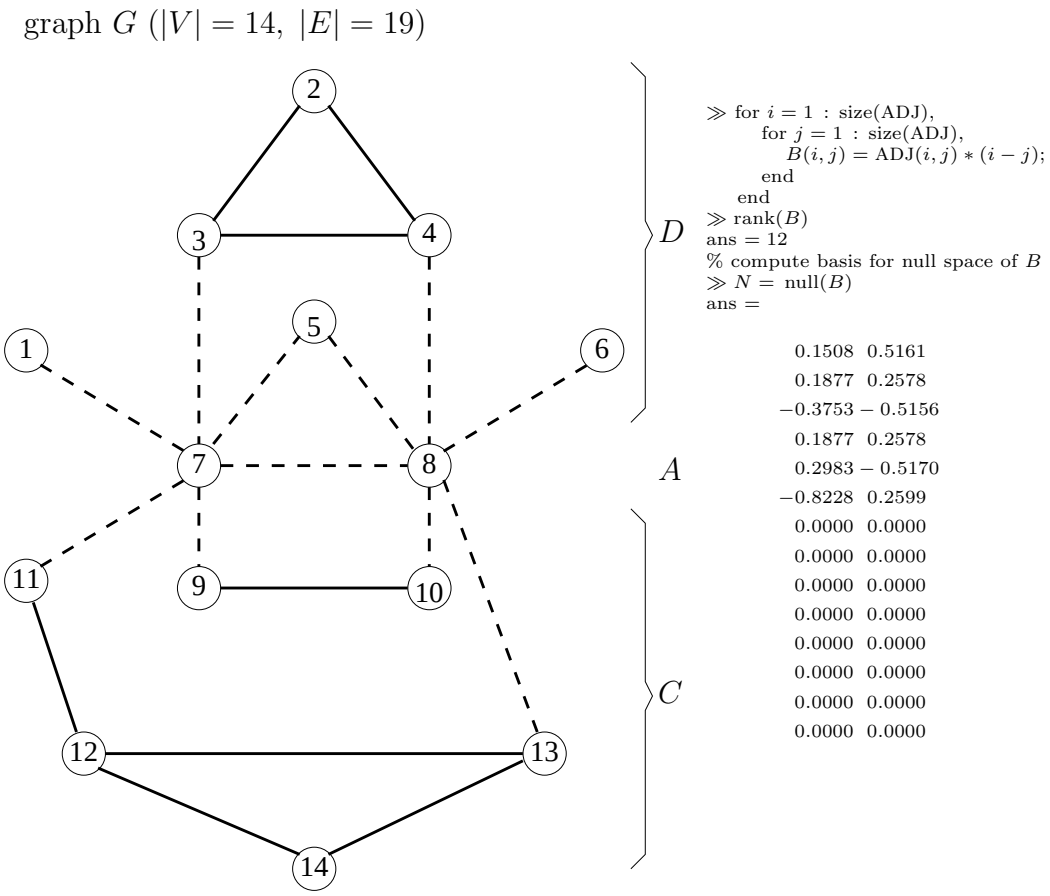


FIG. 1. *Finding the Gallai–Edmonds decomposition of an example graph G, using Lemma* 3.3. *The* MATLAB *code forms a pseudorandom skew symmetric adjacency matrix B from the adjacency matrix* ADJ *of G by substituting $(i - j)$ for each nonzero* ADJ_{ij}. *With high probability, a vertex j is noncritical iff row j of the basis N of the null space of B is nonzero. The resulting partition, $A = \{v_7, v_8\}$, $C = \{v_9, \ldots, v_{14}\}$, $D = \{v_1, \ldots, v_6\}$, is shown. Note that each connected component of D has odd cardinality and each connected component of C has even cardinality. Since* $\text{rank}(B) = 12 = |V| - (\sharp\text{components}(D) - |A|)$, *we have $\nu(G) = 6$. It follows that this partition gives the Gallai–Edmonds decomposition.*

graph is called *elementary* if it has a perfect matching and its allowed edges form a connected spanning subgraph (see section 3.4).

Both Lovász's algorithm for the perfect matching decision problem and our algorithm for the Gallai–Edmonds decomposition are Monte Carlo; however, using results from matching theory, we show how to make both algorithms Las Vegas while achieving the same sequential and parallel complexity bounds. While analyzing our randomized algorithms, we assume that the random bits drawn by the execution have no effect on the sequential or parallel complexity; this assumption may not be appropriate in other contexts. More precisely, for the execution of a randomized algorithm on a *fixed input*, let us take the sequential running time (or parallel running time, or number of parallel processors) to be the maximum sequential running time (or maximum parallel running time, or maximum number of parallel processors) over all possible choices of the random bits. A randomized algorithm is said to be *Monte Carlo* if, for a fixed input, an execution may give incorrect results with small probability. For a randomized algorithm and a problem instance of size n, an event is said to occur with *small* probability if the probability is $\leq 1/\Omega(n)$. A randomized algorithm is said to be *Las Vegas* if, for a fixed input, an execution either returns an output guaranteed to be correct or reports failure, the latter with small probability. A Las Vegas algorithm may be trivially converted into a Monte Carlo algorithm without changing the complexity. To convert a Monte Carlo algorithm into a Las Vegas algorithm, we need a subroutine for verifying whether the output of the Monte Carlo algorithm is correct. If the complexity of the verifying subroutine is bounded by that of the Monte Carlo algorithm, then the Las Vegas and Monte Carlo algorithms have the same order of complexity. This raises a difficulty for our randomized algorithms: We need to verify the correctness of results for problems in matching theory within a complexity bound that is significantly less than that of the best algorithms known for finding a maximum cardinality matching (see the next paragraph). Fortunately, the partition of $V(G)$ computed by our randomized algorithm for the Gallai–Edmonds decomposition can be verified in sequential time $\widetilde{O}(|E| + |V|)$ (or in parallel time $O((\log |V|)^2)$ using $\widetilde{O}(|E| + |V|)$ parallel processors). Consequently, our algorithms for the Gallai–Edmonds decomposition, a minimum vertex cover of a bipartite graph, and a minimum vertex separator of a directed graph all can be made Las Vegas without affecting the complexity. If the graph is bipartite, then our algorithm for finding the allowed edges can be made Las Vegas without affecting the complexity, but for nonbipartite graphs, we do not have a sufficiently efficient subroutine for verifying the allowed edges. Given an elementary graph, there is a sequential $\widetilde{O}(|E| + |V|)$-time algorithm for verifying whether the partition computed by our Monte Carlo algorithm is a canonical partition (see [L 95]), so our sequential algorithm for the canonical partition can be made Las Vegas without affecting the complexity.

We briefly discuss the best sequential and parallel complexities known for computing a maximum cardinality (or a perfect) matching. The fastest known sequential algorithms for finding a maximum matching are due to Micali and Vazirani [MV 80], Blum [B 90], and Gabow and Tarjan [GT 91] (also see [V 94]) and run in time $O(|E|\sqrt{|V|})$ (for dense graphs this is $O(|V|^{2.5})$ time). These algorithms are deterministic, however, and are significantly slower than Lovász's randomized algorithm for the perfect matching decision problem. At present, all efficient (i.e., poly-logarithmic time and polynomial number of processors) parallel algorithms for matching problems use randomization. The best parallel algorithms for finding a maximum matching are the Monte Carlo algorithms of Mulmuley, Vazirani, and Vazirani [MVV 87], Galil and

Pan [GP 88], and Karp, Upfal, and Wigderson [KUW 86]; the parallel complexities are $O((\log|V|)^2)$ time using $\widetilde{O}(|V|\,|E|\,M(|V|))$ processors and $O((\log|V|)^3)$ time using $\widetilde{O}(|V|\,M(|V|))$ processors, respectively. Our parallel complexity bounds are stated for the Exclusive Read Exclusive Write (EREW) PRAM model. Efficient parallel Las Vegas algorithms for matching problems have been designed by Karloff [Kf 86] and Wein [W 91].

It turns out that our algorithm for finding a minimum vertex separator for two given vertex sets X and Y in a directed graph can be developed independently of matching theory; this is done in section 4, building on the work of Linial, Lovász, and Wigderson [LLW 88] and Cheriyan and Reif [CR 94]. Preliminary versions of the results of computing the Gallai–Edmonds decomposition and directed graph $X{\rightarrow}Y$ separators have appeared in [C 94] and [C 93], respectively.

Section 2 contains notation, definitions, and preliminary results. Section 3 develops the algorithms for problems in matching theory. Section 4 is independent of section 3 and develops an algorithm for a minimum $X{\rightarrow}Y$ separator in a directed graph. Finally, Section 5 contains conclusions, and the appendix contains some proofs.

2. Preliminaries. For the given graph $G = (V, E)$, we use n and m to denote the number of vertices and edges, i.e., $n = |V|$ and $m = |E|$. For a subset X of V, $\overline{X}$ denotes $V - X$. For a matrix A with row and column indices from V and two subsets X, Y of V, $A(X, Y)$ denotes the submatrix of A formed by the X-rows and the Y-columns. The vector with a 1 in position j and zeros elsewhere is denoted by e_j, and $\left[\begin{smallmatrix} A \\ e_j \end{smallmatrix}\right]$ denotes the $(n+1) \times n$ matrix formed by adding the $(n+1)$th row e_j to an $n \times n$ matrix A.

A few standard definitions from matching theory are needed; see [LP 86]. An *odd (even) component* of a graph is a connected component whose vertex set has odd (even) cardinality. A *vertex cover* of a graph $G = (V, E)$ is a vertex set $C \subseteq V$ such that each edge is incident with some vertex of C. Given a graph $G = (V, E)$ and a matching E', a vertex is called *matched* if it is incident to an edge of E' and is called *exposed* otherwise. A *near perfect matching* is one with exactly one exposed vertex. For a graph G, $\nu(G)$ denotes the number of edges of a maximum matching. The *deficiency* of G is the number of vertices exposed in a maximum matching $n - 2\nu(G)$. A vertex x is called *noncritical* if it is exposed in at least one maximum matching, otherwise x is called *critical*. Equivalently, x is noncritical if $\nu(G - x) = \nu(G)$ and is critical if $\nu(G - x) < \nu(G)$. A graph H is called *factor critical* if for each of its vertices x, $H - x$ has a perfect matching.

The following lemma due to Zippel [Z 79] and Schwartz [Sc 80] (also see [Ko 91, Corollary 40.2]) is useful for estimating the failure probability of a whole class of randomized algorithms.

LEMMA 2.1 (Zippel–Schwartz). *If $p(x_1, x_2, \ldots, x_m)$ is a nonzero polynomial of degree d with coefficients in a field and S is a subset of the field, then the probability that p evaluates to zero on a random element $(s_1, s_2, \ldots, s_m) \in S^m$ is at most $d/|S|$.*

Recall from section 1 the definition of the skew symmetric adjacency matrix $\widetilde{B} = \widetilde{B}(x_{ij})$ of a graph G and Tutte's observation that $\det(\widetilde{B})$ is not identically zero iff G has a perfect matching. Lovász generalized this observation; for a proof, see [RV 89].

PROPOSITION 2.2 (Lovász). *Let $\widetilde{B} = \widetilde{B}(x_{ij})$ be the skew symmetric adjacency matrix of a graph G. Then* $\operatorname{rank}(\widetilde{B}) = 2\nu(G)$.

A random skew symmetric adjacency matrix B is obtained by substituting the variables x_{ij} in $\widetilde{B}(x_{ij})$ by independent random numbers w_{ij} from a subset $\{1, \ldots, W\}$

of a field. The next result is due to Lovász [Lo 79] and follows from the previous one by applying the Zippel–Schwartz lemma.

PROPOSITION 2.3 (Lovász). *Let* $B = \widetilde{B}(w_{ij})$ *be a random skew symmetric adjacency matrix of a graph* G*, where the* w_{ij} *are independent random numbers from* $\{1, \ldots, W\}$*. Then* $\text{rank}(B) \leq 2\nu(G)$*, and with probability at least* $1 - (n/W)$*,* $\text{rank}(B) = 2\nu(G)$*.*

Given a digraph (directed graph) $G = (V, E)$ and a pair of subsets X and Y of the vertices, an $X \rightarrow Y$ *(vertex) separator* is a set of vertices S such that $G - S$ has no path from a vertex in $X - S$ to a vertex in $Y - S$. For a pair of subsets X and Y of the vertices, $p(X, Y)$ denotes the maximum number of vertex disjoint paths from X to Y (any two of these paths have no vertices in common, not even the terminal vertices). Clearly, every $X \rightarrow Y$ separator has cardinality at least $p(X, Y)$. Menger's theorem states that for every pair of subsets X and Y of the vertices, there exists an $X \rightarrow Y$ separator with cardinality $p(X, Y)$. We call an $X \rightarrow Y$ separator *minimum* if its cardinality is minimum, namely, $p(X, Y)$.

Let us call two problems *linear-time equivalent* if there is a linear-time algorithm to transform an instance of the first problem to an instance of the second such that a solution to the second instance can be transformed in linear time to a solution of the first instance, and vice versa. Part (i) of the next proposition is well known. The novel point of part (ii) is that a digraph minimum vertex separator can be obtained in linear time from an arbitrary minimum vertex cover of an appropriately constructed bipartite graph. See the appendix for a proof of the proposition.

PROPOSITION 2.4.

(i) *The problem of finding a maximum cardinality matching in a bipartite graph is linear-time equivalent to the problem of finding a maximum cardinality set of vertex-disjoint* $X \rightarrow Y$ *paths in a digraph.*

(ii) *The problem of finding a minimum vertex cover in a bipartite graph is linear-time equivalent to the problem of finding a minimum* $X \rightarrow Y$ *separator in a digraph.*

We use the soft-Oh notation to denote the complexity of algorithms. The soft-Oh notation drops poly-logarithmic factors: For functions f and g, f is $\widetilde{O}(g)$ iff there are constants n_0, $k \geq 0$ such that $f(n) \leq g(n)(\log n)^k$, for all $n \geq n_0$. Note that $\widetilde{O}(M(n)) = O(n^{2.38})$, since $M(n)$ is known to be $O(n^{2.376})$ (see section 1). All computations of the algorithms presented below are over the field $\mathbb{Z}_q$ of integers modulo a prime number q. When choosing random numbers w, we assume that they are drawn from the uniform distribution over $\{1, \ldots, W\}$, where W is an integer and $W < q$. Throughout, we take $q = |V(G)|^{O(1)}$; i.e., q is polynomially bounded in the number of vertices of the graph G. Consider the number of bit operations for multiplying two $|V| \times |V|$ matrices over the field of integers modulo q. Since an integer modulo q can be represented using $O(\log q) = O(\log |V|)$ bits and the multiplication of two q-bit numbers takes $O(q^2)$ bit operations, it follows that the number of bit operations is $\widetilde{O}(M(|V|))$.

3. Randomized algorithms for problems in matching theory. This section develops randomized $\widetilde{O}(M(|V|))$-time algorithms for the following problems in matching theory: finding a Gallai–Edmonds decomposition; finding a minimum vertex cover in a bipartite graph; finding the allowed edges of a graph; and finding the canonical partition of an elementary graph.

3.1. A randomized algorithm for the Gallai–Edmonds decomposition. Recall that a vertex x is called *noncritical* if it is exposed in at least one maximum matching, otherwise x is called *critical*. We use $D(G)$ to denote the set of noncritical vertices and $A(G)$ to denote the set of vertices in $V(G) - D(G)$ adjacent to vertices of $D(G)$. The set of remaining vertices, $V(G) - (D(G) \cup A(G))$, is denoted by $C(G)$. For ease of notation, $D(G)$ and $C(G)$ are also used to denote the subgraphs of G induced by the respective vertex sets. See [LP 86, Theorem 3.2.1] for a proof of the next theorem.

THEOREM 3.1 (Gallai–Edmonds). *Let G be a graph, and let $D(G)$, $A(G)$, and $C(G)$ be as defined above. Then*

(i) *each component of the subgraph induced by $C(G)$ has a perfect matching;*
(ii) *each component of the subgraph induced by $D(G)$ is factor critical;*
(iii) *the deficiency of G equals*

$$\sharp\text{components}(D(G)) - |A(G)|,$$

where $\sharp$components$(D(G))$ denotes the number of connected components in the subgraph induced by $D(G)$;
(iv) *every maximum matching of G contains a perfect matching of each component of $C(G)$, a near perfect matching of each component of $D(G)$, and matches all the vertices of $A(G)$ with vertices in distinct components of $D(G)$.*

The key result for our algorithm follows. Recall the notation $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$ from section 2.

LEMMA 3.2. *Let $B = \widetilde{B}(w_{ij})$ be a random skew symmetric adjacency matrix of a graph G, where the w_{ij} are independent random numbers from $\{1, \ldots, W\}$. Consider any vertex x, and let j be its index in B.*

(i) *If x is noncritical, then with probability at least $1 - (2n/W)$ the rank of the matrix $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$ is greater than that of B.*
(ii) *If x is critical, then with probability at least $1 - (n/W)$ the rank of the matrix $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$ equals that of B.*

Proof. Consider the augmented graph G' and its random skew symmetric adjacency matrix B', where G' is obtained from G by adding a new vertex z (with index $n+1$) and the edge xz, and B' is obtained from B by adding a row $r \cdot e_j$ and a corresponding column, where r is a random number independent of the entries of B, i.e.,

$$B' = \begin{bmatrix} B & & & & 0 \\ & & & & \vdots \\ & & & & -r \\ & & & & \vdots \\ 0 & \ldots & r & \ldots & 0 \end{bmatrix}.$$

Consider the cardinality of a maximum matching of G'. If there exists a maximum matching of G with x exposed, then $\nu(G')$ is greater than $\nu(G)$ because the new edge xz of G' may be added to the maximum matching of G. However, if x is matched in every maximum matching of G, then $\nu(G')$ equals $\nu(G)$. In other words, x is noncritical in G iff $\nu(G')$ is greater than $\nu(G)$. Applying Proposition 2.3 to G' and B', we see that if x is noncritical in G, then with probability at least $1 - (2n/W)$,

$$\text{rank}(B') = 2\nu(G') = 2\nu(G) + 2 = \text{rank}(B) + 2.$$

Consider the matrix $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$ obtained from B' by removing the last column and then dividing the last row by r. Since $\text{rank}(\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]) \geq \text{rank}(B') - 1$, part (i) of the lemma follows.

For part (ii) we have seen that $\nu(G')$ equals $\nu(G)$ if x is critical. Hence, with probability at least $1-(n/W)$, $\text{rank}(B) = 2\nu(G) = 2\nu(G') \geq \text{rank}(B') \geq \text{rank}(\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]) \geq \text{rank}(B)$. □

ALGORITHM 1. *Monte Carlo Gallai–Edmonds Decomposition*
INPUT: Graph $G = (V, E)$.
OUTPUT: With high probability, the Gallai–Edmonds decomposition of G.
Step 0:
Order the vertices, and number them $1, 2, \ldots, n$.
Fix the number $W = n^{O(1)}$, and choose a prime q, $W < q = n^{O(1)}$.
For each edge ij, choose a random weight $w(ij) \in \{1, 2, \ldots, W\}$.
Construct a random skew symmetric adjacency matrix B of G,
where for each edge ij, $i > j$,
$B_{ij} = w(ij)$ and $B_{ji} = -w(ij)$ ($B_{ij} = 0$ if ij is not an edge).
Step 1:
Compute the rank r of B over the field $\mathbb{Z}_q$.
Step 2:
For each of the vectors e_j, $j = 1, \ldots, n$,
compute the rank r_j of the matrix $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$ over the field $\mathbb{Z}_q$.
Let D be the set of vertices j with $r_j > r$.
Step 3:
Let A be the subset of $V - D$ adjacent to D, and let C be the set
of vertices neither in D nor in A.
With high probability, the Gallai–Edmonds decomposition of G
is given by A, C, D.

FIG. 2.

The algorithm for finding the Gallai–Edmonds decomposition follows immediately from both the previous lemma and Theorem 3.1. Find the set $D(G)$ of noncritical vertices with high probability by comparing the rank of each $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$, $j = 1, \ldots, n$, with the rank of B. The probability that the set $D(G)$ is correctly computed is at least $1-(2n^2/W)$. Knowing $D(G)$, the sets $A(G)$ and $C(G)$ can be found in $\widetilde{O}(n+m)$ time. See Algorithm 1 in Figure 2 for a full description. The working of the algorithm on an example is illustrated in Figure 1. The following straightforward implementation of Algorithm 1 runs in $\widetilde{O}(n \cdot M(n)) = O(n \cdot n^{2.38})$ time: For each $j = 1, \ldots, n$, use the algorithm of [IMH 82] to find the rank of $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$ in $\widetilde{O}(M(n))$ time. We now improve the running time from $\widetilde{O}(n \cdot M(n))$ to $\widetilde{O}(M(n))$. The $\widetilde{O}(M(n))$ bound holds even for the number of bit operations; to see this, recall the remarks at the end of section 2. To obtain a faster implementation, observe that $\text{rank}(\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right])$ is greater than $\text{rank}(B)$ iff e_j is *not* in the row space of B, i.e., iff e_j is not a linear combination of the row vectors of B. We can "simultaneously" compute the ranks of all the $\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right]$'s by computing a matrix N such that for any row vector v, $v \cdot N = 0$ iff v is a linear combination of the row vectors of B. Once N is computed, we simply find the product of the $n \times n$ identity matrix I_n with N. The nonzero rows of N correspond exactly to the vectors e_j having $\text{rank}(\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right])$ greater than $\text{rank}(B)$. Coming to the computation of N, we take N to be a basis for the null space (i.e., kernel) of B. Note that for any

subspace U (e.g., the row space of B) of a finite-dimensional vector space W over a finite field (e.g., the n-dimensional vector space over $\mathbb{Z}_q$), $\dim U + \dim U^{\perp} = \dim W$, and so $(U^{\perp})^{\perp} = U$ [Lo 93, Exercise 5.31]. Hence, even for the n-dimensional vector space over $\mathbb{Z}_q$ we can check whether a vector v is in the row space of B by checking whether $v \cdot N$ is zero. It is well known that for any $n \times n$ matrix B, a basis for the null space can be computed in sequential time $\widetilde{O}(M(n))$ (see [IMH 82, pp. 53–54]) and in randomized parallel time $O((\log n)^2)$ using $\widetilde{O}(M(n))$ processors (see [KP 91, p. 190]).

LEMMA 3.3. *Let $B = \widetilde{B}(w_{ij})$, $w_{ij} \in \{1, \ldots, W\}$, be a random skew symmetric adjacency matrix of a graph, and let the $n \times (n - \mathrm{rank}(B))$ matrix N be a basis for the null space of B. Let v be an arbitrary vertex, and let j be its index in B. If v is critical (noncritical), then with probability at least $1 - (2n/W)$ the jth row of N is zero (nonzero).*

Let A, C, D denote the partition of V computed by an execution of the Monte Carlo algorithm. To make the algorithm Las Vegas, we need to verify whether A, C, D is the Gallai–Edmonds decomposition. We first verify whether $\nu(G)$ is computed correctly and then verify whether the set D equals the set of noncritical vertices $D(G)$. By Proposition 2.3, $\nu(G)$ is at least $\mathrm{rank}(B)/2$, where B is the random skew symmetric adjacency matrix. Suppose that each component of D is odd and each component of C is even. Then $\nu(G)$ is at most $(|V| - (\sharp\mathrm{components}(D) - |A|))/2$, because every matching E' of G leaves at least $\sharp\mathrm{components}(D) - |A|$ exposed vertices; to see this, observe that for each odd component of $G - A$, either the odd component contains an exposed vertex or an edge of E' matches a vertex of the odd component to a vertex of A. Our verification subroutine (in the Las Vegas algorithm) determines the odd and even components of $G - A$ and compares $\mathrm{rank}(B)$ with $|V| - (\sharp\mathrm{components}(D) - |A|)$. If equality fails to hold in the comparison, or one of the components of D is even, or one of the components of C is odd, then the Las Vegas algorithm reports failure. Otherwise, $\nu(G)$ is guaranteed to equal $\mathrm{rank}(B)/2$, and moreover, the set A is guaranteed to be a barrier. A set $X \subseteq V(G)$ is called a *barrier* if $|V| - 2\nu(G)$ (i.e., the deficiency of G) equals the difference of the number of odd components of $G - X$ and $|X|$. We claim that if A is a barrier and $\nu(G) = \mathrm{rank}(B)/2$, then the computed partition A, C, D is the Gallai–Edmonds decomposition. To see this, note that if a vertex with index j has $\mathrm{rank}(\left[\begin{smallmatrix} B \\ e_j \end{smallmatrix}\right])$ greater than $\mathrm{rank}(B) = 2\nu(G)$, then the vertex is noncritical; hence, every vertex in the computed set D is noncritical. Also, every noncritical vertex is contained in D by the following result (see [LP 86, Theorem 3.3.17]): If $X \subseteq V(G)$ is a barrier, then every noncritical vertex is contained in the union of the odd components of $G - X$. Consequently, $D = D(G)$, and so, by construction, $A = A(G)$ and $C = C(G)$.

THEOREM 3.4. *There is a Las Vegas algorithm with a sequential running time of $\widetilde{O}(M(n))$ for finding the Gallai–Edmonds decomposition and the cardinality of a maximum matching of a graph. A parallel version of the algorithm uses $\widetilde{O}(M(n))$ processors and takes parallel time $O((\log n)^2)$.*

3.2. Finding a minimum vertex cover in a bipartite graph. Due to the information provided by the Gallai–Edmonds decomposition, the above Las Vegas algorithm may be applied to solve other problems in matching theory within the same complexity bounds. In this subsection, we show how the algorithm may be used to find a minimum vertex cover of a bipartite graph. Moreover, by the equivalence of the bipartite minimum vertex cover problem and the digraph minimum $X \rightarrow Y$ separator problem (see Proposition 2.4), we can also find a minimum $X \rightarrow Y$ separator in a digraph. We need a theorem from matching theory; see [LP 86, Theorem 3.2.4].

THEOREM 3.5 (Dulmage and Mendelsohn). *Let $G = (V_1, V_2, E)$ be a bipartite graph, where V_1 and V_2 are the sets of the vertex bipartition. For $i = 1, 2$ let $A_i = A(G) \cap V_i$, $C_i = C(G) \cap V_i$, and $D_i = D(G) \cap V_i$, where $A(G)$, $C(G)$, and $D(G)$ are the three sets of the Gallai–Edmonds decomposition of G. Then $C_1 \cup A_1 \cup A_2$ and $C_2 \cup A_1 \cup A_2$ are minimum vertex covers.*

The above theorem, combined with the Las Vegas algorithm for the Gallai–Edmonds decomposition, immediately yields an efficient Las Vegas algorithm for a minimum vertex cover of a bipartite graph $G = (V_1, V_2, E)$. The algorithm may be simplified by focusing on just one of the sets V_i, $i = 1, 2$, of the vertex bipartition and for each vertex in that set computing whether or not it is critical. Also, instead of using the skew symmetric adjacency matrix we use the bipartite adjacency matrix H, which has a row for each vertex in V_1 and a column for each vertex in V_2; an entry H_{ij} is nonzero iff G has the edge ij, $i \in V_1, j \in V_2$. See Algorithm 2 in Figure 3.

ALGORITHM 2. *Monte Carlo Bipartite Minimum Vertex Cover*
INPUT: Bipartite graph $G = (V_1, V_2, E)$.
OUTPUT: With high probability, a minimum vertex cover of G.
Step 0:
Order the vertices of V_1 and V_2, and number them $1, 2, \ldots$.
Fix the number $W = n^{O(1)}$, and choose a prime q, $W < q = n^{O(1)}$.
For each edge ij, $i \in V_1, j \in V_2$, choose a random weight $w(ij) \in \{1, 2, \ldots, W\}$.
Construct a random bipartite adjacency matrix H of G,
where for each edge ij, $i \in V_1, j \in V_2$, $H_{ij} = w(ij)$ ($H_{ij} = 0$ if ij is not an edge).
Step 1:
Compute the rank r of H over the field $\mathbb{Z}_q$.
Step 2:
For each of the vectors e_j, $j = 1, \ldots, |V_2|$,
compute the rank r_j of the matrix $\left[\begin{smallmatrix} H \\ e_j \end{smallmatrix}\right]$ over the field $\mathbb{Z}_q$.
Let $D_2 \subseteq V_2$ be the set of vertices $j \in V_2$ with $r_j > r$.
Step 3:
Let A_1 be the subset of V_1 adjacent to D_2, i.e.,
$A_1 = \{i \in V_1 : ij \in E \text{ and } j \in D_2\}$.
With high probability, a minimum vertex cover of G is given by
$A_1 \cup (V_2 - D_2)$.

FIG. 3.

THEOREM 3.6. *There is a Las Vegas algorithm with a sequential running time of $\widetilde{O}(M(n))$ for finding a minimum cardinality vertex cover of a bipartite graph. A parallel version of the algorithm uses $\widetilde{O}(M(n))$ processors and takes parallel time $O((\log n)^2)$. The same complexity bounds apply for finding a minimum cardinality $X \rightarrow Y$ separator of a digraph.*

In section 4, an algorithm for finding minimum $X \rightarrow Y$ separators in digraphs (and for finding bipartite minimum vertex covers) is designed using different methods than those used in this section. Yet it turns out that the two algorithms for bipartite minimum vertex covers are identical.

3.3. Finding the allowed edges. Recall from section 1 that an edge of a graph $G = (V, E)$ is called *allowed* if it is contained in at least one maximum matching. The notion of an allowed edge is important in matching theory; see [LP 86, Chapter 5]. We develop a Monte Carlo algorithm for finding the set of allowed edges of an arbitrary

graph; the sequential and parallel complexities are the same as those of our algorithm in Theorem 3.4. The best previous sequential or parallel algorithms for finding the set of allowed edges of a graph take at least as much sequential time (or, parallel time and parallel processors) as needed for computing a maximum matching.

Our method for finding the set of allowed edges first constructs the Gallai–Edmonds decomposition $A(G), C(G), D(G)$ using the algorithm of Theorem 3.4. Now observe that every edge incident to a noncritical vertex v is allowed: First, consider a maximum matching such that v is exposed, and switch the matching by adding any edge vw and removing the matched edge incident to w. Second, every edge with one end vertex in $A(G)$ and the other end vertex in either $A(G)$ or $C(G)$ is *not* allowed, by Theorem 3.1. Finally, we are left with the edges with both end vertices in $C(G)$. Since every component of $C(G)$ has a perfect matching, we apply the following result of Rabin and Vazirani (see [RV 89, Lemma 4]) to find (with high probability) the allowed edges of components of $C(G)$.

LEMMA 3.7 (Rabin and Vazirani). *Let G be a graph with a perfect matching, and let B be a random skew symmetric adjacency matrix of G. If $\det(B) \neq 0$, then for each index i, $1 \leq i \leq n$, there is an index j, $1 \leq j \leq n$, such that $B_{ij} \neq 0$ and $(B^{-1})_{ji} \neq 0$; moreover, for each pair i, j satisfying this condition, the corresponding edge $v_i v_j$ is in some perfect matching of G.*

THEOREM 3.8. *With probability at least $1 - (1/n^{\Theta(1)})$, the set of allowed edges of a graph can be computed in sequential time $\widetilde{O}(M(n))$ and in parallel time $O((\log n)^2)$ using $\widetilde{O}(M(n))$ processors.*

The above algorithm is Monte Carlo but not Las Vegas; if the algorithm reports that an edge with both end vertices in $C(G)$ is not allowed, then there is a small probability that the edge is actually allowed. In all other cases, the algorithm's output is correct. For the special case of bipartite graphs, we give a Las Vegas algorithm that achieves the same complexity bounds. Focus on a component $H = (V_1, V_2, E)$ of $C(G)$, where V_1 and V_2 are the sets of the vertex bipartition. We construct the connected subgraphs $H_1, \ldots, H_k$ of H formed by the computed set of allowed edges. For each connected subgraph H_i, $1 \leq i \leq k$, $|V(H_i) \cap V_1|$ must equal $|V(H_i) \cap V_2|$ and each edge with both end vertices in H_i must be allowed (see [LP 86, Theorem 4.1.1]); otherwise, the algorithm reports failure. Next we construct a bipartite graph H' by contracting to a distinct single vertex each of the two sets in the vertex bipartition of each of the connected subgraphs $H_1, \ldots, H_k$; i.e., each $V(H_i) \cap V_j$, $1 \leq i \leq k$, $j = 1, 2$, is contracted to a distinct vertex. We also replace any parallel edges with single edges. Thus each H_i, $1 \leq i \leq k$ is contracted to a distinct edge; observe that these "contracted edges" form a perfect matching of the contracted graph H'. If the contracted graph H' has a unique perfect matching, then the computed set of allowed edges of H is correct; otherwise, the algorithm reports failure; see [LP 86, Lemma 4.3.1]. To test for a unique perfect matching in H', we start with the perfect matching consisting of the edges formed by contracting $H_1, \ldots, H_k$, and check whether there exists an alternating cycle with respect to this matching. The claimed complexity bounds suffice for testing for an alternating cycle.

THEOREM 3.9. *There is a Las Vegas algorithm with a sequential running time of $\widetilde{O}(M(n))$ for finding the set of allowed edges of a bipartite graph. A parallel version of the algorithm uses $\widetilde{O}(M(n))$ processors and takes parallel time $O((\log n)^2)$.*

3.4. Finding the canonical partition of an elementary graph. Recall from section 1 that a graph $G = (V, E)$ is called elementary if it has a perfect matching and its allowed edges form a connected spanning subgraph. Also recall that a set X

of vertices is called a barrier if the deficiency of G, $|V| - 2\nu(G)$, equals the difference of the number of odd components of $G - X$ and $|X|$. For an elementary graph the deficiency is zero, so $X \subseteq V$ is a barrier iff $|X|$ equals the number of odd components of $G - X$. If G is elementary, then the (inclusionwise) maximal barriers of G form a partition $S_1, S_2, \ldots, S_k$ of the vertex set $V(G)$; this partition is called the *canonical partition* [LP 86, section 5.2]. Here we develop an efficient randomized algorithm for finding the canonical partition of an elementary graph. The Monte Carlo algorithm for the canonical partition was discovered jointly with Padayachee. The sequential $\widetilde{O}(m+n)$-time algorithm for verifying the canonical partition is due to La Poutré [L 95].

The following key result underlies our algorithm (see [LP 86, Theorem 5.2.2]): Two (distinct) vertices x and y are in the same set S_i of the canonical partition iff $G - \{x, y\}$ has *no* perfect matching. Based on this result and [RV 89, Lemma 3], we find the canonical partition as follows: Assume that the given graph G is elementary. We construct a random skew symmetric adjacency matrix B of G. If $\det(B) = 0$, then we stop and report failure. Otherwise we compute the inverse of B, B^{-1}. To compute the canonical partition of $V(G)$, we attempt to construct an equivalence relation Ψ on the vertex pairs such that vertices x and y are related iff the (x, y) entry of B^{-1}, $(B^{-1})_{xy}$, is zero. If Ψ is indeed an equivalence relation, then the algorithm outputs the equivalence classes of Ψ as the computed partition (with high probability, this is the canonical partition); otherwise, if Ψ is not an equivalence relation, then we stop and report failure. Verifying that Ψ is an equivalence relation and computing its equivalence classes takes sequential time $\widetilde{O}(n^2)$ and parallel time $O((\log n)^2)$ using $\widetilde{O}(n^2)$ processors; first, observe that Ψ is reflexive ($(x, x) \in \Psi, \forall x \in V$) and symmetric ($(x, y) \in \Psi$ iff $(y, x) \in \Psi$) since B is skew symmetric and n is even; Ψ is transitive iff each connected component of the graph (V, Ψ) is a clique.

THEOREM 3.10 (with Padayachee). *With probability at least* $1 - (1/n^{\Theta(1)})$, *the canonical partition of an elementary graph can be computed in sequential time* $\widetilde{O}(M(n))$ *and in parallel time* $O((\log n)^2)$ *using* $\widetilde{O}(M(n))$ *processors.*

The above algorithm is Monte Carlo but not Las Vegas. For this paragraph, let $S_1, \ldots, S_k$ denote the partition computed by the Monte Carlo algorithm; the canonical partition may differ from $S_1, \ldots, S_k$. If each set S_i, $1 \leq i \leq k$, is a barrier, then the computed partition is the canonical partition. To see this, observe that for any two vertices x and y in two different sets of the computed partition, the (x, y) entry of B^{-1} is nonzero; hence, by [RV 89, Lemma 3] the graph $G - \{x, y\}$ has a perfect matching. Consequently, by the key result on canonical partitions quoted above x and y must be in different sets of the canonical partition. Hence, the canonical partition is a refinement of the partition $S_1, \ldots, S_k$, and if the two partitions differ, then one of the sets S_i is the union of two or more maximal barriers. To obtain a Las Vegas algorithm we do the following: For each set S_i in the computed partition, we determine whether it is a barrier by comparing the number of odd components of $G - S_i$ with $|S_i|$. If every set S_i is a barrier, then we output $S_1, \ldots, S_k$ as the canonical partition; otherwise we report failure. There is a sufficiently efficient sequential algorithm due to La Poutré [L 95] for the key computation in verifying the partition computed by our Monte Carlo algorithm; this algorithm uses Sleator and Tarjan's [ST 83] dynamic trees data structure to maintain the connected components of the current subgraph and works by appropriately deleting and inserting all edges incident to vertices in the set S_i, $1 \leq i \leq k$; also see La Poutré and Westbrook [LW 94].

THEOREM 3.11 (La Poutré). *Given a graph* $G = (V, E)$ *and a collection of*

pairwise disjoint vertex sets $S_1, \ldots, S_k$, the number of odd components in $G - S_i$ for all i, $1 \le i \le k$, can be determined in (deterministic) sequential time $\widetilde{O}(m+n)$.

Alternatively, the sequential $\widetilde{O}(m+n)$ time bound can be achieved by a randomized Las Vegas algorithm using dynamic data structures recently developed by Henzinger and King [HK 95].

THEOREM 3.12. *Given an elementary graph as input, there is a sequential Las Vegas algorithm with a running time of $\widetilde{O}(M(n))$ for finding the canonical partition.*

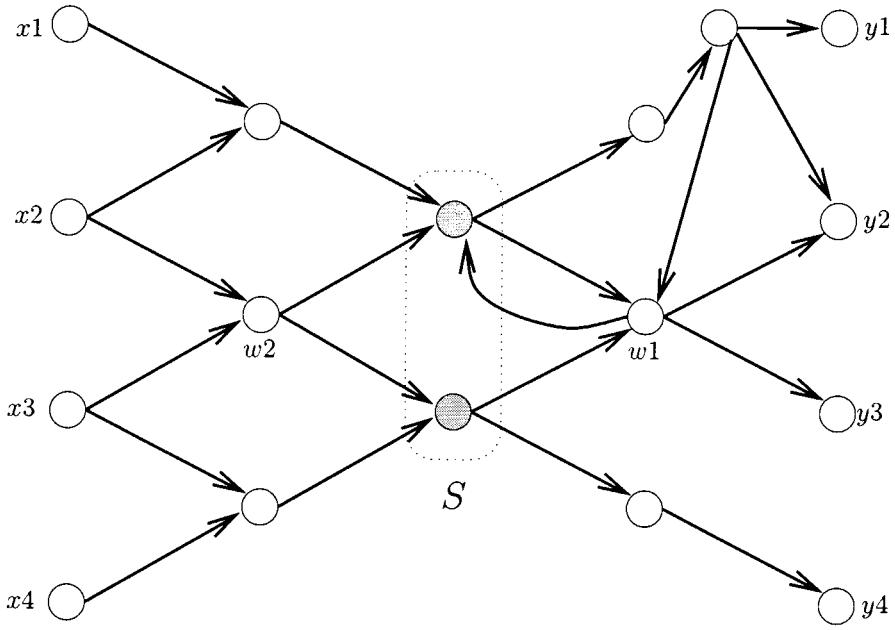


FIG. 4. *A digraph with a unique minimum $X \to Y$ separator S, where $X = \{x_1, x_2, x_3, x_4\}$ and $Y = \{y_1, y_2, y_3, y_4\}$. Vertex w_1 is in $T(S)$ since $p(\{w_1\} \cup X, Y) = 3 > |S|$, but vertex w_2 is not in $T(S)$ since $p(\{w_2\} \cup X, Y) = 2 = |S|$.*

4. An algorithm for digraph minimum $X \to Y$ separators. Using methods different from those employed in the previous section, this section develops a randomized Monte Carlo algorithm for finding a minimum $X \to Y$ separator of a given digraph $G = (V, E)$, where X and Y are specified sets of vertices. In every instance the algorithm outputs a correct solution with high probability, but it may output an incorrect result with small probability. The complexity bounds of this Monte Carlo algorithm suffice for verifying that the computed solution is indeed a minimum $X \to Y$ separator, thus giving a Las Vegas algorithm with the same complexity bounds.

For an $X \to Y$ separator S with $|S| = p(X, Y)$, let $T(S)$ denote the set of vertices such that $G - S$ has a path from each vertex in $T(S)$ to at least one vertex in $Y - S$ (note that $T(S)$ is empty iff $S = Y$). Informally, $T(S)$ forms the "Y-side" of the separator S.

First, consider the simple case when the digraph has a *unique* minimum $X \to Y$ separator S, i.e., any other set of vertices whose removal from G leaves no $X \to Y$ paths has cardinality at least $|S| + 1$. See Figure 4. Then the set $T = T(S)$ has the key property that a vertex v is in T if and only if $p(\{v\} \cup X, Y)$ is greater than $p(X, Y)$. To see this, deduce from Menger's theorem that G must have a separator

S' of cardinality $p(\{v\} \cup X, Y)$ whose removal from G leaves no path from $(\{v\} \cup X)$ to Y. Clearly, S' is also an $X \rightarrow Y$ separator, so $p(\{v\} \cup X, Y) \geq p(X, Y)$. If $v \notin T$, take the separator S' to be S, since $G - S$ has no path from a vertex in $\{v\} \cup X$ to a vertex in Y. Otherwise, if the vertex v is in T, then v is a witness to the fact that $S \neq S'$ and hence, by the uniqueness of S, $|S'| > |S|$. Suppose that there is an efficient method of computing $p(X, Y)$ for any specified pair of sets X and Y of the vertices, i.e., suppose that a fast "black box" subroutine for computing $p(X, Y)$ is available. (Such a method is described below.) Then the separator S may be found as follows. For each vertex $v \in V$, check whether $p(\{v\} \cup X, Y)$ is greater than $p(X, Y)$. Then construct the set T of vertices v that satisfy the inequality. The required separator S consists of the predecessors of T together with the Y-vertices not in T, i.e.,

$$S = \{s \in V - T \mid (s, v) \in E \text{ and } v \in T\} \cup (Y - T).$$

In general, a digraph may have many minimum $X \rightarrow Y$ separators. Fortunately, one of these separators satisfies the key property of the separator S and the vertex set $T(S)$ used above. This is proved in the next lemma. Although a full proof is given, the first part of the lemma is well known.

LEMMA 4.1. *Let $S^\star$ be an $X \rightarrow Y$ separator with cardinality $p(X, Y) = k$ such that $T(S^\star)$ is (inclusionwise) minimal over all $X \rightarrow Y$ separators with cardinality k. Then*

(i) *$S^\star$ is unique, and*

(ii) *for each vertex v of G,*

$$v \in T(S^\star) \qquad \textit{iff} \qquad p(\{v\} \cup X, Y) > k.$$

Proof. For a subset A of G's vertices, define $\Delta(A)$ to be the set of vertices

$$\{u \in V - A \mid (u, v) \in E \text{ and } v \in A\} \cup (Y - A),$$

i.e., $\Delta(A)$ consists of the predecessors of A as well as the Y-vertices not in A. Note that if A is the empty set, then $\Delta(A) = Y$. For every $A \subseteq V$, Y is a subset of $A \cup \Delta(A)$, and moreover, if A is a subset of $V - X$, then note that $\Delta(A)$ is an $X \rightarrow Y$ separator (since every path from a vertex in X to a vertex in $A \cup \Delta(A)$ must contain a vertex in $\Delta(A)$). Let $\delta(A)$ denote the cardinality of $\Delta(A)$. The proof hinges on the fact that the function $\delta : 2^V \longrightarrow \mathbb{Z}$ is submodular, i.e., for any two subsets A and B of G's vertices,

$$\delta(A \cap B) + \delta(A \cup B) \leq \delta(A) + \delta(B). \tag{1}$$

To see this, observe that if a vertex u contributes two to the left-hand side (i.e., $u \in \Delta(A \cap B) \cap \Delta(A \cup B)$), then either $u \in Y - (A \cup B)$ or $u \in V - (A \cup B \cup Y)$ and there is an edge uv, $v \in A \cap B$, so u contributes two to the right-hand side; otherwise, if u contributes one to the left-hand side, then u contributes at least one to the right-hand side.

To prove the first part of the lemma, by way of contradiction, assume that there are two minimum $X \rightarrow Y$ separators S_1 and S_2 such that $T_1 = T(S_1)$ is minimal and $T_2 = T(S_2)$ is minimal. Then note that $T_1 \cap T_2$ is both a proper subset of T_1 and a proper subset of T_2 (possibly, $T_1 \cap T_2$ is the empty set). Consider the vertex sets $T_1 \cap T_2$ and $T_1 \cup T_2$. Neither $T_1 \cap T_2$ nor $T_1 \cup T_2$ has any X-vertices since T_1 has no X-vertices and T_2 has no X-vertices. Hence, $\Delta(T_1 \cap T_2)$ is an $X \rightarrow Y$ separator

and $\Delta(T_1 \cup T_2)$ is an $X \to Y$ separator. Since the minimum cardinality of an $X \to Y$ separator is $p(X,Y) = k$, it is clear that $\delta(T_1 \cap T_2) \geq k$ and $\delta(T_1 \cup T_2) \geq k$. Now, using the submodularity of δ (equation (1)), it follows that $\delta(T_1 \cap T_2) = k$ and $\delta(T_1 \cup T_2) = k$. Let S' denote $\Delta(T_1 \cap T_2)$. Observe that $T(S')$ is a subset of $T_1 \cap T_2$ because, for each vertex $v \notin (T_1 \cap T_2)$, every path from v to a vertex in Y contains a vertex of $\Delta(T_1 \cap T_2)$. This gives the desired contradiction and completes the first part of the lemma, since neither $T_1 = T(S_1)$ nor $T_2 = T(S_2)$ is minimal.

For the second part of the lemma consider any vertex $v \in T(S^\star)$. The maximum number of vertex disjoint paths from $(\{v\} \cup X)$ to Y is either exactly $k = p(X,Y)$ or greater than k. Suppose that the number is k. Then, by Menger's theorem, there exists a separator S' of cardinality k whose removal from G leaves no path from $(\{v\} \cup X)$ to Y. Clearly S' is also a minimum $X \to Y$ separator. Now consider a minimum $X \to Y$ separator S such that $T(S)$ is a subset of $T(S')$ and $T(S)$ is (inclusionwise) minimal over all such separators; since S' exists, S must exist. By the first part of the lemma the separators S and $S^\star$ are the same. This gives the desired contradiction, since $v \in T(S^\star) - T(S)$. We conclude that the maximum number of vertex disjoint paths from $(\{v\} \cup X)$ to Y is greater than k. □

Two results are needed to develop a fast, probabilistic method for computing $p(X,Y)$. The first result is attributed to Ingleton and Piff [IP 73]; for completeness, a proof that follows [LLW 88, Theorem 3.1] is included in the appendix. Associate a variable $x(i,i)$ with each vertex i and a variable $x(i,j)$ with each edge (i,j) (all variables are distinct). The *free adjacency matrix* $\widetilde{F} = \widetilde{F}(x(i,j))$ of G is an $n \times n$ matrix whose entries are given by

$$\widetilde{F}_{ij} = \begin{cases} x(i,i) & \text{if } i = j, \\ x(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

THEOREM 4.2 (Ingleton and Piff). *Let G be a digraph, and let $\widetilde{F}$ be its free adjacency matrix. Then for any k-vertex set X and any k-vertex set Y,*

$$p(X,Y) = k \quad \textit{iff} \quad \det \widetilde{F}(\overline{Y}, \overline{X}) \textit{ is not identically zero.}$$

We also need a matrix identity of Jacobi (see [BR 91, Lemma 9.2.10]).

FACT (Jacobi). *If a matrix F is nonsingular, then a square submatrix $F(\overline{Y}, \overline{X})$ is nonsingular iff the complementary submatrix $F^{-1}(X,Y)$ is nonsingular. More precisely,*

$$\det(F^{-1}(X,Y)) = \det(F(\overline{Y}, \overline{X}))/\det(F).$$

To apply the above theorem to the algorithm, the variables are substituted by random values. This is motivated by the Zippel–Schwartz lemma (Lemma 2.1).

THEOREM 4.3. *Let $G = (V,E)$ be a digraph, and let $F = \widetilde{F}(w(i,j))$ be obtained from G's free adjacency matrix by randomly and independently assigning each variable $x(i,j)$ a random number $w(i,j)$ from $\{1, \ldots, W\}$. Then with probability at least $1 - (n/W)$, F is nonsingular. If F is nonsingular, then for every pair of sets X and Y of the vertices,*

$$p(X,Y) \geq \operatorname{rank}(F^{-1}(X,Y)),$$

and with probability at least $1 - (n^2/W)$, $p(X,Y)$ equals $\operatorname{rank}(F^{-1}(X,Y))$.

Proof. View the determinant of the free adjacency matrix $\widetilde{F}$ as a polynomial of degree n in the variables $x(i,j)$, $1 \le i,j \le n$, and notice that it is not identically zero because the diagonal term $\Pi_{i=1}^{n} x(i,i)$ is nonzero and no two nonzero terms cancel out. Hence by Lemma 2.1, F is nonsingular with probability at least $1-(n/W)$.

Consider a maximum-cardinality set of vertex disjoint paths from X to Y. Let A be the set of start vertices of these $X \rightarrow Y$ paths, and let B be the set of end vertices. Obviously, $A \subseteq X$, $B \subseteq Y$, and $|A| = |B| = p(X,Y) = p(A,B)$. Let $\widetilde{H} = \widetilde{H}(x(i,j))$ denote $(\det \widetilde{F})\widetilde{F}^{-1}$, i.e., $\widetilde{H}$ is the $n \times n$ matrix whose (k,ℓ) entry is $(-1)^{k+\ell}$ times the (ℓ,k) minor of $\widetilde{F}(x(i,j))$; every entry of $\widetilde{H}$ is a polynomial of degree $n-1$ in the variables $x(i,j)$. By Jacobi's identity and Theorem 4.2, $\det \widetilde{H}(A,B)$ is not identically zero, while for every integer $q > p(X,Y)$ the determinant of every $q \times q$ submatrix of $\widetilde{H}(X,Y)$ is identically zero. The second part of the theorem follows by observing that if F is nonsingular, then $F^{-1} = \widetilde{F}(w(i,j))^{-1} = \widetilde{H}(w(i,j))/\det \widetilde{F}(w(i,j))$; now apply Lemma 2.1 to $\det \widetilde{H}(A,B)$. □

The algorithm can now be sketched. See Algorithm 3 in Figure 5. Fix a number $W = n^{O(1)}$, and let q be a prime such that $W < q = n^{O(1)}$. All computations are over the field $\mathbb{Z}_q$ of integers modulo q. The matrix F is constructed, and with high probability it is nonsingular. Inverting F gives the matrix F^{-1}. If $r = \text{rank}(F^{-1}(X,Y))$ equals $|Y|$, then by Theorem 4.3 $p(X,Y)$ equals $|Y|$; therefore Y is a minimum $X \rightarrow Y$ separator. Otherwise consider the unique minimum $X \rightarrow Y$ separator $S^\star$ with $T(S^\star)$ minimal. The algorithm attempts to compute the vertex set $T(S^\star)$ by finding the set T of vertices v such that $\text{rank}(F^{-1}(\{v\} \cup X, Y))$ is greater than r. With probability at least $1 - \Theta(n^3)/W$, T equals $T(S^\star)$. Hence, with high probability, the set $S = \Delta(T)$ (i.e., the set of the predecessors of T and the Y-vertices not in T) is the minimum $X \rightarrow Y$ separator $S^\star$.

To efficiently compute for each vertex v whether $\text{rank}(F^{-1}(\{v\} \cup X, Y)) > r$, the algorithm needs to check that v's row vector $F^{-1}(\{v\}, Y)$ is not a linear combination of the row vectors of $F^{-1}(X,Y)$. As in section 3, we "simultaneously" compute the ranks of all the matrices $F^{-1}(\{v\} \cup X, Y)$, $v \in V$, by computing a matrix N such that for any row vector w, $w \cdot N = 0$ iff w is a linear combination of the row vectors of $F^{-1}(X,Y)$. The matrix N is easily obtained by computing a basis for the null space of $F^{-1}(X,Y)$. Once N is computed, we simply find the product of the matrix $F^{-1}(V,Y)$ with N.

To check that the computed set S is indeed a minimum $X \rightarrow Y$ separator, observe that the cardinality of every $X \rightarrow Y$ separator is at least $p(X,Y) \ge r$. Consequently, if the removal of S from G leaves no path from $X - S$ to $Y - S$, and $|S| = r$, then $|S| = p(X,Y) = r$ and hence S is a minimum $X \rightarrow Y$ separator. Also, by using Proposition 2.4 this algorithm may be applied to find a minimum vertex cover in a bipartite graph.

Consider the sequential complexity of the above algorithm. Inverting F takes $\widetilde{O}(M(n))$ bit operations [AHU 74, Theorem 6.5]. Finding a basis for the null space of $F^{-1}(X,Y)$ takes $\widetilde{O}(M(n))$ bit operations [IMH 82, pp. 53–54]. The remaining computations are easy to execute within this bound. Consider the randomized parallel complexity of the algorithm. Inverting F takes parallel time $O((\log n)^2)$ using $\widetilde{O}(M(n))$ processors ([KP 91, Theorem 6]), and these complexity bounds suffice for finding a basis for the null space of $F^{-1}(X,Y)$ ([KP 91, p. 190]) (both computations are randomized, and on a given matrix the computed results may be incorrect with small probability). The remaining steps are easy to implement within these complexity bounds.

ALGORITHM 3. *Monte Carlo Minimum $X \to Y$ Separator*
INPUT: Graph $G = (V, E)$.
OUTPUT: With high probability, a minimum $X \to Y$ separator of G.

Order the vertices, and number them $1, 2, \ldots, n$.
Fix the number $W = n^{O(1)}$, and choose a prime q, $W < q = n^{O(1)}$.
Construct the matrix F by replacing each nonzero entry in the free adjacency matrix of G by an independent random number from $\{1, 2, \ldots, W\}$.

Invert F over the field $\mathbb{Z}_q$ to obtain the matrix F^{-1}.
(If F is singular, the algorithm stops and reports failure.)

Compute the rank r of the submatrix $F^{-1}(X, Y)$ over $\mathbb{Z}_q$.
If $r = |Y|$, then the required separator is $S = Y$. Stop.

Otherwise, compute a basis $\{N_1, \ldots, N_{|Y|-r}\}$ for the null space of the submatrix $F^{-1}(X, Y)$ over $\mathbb{Z}_q$ (each N_i is a vector of dimension $|Y|$).
Compute the matrix $Z = F^{-1}(V, Y) \cdot N$ over $\mathbb{Z}_q$,
where N is the matrix whose ith column is N_i.
Let Z_v denote the row of Z given by $F^{-1}(\{v\}, Y) \cdot N$.
Construct the set of vertices $T = \{v \mid Z_v \text{ is a nonzero vector}\}$.

With high probability, the required separator S consists of the predecessors of T together with the Y-vertices not in T, i.e.,

$$S = \Delta(T) = \{s \in V - T \mid (s, v) \in E \text{ and } v \in T\} \cup (Y - T).$$

Making the algorithm Las Vegas:
If $G - S$ has no path from $X - S$ to $Y - S$, and $|S| = r$,
then guarantee that S is a minimum $X \to Y$ separator; otherwise, report failure.

FIG. 5.

THEOREM 4.4. *Given a digraph G and a pair of sets X, Y of G's vertices, a minimum $X \to Y$ separator can be computed by a Las Vegas algorithm. The sequential running time is $\widetilde{O}(M(n))$. The parallel complexity is $O((\log n)^2)$ time using $\widetilde{O}(M(n))$ processors. The same complexity bounds apply for finding a minimum vertex cover of a bipartite graph.*

5. Conclusions. The most important problem left open is whether a maximum matching can be computed in deterministic or randomized time $O(n^{2.5-\epsilon})$, $\epsilon > 0$. The same problem specialized to bipartite graphs, or equivalently (by Proposition 2.4) the problem of finding a maximum-cardinality set of vertex disjoint $X \to Y$ paths in a digraph in (randomized) time $O(n^{2.5-\epsilon})$, is also open. Another interesting open problem pertains to graphs with 0–1 weights on the edges: Can the maximum weight of a perfect matching, but not necessarily the edge-set of the matching, be computed in (randomized) time $O(n^{2.5-\epsilon})$? Can the algorithm of Theorem 3.8 for finding the allowed edges be made Las Vegas without affecting the complexity bounds? The algorithm for finding a minimum bipartite vertex cover may be derived starting either

from the Gallai–Edmonds theorem (Theorem 3.1) or from the theorem on the free adjacency matrix (Theorem 4.2). Do these two theorems have other connections?

6. Appendix. Proofs of Proposition 2.4 and Theorem 4.2.

Proof of Proposition 2.4. First, we show how to transform instances and recover solutions of the two bipartite graph problems using the corresponding digraph problems. Let $H = (V_1, V_2, E)$ be a bipartite graph, where V_1 and V_2 are the sets of the vertex bipartition. Construct a digraph $G = (V_1 \cup V_2, F)$ from H by orienting all edges from V_1 to V_2. Every matching of H corresponds to a set of vertex disjoint $V_1 \rightarrow V_2$ paths in G. Hence, a maximum matching of H can be found by computing a maximum cardinality set of vertex disjoint $V_1 \rightarrow V_2$ paths in G. Consider the bipartite graph minimum vertex cover problem. A subset of $V_1 \cup V_2$ is a vertex cover of H iff it is a $V_1 \rightarrow V_2$ separator of the digraph G. Therefore, a minimum vertex cover of H can be found by computing a minimum $V_1 \rightarrow V_2$ separator of G.

Next consider the transformation of the two digraph problems to the corresponding bipartite graph problems, and the transformation of the solutions. Let $G = (V, F)$ be the given digraph, and let X and Y be specified subsets of V. Without loss of generality assume that $X \cap Y = \emptyset$; the method here easily extends to the case when $X \cap Y \neq \emptyset$. Let $\overline{n}$ denote $|V(G) - X - Y|$. We construct a bipartite graph $H = (V_1, V_2, E)$ starting from G, X, Y. For each vertex $v \in V(G) - X - Y$, H has a pair of vertices v_1, v_2 with $v_1 \in V_1$ and $v_2 \in V_2$; H also has the edge $v_1 v_2$; for each vertex $x \in X$, H has a vertex $x_1 \in V_1$; and for each vertex $y \in Y$, H has a vertex $y_2 \in V_2$. For every vertex v of G let v_1 and v_2 denote the corresponding vertices in V_1 and V_2 (if they exist); let X_1 denote the set of vertices of H that corresponds to X. For each edge (v, w) of G, $v \notin Y$ and $w \notin X$, there is an edge $v_1 w_2$ in H.

A set of vertex disjoint $X \rightarrow Y$ paths of G of maximum cardinality (namely, $p(X, Y)$) gives a matching E' of H with $|E'| = p(X, Y) + \overline{n}$: start with $E' = \{v_1 v_2 : v \in V(G) - X - Y\}$ and then, sequentially for each of the $X \rightarrow Y$ paths of G in the set mentioned above, augment E' using the corresponding alternating path of H. Moreover, we claim that a matching E' of H gives a set of at least $|E'| - \overline{n}$ vertex disjoint $X \rightarrow Y$ paths of G: starting from the vertices in X_1 in H, use the matching E' and the edges $v_1 v_2$, $v \in V(G) - X - Y$, to construct a set of vertex disjoint paths in G; each of these paths ends either at a vertex in Y or at a vertex $v \notin Y$ such that in H the corresponding vertex v_1 is exposed; hence, at most $|V_1| - |E'| = |X_1| + \overline{n} - |E'|$ of these paths in G have their end vertices in $V - Y$; our claim follows since the number of these paths in G is $|X_1|$. Consequently, $\nu(H) = p(X, Y) + \overline{n}$, and every maximum matching of H yields a set of $p(X, Y)$ vertex disjoint $X \rightarrow Y$ paths of G.

Now consider the problem of finding a minimum $X \rightarrow Y$ separator S of G. We find a minimum vertex cover C of H and then construct S as follows: S contains a vertex $x \in X$ iff C contains the vertex x_1; S contains a vertex $y \in Y$ iff C contains the vertex y_2; and S contains a vertex $v \in V(G) - X - Y$ iff C contains both the vertices v_1 and v_2. Since $|C| = \nu(H) = p(X, Y) + \overline{n}$, and since either $v_1 \in C$ or $v_2 \in C$ for each vertex $v \in V(G) - X - Y$, we see that $|S| = p(X, Y)$. We claim that S is an $X \rightarrow Y$ separator of G. By way of contradiction, suppose that there is a path P in $G - S$ with start vertex $x \in X$ and end vertex $y \in Y$. Focus on the subgraph $H(P)$ of H formed by the edges that correspond to the edges of P, together with the edges $v_1 v_2$ of H that correspond to the internal vertices v of P (i.e., $v \in V(P) - \{x, y\}$). Since C is a vertex cover of H, every edge of $H(P)$ must be incident with some vertex of C. Consequently, either $x \in C$ or $y \in C$ or there is an internal vertex v of P such that $v_1 \in C$ and $v_2 \in C$. We have the desired contradiction since S intersects

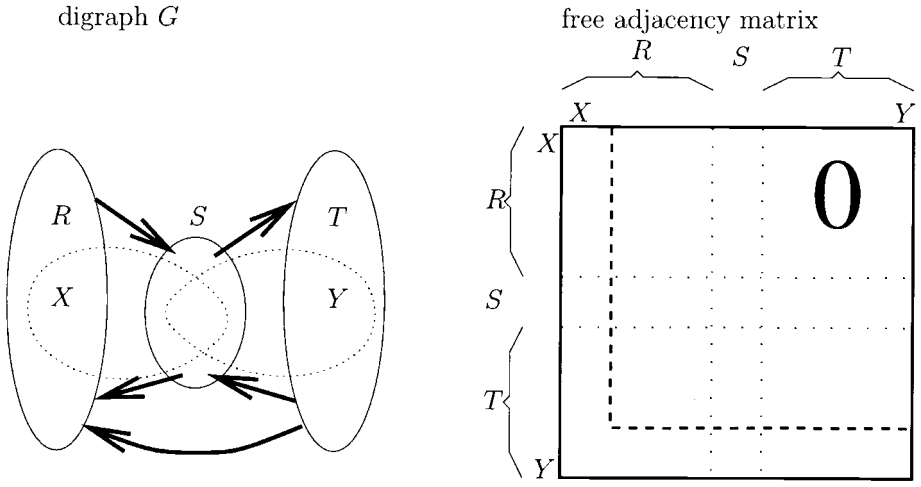


FIG. 6. *An illustration of the proof of Theorem* 4.2. *The submatrix* $\widetilde{F}(\overline{Y}, \overline{X})$ *is indicated by dashed lines.*

P. □

Proof of Theorem 4.2. First, consider the case when $p(X, Y)$ is less than $k = |Y|$. Let S be an $X \rightarrow Y$ separator of cardinality $p(X, Y)$ and let T denote the set of vertices that have paths to $Y - S$ in $G - S$. Let R denote $V - (S \cup T)$. Note that X is a subset of $R \cup S$ and Y is a subset of $S \cup T$. Since G has no edges of the form (r, t), $r \in R$, $t \in T$, each entry of the submatrix $\widetilde{F}(R, T)$ is zero. (See Figure 6.) A line denotes either a row or a column of a matrix. Focus on the number of lines needed to cover all the nonzero entries of $\widetilde{F}(\overline{Y}, \overline{X})$, and consider the columns corresponding to the vertex set $(R \cup S) - X$ and the rows corresponding to the vertex set $(S \cup T) - Y$. Each entry of $\widetilde{F}(\overline{Y}, \overline{X})$ that is not covered by these lines is in an R-row and a T-column; hence the entry is zero. Thus the number of lines needed is at most

$$((n - |T|) - k) + ((n - |R|) - k) \leq (n - k - 1),$$

since $n = |R| + |S| + |T| \leq (k - 1) + |R| + |T|$. Now use the fact that for a bipartite graph, the cardinality of every matching is less than or equal to the cardinality of every vertex cover. It follows that there are at most $(n - k - 1)$ nonzero entries in $\widetilde{F}(\overline{Y}, \overline{X})$ with no two of these entries on a line. Hence, $\det \widetilde{F}(\overline{Y}, \overline{X})$ is identically zero since each term in the standard expansion of the determinant is zero.

Next suppose that $p(X, Y)$ equals $k = |Y|$. Let $P_1, \ldots, P_k$ be a maximum set of vertex disjoint $X \rightarrow Y$ paths. Denote the start vertex of path P_i $(1 \leq i \leq k)$ by x_i and denote the end vertex by y_i. Let A denote the set of vertices not in these paths. For each vertex $v \in A$ define $\sigma(v)$ to be v, and for each vertex $v \in (V - A - Y)$, define $\sigma(v)$ to be the successor of v in the path P_i containing v. Note that σ is well defined even if trivial paths P_i (having $x_i = y_i$) are present. For each $v \in \overline{Y}$, note that $\sigma(v)$ belongs to $\overline{X}$ and that $\widetilde{F}_{v,\sigma(v)}$ is a nonzero entry of the submatrix $\widetilde{F}(\overline{Y}, \overline{X})$. Moreover, observe that σ is one-one, i.e., $\sigma(v) = \sigma(w)$ iff $v = w$, and therefore no two entries from the set $\{\widetilde{F}_{v,\sigma(v)} \mid v \in \overline{Y}\}$ are on a line. It follows that the product $(\pm 1) \cdot \Pi_{v \in \overline{Y}} \widetilde{F}_{v,\sigma(v)}$ is one of the terms in the standard expansion of $\det \widetilde{F}(\overline{Y}, \overline{X})$. Clearly the product is

nonzero. Hence, the determinant evaluates to ± 1 when the value 1 is assigned to each entry $\widetilde{F}_{v,\sigma(v)}$, $v \in \overline{Y}$, and the value 0 is assigned to the remaining entries of $\widetilde{F}(\overline{Y}, \overline{X})$. Therefore, the determinant is not identically zero. □

Acknowledgments. Section 4 has benefited from discussions with Éva Tardos. The Monte Carlo algorithm for the canonical partition in section 3.4 was discovered jointly with K. Padayachee and is included with his consent. J. A. La Poutré communicated an almost linear-time algorithm for verifying the canonical partition. The careful comments by the referees are appreciated.

REFERENCES

[AHU 74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[B 90] N. Blum, *A new approach to maximum matching in general graphs*, Proc. 17th ICALP, Lecture Notes in Comput. Sci. 443, Springer-Verlag, Berlin, 1990, pp. 586–597.

[BR 91] R. A. Brualdi and H. J. Ryser, *Combinatorial Matrix Theory*, Cambridge University Press, New York, 1991.

[C 93] J. Cheriyan, *Random weighted Laplacians, Lovász minimum digraphs and finding minimum separators*, extended abstract in Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, TX, SIAM, Philadelphia, 1993, pp. 31–40.

[C 94] J. Cheriyan, *A Las Vegas $O(n^{2.38})$ algorithm for the cardinality of a maximum matching*, extended abstract in Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, SIAM, Philadelphia, 1994, pp. 442–451.

[CR 94] J. Cheriyan and J. H. Reif, *Directed s-t numberings, rubber bands, and testing digraph k-vertex connectivity*, Combinatorica, 14 (1994), pp. 435–451.

[CW 90] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[E 91] W. Eberly, *Efficient parallel independent subsets and matrix factorizations*, in Proc. 3rd IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 204–211.

[E 65] J. Edmonds, *Paths, trees and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[GT 91] H. N. Gabow and R. E. Tarjan, *Faster scaling algorithms for general graph-matching problems*, J. Assoc. Comput. Mach., 38 (1991), pp. 815–853.

[GP 88] Z. Galil and V. Pan, *Improved processor bounds for combinatorial problems in RNC*, Combinatorica, 8 (1988), pp. 189–200.

[Ga 64] T. Gallai, *Maximale Systeme unabhängiger Kanten*, Magyar Tud. Akad. Mat. Kutató Int. Közl, 9 (1964), pp. 401–413.

[HK 95] M. Rauch Henzinger and V. King, *Randomized dynamic algorithms with polylogarithmic time per operation*, in Proc. 27th Annual ACM Symposium on Theory of Computing, Las Vegas, NV, ACM, New York, 1995, pp. 519–527.

[IMH 82] O. H. Ibarra, S. Moran, and R. Hui, *A generalization of the fast LUP matrix decomposition algorithm and applications*, J. Algorithms, 3 (1982), pp. 45–56.

[IP 73] A. W. Ingleton and M. J. Piff, *Gammoids and transversal matroids*, J. Combin. Theory Ser. B, 15 (1973), pp. 51–68.

[KP 91] E. Kaltofen and V. Pan, *Processor efficient parallel solution of linear systems over an abstract field*, in Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1991, pp. 180–191.

[Kf 86] H. J. Karloff, *A Las Vegas RNC algorithm for maximum matching*, Combinatorica, 6 (1986), pp. 387–391.

[KUW 86] R. M. Karp, E. Upfal, and A. Wigderson, *Constructing a perfect matching is in Random NC*, Combinatorica, 6 (1986), pp. 35–48.

[Ko 91] D. Kozen, *The Design and Analysis of Algorithms*, Springer-Verlag, Berlin, 1991.

[L 95] J. A. La Poutré, *Personal communication*, 1995.

[LW 94] J. A. La Poutré and J. Westbrook, *Dynamic two-connectivity with backtracking*, in Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, SIAM, Philadelphia, 1994, pp. 204–212.

[LLW 88] N. Linial, L. Lovász, and A. Wigderson, *Rubber bands, convex embeddings and graph connectivity*, Combinatorica, 8 (1988), pp. 91–102.

[Lo 79] L. Lovász, *On determinants, matchings and random algorithms*, in Fundamentals of

Computation Theory, L. Budach, ed., Akademie-Verlag, Berlin, 1979, pp. 565–574.

[Lo 93] L. Lovász, *Combinatorial Problems and Exercises*, 2nd ed., North-Holland, Amsterdam, 1993.

[LP 86] L. Lovász and M. Plummer, *Matching Theory*, Akadémiai Kiadó, Budapest, Hungary, 1986.

[MV 80] S. Micali and V. V. Vazirani, *An $O(\sqrt{|V||E|})$ algorithm for finding maximum matching in general graphs*, in Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, Syracuse, NY, IEEE Computer Society Press, Los Alamitos, CA, 1980, pp. 17–27.

[MVV 87] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[RV 89] M. O. Rabin and V. V. Vazirani, *Maximum matchings in general graphs through randomization*, J. Algorithms, 10 (1989), pp. 557–567.

[Sc 80] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, J. ACM, 27 (1980), pp. 701–717.

[ST 83] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

[T 47] W. T. Tutte, *The factorization of linear graphs*, J. London Math. Soc., 22 (1947), pp. 107–111.

[V 94] V. V. Vazirani, *A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph matching algorithm*, Combinatorica, 14 (1994), pp. 71–109.

[W 91] J. Wein, *Las Vegas RNC algorithms for unary weighted perfect matching and T-join problems*, Inform. Process. Lett., 40 (1991), pp. 161–167.

[Z 79] R. E. Zippel, *Probabilistic algorithms for sparse polynomials*, in Proc. EUROSAM 79, Edward W. Ng, ed., Lecture Notes in Comput. Sci. 72, Springer-Verlag, Berlin, 1979, pp. 216–226.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1656–1669, December 1997

005

MAXIMUM AGREEMENT SUBTREE IN A SET OF EVOLUTIONARY TREES: METRICS AND EFFICIENT ALGORITHMS*

AMIHOOD AMIR† AND DMITRY KESELMAN†

Abstract. The *maximum agreement subtree* approach is one method of reconciling different evolutionary trees for the same set of species. An agreement subtree enables choosing a subset of the species for whom the restricted subtree is *equivalent* (under a suitable definition) in all given evolutionary trees.

Recently, dynamic programming ideas were used to provide polynomial time algorithms for finding a maximum homeomorphic agreement subtree of *two* trees. Generalizing these methods to sets of more than two trees yields algorithms that are exponential in the number of trees. Unfortunately, it turns out that in reality one is usually presented with more than two trees, sometimes as many as thousands of trees.

In this paper we prove that the maximum homeomorphic agreement subtree problem is $\mathcal{NP}$-complete for three trees with unbounded degrees. We then show an approximation algorithm of time $O(kn^5)$ for choosing the species that are *not* in a maximum agreement subtree of a set of k trees. Our approximation is guaranteed to provide a set that is no more than 4 times the optimum solution.

While the set of evolutionary trees may be large in practice, the trees usually have very small degrees, typically no larger than three. We develop a new method for finding a maximum agreement subtree of k trees, of which one has degree bounded by d. This new method enables us to find a maximum agreement subtree in time $O(kn^{d+1} + n^{2d})$.

Key words. evolutionary trees, maximum agreement subtrees, classification

AMS subject classifications. 68Q20, 68Q25

PII. S0097539794269461

1. Introduction.

One of the methods for classifying hierarchical relations between different objects is by representing them in a tree [12]. In particular, trees have been used to represent evolutionary splits among species (cf. [11, 18, 3]). Different methods of classification may lead to different trees. It is natural to try to resolve differing evolutionary trees in a manner that will increase our confidence in the results.

There are two ways one can go about handling different evolutionary trees for the same species. One may try to construct a *consensus tree* that is "close" to all given trees. This direction was taken by several researchers (e.g., [15]). Another direction, the one that concerns us in this paper, is extracting a maximum set of species about whom we are confident. This method, introduced by Gordon [10], involves obtaining a *maximum agreement subtree*.

There are several alternate ways to define a maximum agreement subtree. One approach was taken by Finden and Gordon [7].

DEFINITION. *Let $S = \{s_1, \ldots, s_n\}$ be a set of* labels. *An S-*labeled *tree T is a tree with n leaves, each labeled with a distinct element of S; i.e., no two leaves have the same label. Let T be an S-labeled tree and let $S' \subseteq S$. $H(T, S')$ is the minimal*

* Received by the editors June 10, 1994; accepted for publication (in revised form) October 30, 1995. A preliminary version of this paper appeared in Proc. FOCS 94, 35th IEEE Symposium on Foundations of Computer Science, Santa Fe, NM, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 758–769.

http://www.siam.org/journals/sicomp/26-6/26946.html

† College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280 (amir@cc.gatech.edu, dmitry@cc.gatech.edu). Part of this work was completed while the first author was at Bar-Ilan University, Israel. The research of the first author was partially supported by NSF grant CCR-92-23699 and Israel Ministry of Science and Arts grant 6297.

homeomorphic subtree of T (all degree 2 nodes are contracted) containing exactly the leaves labeled by S'.

Let $T_1, \ldots, T_k$ be S-labeled trees. A maximum homeomorphic agreement subtree *of $T_1, \ldots, T_k$ ($MHT(T_1, \ldots, T_k)$) is a maximum cardinality set $S' \subseteq S$ such that $H(T_1, S') = H(T_2, S') = \cdots = H(T_k, S')$. We will also refer to the problem of finding a maximum homeomorphic agreement subtree as* the MHT problem.

We will interchangeably refer to an MHT as a *tree* or *set of labels*. There is no ambiguity since a set of labels uniquely defines the contracted subtree whose leaves are exactly the given set, if that set of labels is in a homeomorphic agreement subtree.

The most general possible way to view agreement subtree is by assigning each edge an *interval weight*, i.e., the conjectured *range* of time it took to evolve along this edge. This idea is very similar to the "graph-sandwich" of Farach, Kannan, and Warnow [4]. Informally, the maximum interval weight agreement subtree (MIWT) of k S-labeled trees $T_1, \ldots, T_k$ is a maximum set $S' \subseteq S$ such that $\forall s, t \in S'$ the distance between s and t is within the allowable range in all trees $T_1, \ldots, T_k$. A precise definition is provided in section 7.

Finden and Gordon [7] gave a heuristic algorithm for the MHT problem for *two trees* T_1, T_2. Their result is not guaranteed to be an $MHT(T_1, T_2)$. An algorithm of complexity $O(n^{1/2+\epsilon \log_2 n})$ was presented by E. Kubicka, G. Kubicki, and McMorris [14]. This algorithm is also for the MHT problem of two trees. This result was improved by Warnow and Steel [17] to finding $MHT(T_1, T_2)$ in time $O(n^2)$ for bounded degree trees and $O(n^{4.5} \log n)$ for unbounded degree trees. A similar method was independently found by Goddard et al. [9] for finding $MHT(T_1, T_2)$ of binary trees in time $O(n^2)$. Farach and Thorup [6] further improved these results to $O(n^2 c^{\sqrt{\log n}})$ for unbounded degree trees.

The algorithms of [17, 6, 9] handle only the restricted case of two trees. They can be generalized, but then they become exponential in the number of trees. In reality, sometimes *thousands* of trees need to be considered [9]. As for the maximum interval weight agreement subtree, there are currently no solutions to this problem.

In this paper we present the first known solution to the realistic versions of the maximum agreement subtree problem. In addition, we use novel methods for tackling this problem. The main idea behind our algorithms is to use divide and conquer rather than dynamic programming. A naive top-down algorithm would quickly reach exponential time, but we exploit some subtle internal properties of tree partitions in a manner that assures an efficient algorithm. Surprisingly, the algorithm is simple and easily programmable.

The main contributions of this paper are the following:

- We show that for unbounded degree trees, $MHT(T_1, T_2, T_3)$ and $MIWT(T_1, T_2, T_3)$ are $\mathcal{NP}$-complete.
- We present an approximation algorithm for finding a set of leaves that are *not* in an MHT of k unbounded degree evolutionary trees. Our algorithm runs in time $O(kn^5)$. The approximation factor of our algorithm is 4.
- For trees $T_1, \ldots, T_k$, where at least one of which has degree bounded by d, we present algorithms for finding $MHT(T_1, \ldots, T_k)$ and $MIWT(T_1, \ldots, T_k)$ in time $O(kn^{d+1} + n^{2d})$.

This paper is organized as follows. In section 2, we prove the $\mathcal{NP}$-completeness of the $MHT(T_1, T_2, T_3)$ and $MIWT(T_1, T_2, T_3)$ problem for unbounded degree trees. In section 3 we give the approximation algorithm for the MHT problem of k unbounded degree trees. In section 4, we define a very simple version of the MIWT problem (the

maximum isomorphic agreement subtree or MIT problem) and give an efficient algorithm for trees of bounded degree. In section 5 we show the algorithm for computing an MHT of k trees, one of which has degree bounded by d. In section 6 we consider the special case of the MIWT problem where all weight ranges are a single weight (the MWT problem). We give an efficient algorithm for finding $MWT(T_1, \ldots, T_k)$, where at least one of the trees has a small bounded degree. In section 7 we provide an efficient algorithm for the maximum interval agreement subtree of a set of trees. We conclude with open problems and future research.

2. Multiple unbounded degree trees. We show that finding the $MHT(T_1, T_2, T_3)$ is an $\mathcal{NP}$-complete problem (the same proof applies to $MIWT(T_1, T_2, T_3)$). First define the decision problem:

Homeomorphic agreement subtree of 3 *unbounded degree trees (*3*-HUT).*

INSTANCE: Three S-labeled trees T_1, T_2, T_3 of unbounded degree, where $S = \{s_1, \ldots, s_n\}$;
integer $i \leq n$.

QUESTION: Is there a subset $S' \subseteq S$ of size i such that $H(T_1, S') = H(T_2, S') = H(T_3, S')$?

THEOREM 1. *The* 3-HUT *problem is $\mathcal{NP}$-complete.*

Proof. It is clearly in $\mathcal{NP}$. We will reduce the 3-dimensional matching problem (3DM) [8] to 3-HUT. In 3DM the input is a set $M \subseteq W \times X \times Y$, where W, X, Y are disjoint q-element sets. We need to answer if there is a set $M' \subseteq M$ of size q, where no two elements of M' agree in any coordinate.

Construct three trees T_1, T_2, T_3 as follows. Each tree T_i has a root r_i and each root has q children. The children of r_1 correspond to the elements of W, the children of r_2 correspond to the elements of X, and the children of r_3 correspond to the elements of Y. Take $S = M$. For every element $e = \langle w, x, y \rangle \in M$ attach a child labeled e to the node that corresponds to w in T_1, to the node that corresponds to x in T_2, and to the node that corresponds to y in T_3.

Assuming that the roots r_1, r_2, r_3 appear in a maximum homeomorphic subtree, then any two elements of M that agree on one or two coordinates will not be on a homeomorphic subtree of all three trees. We force the roots to be in a maximum homeomorphic subtree, by adding to each root $q + 1$ children labeled $x_1, \ldots, x_{q+1}$, where the x_i are new and distinct symbols. Thus it is easy to see that there is an M' of size q iff there is a homeomorphic subtree of all three trees of size $2q + 1$. □

3. An approximation algorithm. We have seen that constructing the set of leaves whose restricted subtree is homeomorphic to their restricted subtree in all trees is an $\mathcal{NP}$-hard problem. We now provide an approximation algorithm for constructing the set of leaves that are not in an MHT. The approximation algorithm is based on the following property, which was first proved by Bandelt and Dress [2]. We present the theorem and a new proof.

THEOREM 2. *Two evolutionary trees are homeomorphic iff all* 4*-leaf subtrees generate homeomorphic subtrees.*

Proof. One direction is immediate. We will prove that if all 4-leaf subtrees generate homeomorphic subtrees then the trees are homeomorphic.

Call a vertex of degree greater than 2 a *non-*2*-vertex.* Call two leaves *twins* if the path connecting them has at most one intermediate non-2-vertex. Every tree contains at least one pair of twins. This can be seen by the following argument. Consider a path with the largest number of non-2-vertices. Let the leaf ends of this path be a and b. Let v be the closest non-2-vertex to a (a similar argument works for b). There

is a path from v to a, to b, and to at least some other leaf c. In the path from v to a and from v to c there are no non-2-vertices otherwise, the path from a to b would not be maximal. Therefore, a and c are twins.

Proceed with proving theorem by induction on the number of leaves n. If $n \leq 4$ theorem is trivially true. For $n > 4$ let x and y be a pair of twins in tree T_1 whose intermediate non-2-vertex is z. We claim that x and y are twins in tree T_2 as well. Otherwise, let z_1 and z_2 be two non-2-vertices on the path from x to y in T_2. Since both are non-2-vertices, there exist leaves p and q such that p branches off z_1 and q branches off z_2. Thus, in the subtree generated by x, y, p, q in T_2, x and y are separated by 2 non-2-vertices; whereas in the subtree generated by x, y, p, q in T_1 they are separated by only one non-2-vertex. Hence, these two subtrees cannot be homeomorphic. See Figure 1.

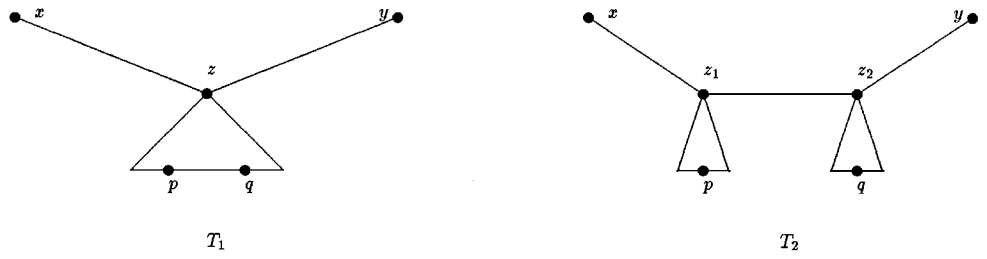


FIG. 1.

Now remove the path from z to y in both trees. Inductively, the resulting two trees are homeomorphic. Clearly then, adding the path from z to y in both T_1 and T_2 will still yield two homeomorphic trees. □

Let $S4$ be the set of all 4-element subsets of S that do not generate a homeomorphic subtree in all trees. $S4$ can be constructed in time $O(kn^5)$. A set $S' \subset S$ is a *cover* of $S4$ if $\forall\{a, b, c, d\} \in S4\ \exists e \in S'$ such that $e \in \{a, b, c, d\}$. S' is a *minimum cover* if S' is a cover of $S4$ and every other cover is at least as large as S'.

Theorem 2 implies that if S' is a minimum cover of $S4$ then the homeomorphic restriction of $S - S'$ in all trees is an MHT.

The following algorithm is similar to the approximation algorithm for vertex cover [8].

Approximation Algorithm

1. $S_a \leftarrow \emptyset$
2. While $S4 \neq \emptyset$ do
   - 2.1 Choose an element $\alpha \in S4$
   - 2.2 $S_a \leftarrow S_a \cup \alpha$
   - 2.3 Delete from $S4$ all elements β such that $\alpha \cap \beta \neq \emptyset$

end Algorithm

Algorithm time $O(|S4|) = O(n^4)$. (The construction of $S4$ requires an additional $O(kn^5)$ time.)

THEOREM 3. *The set S_a produced by the approximation algorithm is a cover of $S4$ and is at most four times the size of a minimum cover of $S4$.*

Proof. S_a is clearly a cover. Let A be the set of all elements of $S4$ chosen by step 2.1. For any $\alpha_1, \alpha_2 \in A$, $\alpha_1 \cap \alpha_2 = \emptyset$. Thus any cover of $S4$ has at least $|A|$ elements. S_a has $4|A|$ elements. □

4. Maximum isomorphic agreement subtree. For simplicity's sake, we start by presenting an algorithm for the MIWT problem for k trees, where the edge weights are uniformly 1. We call this the maximum isomorphic agreement subtree problem (MIT), formally defined below. The general MIWT problem is handled in section 7.

DEFINITION. *Let $T_1, \ldots, T_k$ be S-labeled trees. A* maximum isomorphic agreement subtree *of $T_1, \ldots, T_k$ ($MIT(T_1, \ldots, T_k)$) is a maximum set $S' \subseteq S$ such that $\forall s, t \in S'$ the distance between s and t is the same in all trees $T_1, \ldots, T_k$.*

The following tree property, due to Smolenskii [16], establishes that the subtrees induced by $MIT(T_1, \ldots, T_k)$ in $T_1, \ldots, T_k$ are indeed isomorphic.

THEOREM 4. *Two labeled trees are isomorphic iff the distance between any two leaves with corresponding labels is the same in both trees.*

To further elucidate our idea, let us restrict ourselves to the case of binary trees $T_1, \ldots, T_k$. We also introduce the following notation. Assume we fix a node labeled r as the root. Since r appears in all our trees, we may now consider $T_1, \ldots, T_k$ as directed trees rooted at r. Since all edges are now directed, there is a unique subtree rooted at every node. Let x be a node in T_i. Denote the subtree rooted at x by $T_i(x)$. We denote the distance of node x from the root as a superscript, i.e., if $d(x, r) = p$ then we write x^p.

The main idea behind our algorithm is to use a top-down approach. We will show that the structure of our problem prevents an exponential time blow-up.

The first crucial observation we make is that there is at least one label in $MIT(T_1, \ldots, T_k)$. If we had a priori knowledge of such a label, say r, we could root all the trees at r. This fixes the direction of the edges, so for the next levels in the recursion there is only one possible root for every subtree.

The following lemma is also an important factor in understanding our algorithm.

LEMMA 5. *Let s and t be leaves of $MIT(T_1, \ldots, T_k)$. Suppose there is a tree T_i and $x_i^p \in T_i$ such that $s, t \in T_i(x_i^p)$. Then for every $j = 1, \ldots, k$ there exists a node $x_j^p \in T_j$ for which $s, t \in T_j(x_j^p)$.*

Proof. Consider the meeting point of the three pairwise paths of the leaves s, t, and r. Since s, t, r are leaves of $MIT(T_1, \ldots, T_k)$, then Theorem 4 says their distances from each other are the same in all the trees $T_1, \ldots, T_k$. These distances $d(s, t), d(s, r)$, and $d(t, r)$ determine that the distance from r to the meeting point is $(d(r, s) + d(r, t) - d(s, t))/2 = p$ in all trees. ∎

Algorithm Outline

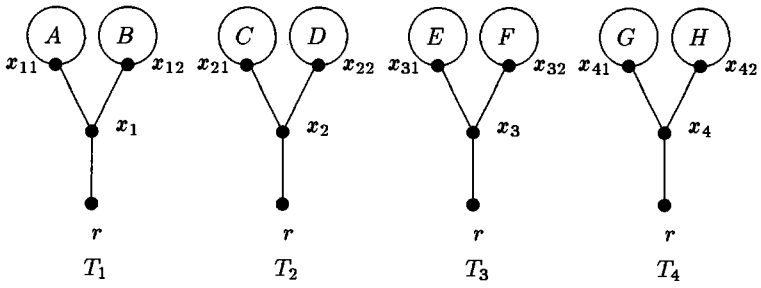
for $r = s_1$ to s_n do
    root all trees at r.
    find $MIT(T_1, \ldots, T_k)$ (A recursive algorithm for rooted MIT follows).
choose the largest of all n rooted MIT's.

end Algorithm

Before we proceed to the algorithm for rooted MIT, we define the concept of *thread intersection*. This is a key concept that appears in many of our algorithms. It is properties of these intersections that prevent an exponential time blow-up of the algorithms.

We first intuitively describe the concept. Suppose a set of k nodes $x_i \in T_i,\ i = 1, ..., k$ is given. Each of these nodes may have several children. A thread is a choice of one son of each of the x_i's.

DEFINITION. *Let $\{x_1, \ldots, x_k\}$ be a fixed set of nodes where $x_i \in T_i,\ i = 1, \ldots, k$, and let $L(x_i) \subseteq S$ be the set of labels in $T(x_i)$ (the subtree rooted at x_i). Let $\{x_{i1}, \ldots, x_{ip_i}\}$ be the set of children of x_i.*



The thread $\sigma = \langle 1, 1, 2, 1\rangle$ for nodes x_1, x_2, x_3, x_4 defines thread intersection $A \cap C \cap F \cap G$.
The thread $\overline{\sigma} = \langle 2, 2, 1, 2\rangle$ for nodes x_1, x_2, x_3, x_4 defines thread intersection $B \cap D \cap E \cap H$.

FIG. 2. *Examples of threads.*

A k-tuple $\sigma_{\langle x_1,\ldots,x_k\rangle} = \sigma[1], \ldots, \sigma[k]$ where $\sigma[i] \in \{1, \ldots, p_i\}$ is called a thread *for nodes $x_1, \ldots, x_k$. (In other words, a thread for a set of k nodes is a pointer from each of the k nodes to a specific one of its children.) Wherever the set of nodes $\{x_1, \ldots, x_k\}$ is clear from the text we will drop the index $\langle x_1, \ldots, x_k\rangle$ and denote the thread simply by σ. A nonempty intersection $\cap_{i=1}^{k} L(x_{i\sigma[i]})$ is a* thread intersection.

For an example see Figure 2.

Let A be the union of all thread intersections for a fixed set of nodes. The following simple observation guarantees that the nonempty thread intersections form a partition of A.

LEMMA 6. *Let σ_1, σ_2 be threads for nodes $x_1, \ldots, x_k$. If $\sigma_1 \neq \sigma_2$ then*

$$\left(\bigcap_{i=1}^{k} L(x_{i\sigma_1[i]})\right) \cap \left(\bigcap_{i=1}^{k} L(x_{i\sigma_2[i]})\right) = \emptyset.$$

Proof. If $\sigma_1 \neq \sigma_2$ then there is a j such that $\sigma_1[j] \neq \sigma_2[j]$.
$\cap_{i=1}^{k} L(x_{i\sigma_1[i]}) \subseteq L(x_{j\sigma_1[j]})$.
$\cap_{i=1}^{k} L(x_{i\sigma_2[i]}) \subseteq L(x_{j\sigma_2[j]})$.
But $\sigma_1[j] \neq \sigma_2[j]$; therefore $L(x_{j\sigma_1[j]}) \cap L(x_{j\sigma_2[j]}) = \emptyset$, and thus

$$\left(\bigcap_{i=1}^{k} L(x_{i\sigma_1[i]})\right) \cap \left(\bigcap_{i=1}^{k} L(x_{i\sigma_2[i]})\right) = \emptyset. \qquad \square$$

The idea behind our algorithm is the straightforward one. Assume we are guaranteed that node r is in a MIT. We now root all the trees at r. Theorem 4 and Lemma 5 now assure us that if other nodes appear in the MIT then the child x_i of r in each tree T_i, $i = 1, \ldots, k$, has to be in the MIT also.

Thus a naive way of constructing the MIT is by considering all possible combinations of children of the x_i's, $i = 1, \ldots, k$. In particular, if all the trees are binary trees, assume the children of x_i are x_{i1} and x_{i2}. For each thread σ of the nodes $x_1, \ldots, x_k$ recursively find an MIT T_1 of $\cap_{i=1}^{k} L(x_{i\sigma[i]})$ and an MIT T_2 of $\cap_{i=1}^{k} L(x_{i\overline{\sigma}[i]})$. Construct a tree whose root is r, its child is x_1 and make the roots of T_1 and T_2 the children of x_1 . Each such tree is an isomorphic agreement subtree. The largest of them is a maximal isomorphic agreement subtree.

The naive algorithm described above has exponential running time since there are 2^k combinations to check at each level of the recursion. The algorithm below is

essentially the same as the naive algorithm but avoids the need to check an exponential number of thread combinations by exploiting the fact that the threads form a partition; hence there are only very few nonempty thread intersections. We check only the nonempty ones.

Recursive Algorithm for Finding Rooted MIT of Binary Trees

1. prune all trees to include only the labels that are equidistant to r in all trees.
   The trees are now of the following form: Tree T_i has root r. r has one child x_i^1. x_i^1 has two children x_{i1}^2 and x_{i2}^2. The rest of the tree is rooted at x_{i1}^2 and x_{i2}^2. We would like to have recursion and choose a maximal isomorphic agreement subtrees of all combinations of children. However, there are 2^k combinations, which we cannot afford. The partition property of thread intersections means there are at most $n-1$ nonempty candidate intersections.
2. construct the i disjoint thread intersections $\{S_1, \ldots, S_i\}$ of $S - \{r\}$ $(i \leq n-1)$.
   Details of the construction appear in the implementation of step 2 below.
   Let the thread intersections be of sizes $n_1, \ldots, n_i$, respectively. ($\sum_{j=1}^{i} n_j \leq n-1$.) Let T_{jl} be the minimal subtree of T_j rooted at x_j and containing exactly the leaves in $T_j \cap S_l$. In the next level of the recursion, x_j has only the single child appearing in the chosen thread. The role of r will be played by x_j. We continue using only the nonroot leaves of the tree as our counting basis, with the root never contributing.
   construct i tuples $\langle T_{11}, \ldots, T_{k1}\rangle, \ldots, \langle T_{1i}, \ldots, T_{ki}\rangle$.
   recursively find the MIT of each of the i tuples.
3. The MIT will be composed of putting together the MITs of different thread intersections. Since our trees are binary trees, a choice of thread σ forces at most one possibility for its counterpart, the thread $\bar{\sigma}$ such that $\bar{\sigma}[i] = 3 - \sigma[i]$. Note that $\bar{\sigma}$ may be empty either because of an empty intersection or even because some x_i has only a single child.
   Let $MIT(\sigma) = MIT(T_{1t}, T_{2t}, \ldots, T_{kt})$ where $S_t = \cap_{l=1}^{k} L(x_{l\sigma[l]})$.
   Pair all threads into $\{\langle \sigma_1, \bar{\sigma_1}\rangle, \ldots, \langle \sigma_{i'}, \bar{\sigma_{i'}}\rangle\}$ where each pair has at least one thread with a nonempty intersection. It is clear that $i' \leq i$. (Recall that i is the number of nonempty disjoint thread intersections found in step 2.)
   Each $MIT(\sigma)$ is a binary tree whose root has a single child. Let $MIT(\langle \sigma, \bar{\sigma}\rangle)$ be the tree resulting from merging the roots of $MIT(\sigma)$ and $MIT(\bar{\sigma})$ into a single node that has two children. This node is the single child of r.
   choose $MIT(T_1, \ldots, T_k)$ as the largest of $\{MIT(\langle \sigma_1, \bar{\sigma_1}\rangle), \ldots, MIT(\langle \sigma_{i'}, \bar{\sigma_{i'}}\rangle)\}$.

end Algorithm

Correctness. The algorithm's main action takes place in step 2. The fact that the MIT is an isomorphic subtree of all trees means that each of $x_1^1, x_2^1, \ldots, x_k^1$ is the child of r in the MIT restricted to $T_1, \ldots, T_k$, respectively. The MIT is then the best combination of children of the x_i's. The partition property of the thread intersections limits the number of recursive cases that need to be considered to $i < n$ rather than 2^k since all other thread intersections are necessarily empty.

Step 3 is correct because in a binary tree every choice of right or left forces a single remaining option. □

Implementation of step 2. Our goal is to decide in polynomial time which of the 2^k possible threads gives a nonempty thread intersection. When constructing a thread, at each of the k trees we make one choice. Either we choose a 1 (if x_{i1} is in the thread) or a 2 (x_{i2} in the thread). Constructing all these choices is equivalent to a breadth-first-search of the complete depth-k binary tree where each node has a

left son labeled 1 and a right son labeled 2. Visiting child e, $e \in \{1,2\}$ at level d means adding x_{de} to the thread. We will in fact be intersecting with $L(x_{de})$. We are guaranteed that the nonempty sets are disjoint at every node at every fixed level. We do not continue the search at any branch where the empty set is reached. The sum of all elements in the sets for each level is at most $n-1$, thus the total time to construct step 2 is $O(kn)$.

Algorithm time. The time for computing the MIT of k rooted trees is given by the following recurrence:

$$f(1) = 1,$$
$$f(n) = f(n_1) + f(n_2) + \cdots + f(n_i) + kn, \quad \text{where } \sum_{l=1}^{i} n_l = n.$$

The closed form is $f(n) = O(kn^2)$.

Total algorithm time. Since the rooted MIT algorithm has to be run n times, the total time is $O(kn^3)$.

It is easy to see that the above algorithm will also work for trees where T_1 has degree bounded by $d > 2$. The only use we made of the fact that the trees are binary was in step 3. In the binary tree case, choosing a σ fixes $\overline{\sigma}$ for purposes of recursively constructing an MIT. Now, however, we need to make sure that we pick the maximum of the choice of all nonconflicting threads, where a thread σ_1 conflicts with a thread σ_2 of the same nodes if $\exists i$ such that $\sigma_1[i] = \sigma_2[i]$. Obviously, two conflicting threads cannot be joined together in the same agreement subtree. There are going to be $\binom{n}{d-1}$ possibilities. The nonconflicting threads may be put together in a brute-force fashion. Each of the $\binom{n}{d-1}$ possibilities requires $O(kd)$ time for verification of nonconflict (since there are at most $d-1$ threads in a combination, and each thread is of length k). Thus all nonconflicting threads can be generated in time $O(k(d-1)n^{d-1}/(d-1)!) = O(kn^{d-1})$. We now choose the best MIT over all nonconflicting thread combinations.

The recurrence is now

$$f(1) = 1,$$
$$f(n) = f(n_1) + f(n_2) + \cdots + f(n_i) + kn^{d-1}, \quad \text{where } \sum_{l=1}^{i} n_l = n.$$

Thus the total algorithm time for degree bounded by d trees is $O(kn^{d+1})$. It should be noted that the algorithm does not require all trees to have bounded degree. It will work even for the case where only *one* tree has bounded degree d. However, if all trees have degree bounded by d, then each set of $d-2$ threads forces a unique $(d-1)$th thread (as in the binary tree case above) and the total algorithm time is then $O(kn^d)$.

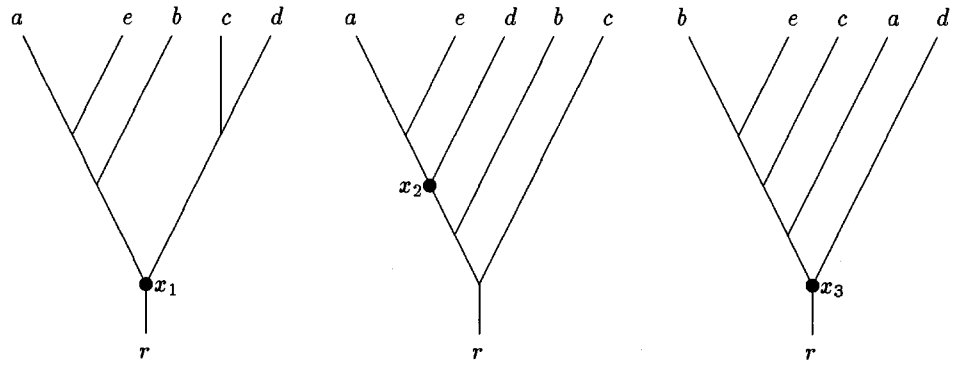
5. Maximum homeomorphic agreement subtree. We now consider the task of finding an MHT subtree of k trees, where the degree of at least one of the trees is bounded.

We follow the same ideas as before. We basically use a naive brute-force algorithm that normally produces an exponential time solution. However, we will define special sets of labels—*maximal decomposable sets*—and show that considering these sets alone is sufficient for finding an MHT. Since there is only a polynomial number of such sets, our algorithm's time is therefore polynomial.

Before presenting the algorithm, we need some more definitions and observations.

Notation. Let $A \subseteq S$. We denote the tree $MHT(H(T_1, A), \ldots, H(T_k, A))$ by $MHT(A)$. To avoid unnecessary notation, we will also use the notation $MHT(A)$ to denote the set of labels of $MHT(A)$.

DEFINITION. *A set $A \subseteq S$ is* decomposable *if there exist $x_1 \in T_1, \ldots, x_k \in T_k$ such that each x_i, $i = 1, \ldots, k$, has at least p children which can be labeled*



$\{a, e, d\}$ is a maximal decomposable set that decomposes to $\{a, e\} \cup \{d\}$.
The roots of the decomposition are x_1, x_2, x_3.

FIG. 3. *Example of a decomposable set.*

$x_{i1}, \ldots, x_{ip}$, $p \geq 2$, *and such that* $A = A_1 \cup \cdots \cup A_p$, *where* A_j *are mutually disjoint nonempty sets and* $A_j \subseteq L(x_{ij})$, $j = 1, \ldots, p$; $i = 1, \ldots, k$. *We say that* A decomposes to $A_1 \cup \cdots \cup A_p$, *and that* $x_1, \ldots, x_k$ *are the* roots of the decomposition.

In other words, there is a node in each one of the trees, where the label set is split among p children in the same fashion. See Figure 3.

Note that the number of decomposable sets may still be exponential. Consider as an example two identical complete binary trees. Every label subset with more than two leaves is a decomposable set. We would like to limit the number of label sets that need to be considered.

DEFINITION. *Let* A *be a decomposable set which decomposes to* $A_1 \cup \cdots \cup A_p$. A *is a* maximal decomposable *set if there does not exist a decomposable set* B *such that* A *is a proper subset of* B *and such that* B *decomposes to* $B_1 \cup \cdots \cup B_q$, $q \geq p$ *where every* A_i, $i = 1, \ldots, p$, *is a subset of a distinct* B_j.

An alternate way of viewing maximal decomposable sets is by considering the roots of the decomposition. A set of nodes $x_1 \in T_1, \ldots, x_k \in T_k$ defines a set of maximal decomposable sets. Each of these maximal decomposable sets is defined by a maximum p ($p \geq 2$) and an ordered list of p children of each x_i, which we will label $x_{i1}, \ldots, x_{ip}$ (although note that the order of these nodes is not necessarily their left-to-right order in the trees). If $\cap_{i=1}^{k} L(x_{ij}) \neq \emptyset$, $j = 1, \ldots, p$, then $\cup_{j=1}^{p} \cap_{i=1}^{k} L(x_{ij})$ is a maximal decomposable set.

The maximality, then, is the fact that this set is the largest decomposable set defined by the given set of nodes. This limitation reduces the number of sets from exponential to polynomial, as can be seen from the following theorem.

THEOREM 7. *Let* $T_1, \ldots, T_k$ *be a set of trees where* T_1 *has degree bounded by* d. *Assume the trees are rooted at a common label* r. *Then there are* $O(n^{d-1})$ *maximal decomposable sets.*

Proof. We know that T_1 is a degree d tree, so in our case, no set can decompose to more than $d-1$ subsets. Observe that if A decomposes to $A_1 \cup \cdots \cup A_{d-1}$, then for every $(d-1)$-tuple $\langle a_1, a_2, \ldots, a_{d-1} \rangle$, where $a_i \in A_i$, $i = 1, \ldots, d-1$, the lowest common ancestor of the $d-1$ elements in each of the k tree is the node defined as the root of the decomposition in that tree. If, however, we are considering a maximal decomposable set, then each such tuple uniquely defines the set. The reason is

that the LCA of the tuple finds the root of the decomposition, and the maximum subset in each of the $d-1$ children that appears in all k trees is the maximal decomposition. □

The following observation guarantees the polynomial time of our algorithm.

Observation 8. If $A \subseteq S$ is a maximal decomposable set decomposable to $A = A_1 \cup \cdots \cup A_p$, then $MHT(A) = MHT(A_1) \cup \cdots \cup MHT(A_p)$. If A is not decomposable, then $MHT(A) = \max\{MHT(B) | B \subseteq A$ and B is maximal decomposable$\}$.

Proof. Let A be maximal decomposable, and let H be the homeomorphic restriction of T_i to $A \cup \{r\}$. The root of H is r. Each x_i, the root of the decomposition in T_i, is the restriction of the only child of r in H. It is clear that the only chance of getting a nonempty MHT is by matching equal subsets to each other. The maximum number we can get is by matching *all* equal subsets to each other.

If A is not maximal decomposable then for each $i,\ i = 1, \ldots, k$, let x_i be the node in T_i that is the restriction of r's child in the MHT. Because $x_1, \ldots, x_k$ appear in the MHT then they are either leaves or each has children $x_{i1}, \ldots, x_{ip},\ p > 1$, where $L(x_{ij}) \cap L(x_{lj}) \neq \emptyset$. Thus if B is the maximal decomposable set of A for nodes $x_1, \ldots, x_k$, then $MHT(A) = MHT(B)$ and $B \subseteq A$. □

The above observation means that it is sufficient to compute the MHT of maximal decomposable sets.

We now present an algorithm for finding the MHT of a set of trees where tree T_1 (which we may assume has the smallest degree) has degree bounded by d. As in the previous sections we present the part that assumes root r. The general algorithm chooses as the MHT the largest tree produced by each of the rooted cases.

For simplicity of the presentation, we will separate the algorithm into two parts. One part that constructs all maximal decomposable subsets of $S - \{r\}$, and a second part that constructs the MHT of all maximal decomposable subsets.

Algorithm for Constructing all Maximal Decomposable Subsets (degree bounded by *d* tree)

For each $(d-1)$-tuple $\langle a_1, \ldots, a_{d-1} \rangle$ of elements from $S - \{r\}$ do:

1. For every tree $T_i,\ i = 1, \ldots, k$ find if there is $x_i \in T_i$ that has $d-1$ distinct children $x_{i1}, \ldots, x_{i(d-1)}$ where $a_j \in L(x_{ij}),\ i = 1, \ldots, d-1$.
2. If such an x_i is found for every $T_i,\ i = 1, \ldots, k$ then construct the maximal sets $A_1, \ldots, A_{d-1}$ such that $a_j \in A_j \subseteq L(x_{ij}),\ i = 1, \ldots, k;\ j = 1, \ldots, d-1$. Take $A = A_1 \cup \cdots \cup A_{d-1}$ as a maximal decomposable set.

end

end Algorithm

Time. The algorithm loops n^{d-1} times. It takes time $O(kn)$ to find the x_i's (step 1) and to construct the A_i's (step 2). The total time is $O(kn^d)$.

Correctness. We have correctness by Theorem 7.

We are now ready to present the algorithm for finding the MHT of k trees one of which has degree bounded by d.

Algorithm for Rooted MHT (degree bounded by *d* tree)

1. Construct all n^{d-1} maximal decomposable sets.
2. Order the maximal decomposable sets by the subset relation.
3. The base subsets (those that have no maximal decomposable subsets) are sets $A = A_1 \cup \cdots \cup A_{d-1}$ where each of the A_i's, $i = 1, \ldots, d-1$, is a singleton set $\{a_i\}$. $MHT(A)$ is A.
   For subsequent maximal decomposable subsets, construct the MHT by the relation $MHT(A) = MHT(A_1) \cup MHT(A_2) \cup \cdots \cup MHT(A_{d-1})$. For any maximal

decomposable A_i, we have already computed the MHT. If any of the A_i is not maximal, its MHT is that of its largest maximal decomposable subset.
$MHT(S)$ is $MHT(A)$, where A is the largest maximal decomposable subset of S.

end Algorithm

Time. Step 1 was already seen to take time $O(n^d)$. For steps 2 and 3, there are $O(n^{d-1})$ maximal decomposable sets. Comparing every two sets takes time $O(n)$. A straightforward implementation of steps 2 and 3 is by a pairwise comparison of all maximal decomposable sets. This can be done in time $O(n^{2d-1})$. The total algorithm time for rooted trees is then $O(kn^d + n^{2d-1})$. For unrooted trees the time is $O(kn^{d+1} + n^{2d})$.

Correctness. The correctness is clear by Observation 8.

6. Maximum weighted agreement subtree (MWT). An MHT of a set of trees produces a tree whose *shape* is common to all trees. However, this does not take into account the conjectured *time* each evolutionary split took. Our final goal is a tree where the distances between nodes are conjectured, i.e., given as a *range* of numbers. However, we start with the case where the distances between nodes in each of the evolutionary trees are given exactly. This case is a special case of the maximum interval subtree discussed in the next section. However, we treat it separately for two reasons: 1) it introduces some concepts that are used later in the interval case 2) it has a faster algorithm than the general case.

DEFINITION. *Let $T_1, \ldots, T_k$ be S-labeled trees with integer weights on the edges. A* maximum weighted agreement subtree *of $T_1, \ldots, T_k$ ($MWT(T_1, \ldots, T_k)$) is a maximum set $S' \subseteq S$ such that $\forall s, t \in S'$ the distance between s and t (the sum of the weights on the edges of the path between them) is the same in all trees $T_1, \ldots, T_k$. We will also refer to the problem of finding a maximum weighted agreement subtree as* the MWT problem.

This is a generalization of the MIT problem (there every edge had weight 1). However, we cannot immediately reduce the MWT problem to the MIT problem by replacing every weight-i edge by i weight-1 edges, since then the problem size may grow exponentially.

Let T_1 and T_2 be two S-labeled trees with integer weights on the edges. T_1 and T_2 are *w-isomorphic* if the distance between every two labeled edges is the same in both trees.

THEOREM 9. *Two labeled trees with no* 2-vertices *(vertices of degree* 2*) are w-isomorphic iff they are isomorphic as leaf-labeled trees and corresponding edges have the same weight.*

Proof. For rational weights, this is an immediate corollary of Theorem 4. □

Theorem 9 asserts that in different trees of the given set, subtrees corresponding to the MWT may differ only in edges incident to 2-vertices. In addition, the sum of the weights of a maximal chain of 2-vertices (whose end points are not 2-vertices) must be the same for all corresponding pairs of non-2-vertices in all the trees.

For the sake of exposition we again assume that all our trees are binary trees. The algorithm outline is generally unchanged from the MIT case. The only exceptions are the following necessary changes.

- As in the MIT case, the recursion always treats rooted trees where the root has a single child x and that child has two children x_1 and x_2. Unlike the MIT case, our root is *always* taken to be r and the weight of the edge from r to x is the sum of the weights from r to x in the original tree.

- If the children of r in all the trees have the same distance to r the case is handled exactly like the MIT algorithm.
- If there are at least two children of r with different distances to r, let i be the tree with the smallest distance $d(x_i, r)$. Because of Theorem 9, it is impossible for x_i to be in an MWT as a 3-vertex. Therefore, we split the tree $T_i(x_i)$ into two trees $T_i(x_{i1})$ and $T_i(x_{i2})$, where x_{i1} and x_{i2} are x_i's children. We now have two MWT problems of smaller size.

Termination. The algorithm terminates since a tree split always reduces the size of the remaining trees. Since there are kn nodes initially, and every split reduces the size of the trees by at least one node, there can be no more than kn splits.

Time. The time complexity of the algorithm does not change. Although we may split the problem several times during the course of the algorithm, the sizes of the problems are reduced with each split. Because of convexity of the time function, the complexity is worst when there is no split. This is exactly the MIT case.

Total algorithm time. $O(kn^3)$.

Total algorithm time for degree bounded by d trees. $O(n^d + kn^3)$.

7. Maximum interval agreement subtree. An MHT of k trees provides, in a sense, the greatest number of leaves whose underlying tree structure has a similar shape. An MWT provides a maximum set of leaves where the exact distances between them are preserved. The big problem is that the exact distances are not usually known. We would like a metric that gives us the advantage of both structure and "approximate" distance. We approximate the distance between two nodes by a pair of numbers $[a, b]$, where $a \leq b$. This pair represents the interval between a and b and its meaning is that the distance between the edge's head and tail is within that interval. We are interested in the largest set of leaves that has an underlying tree with edge weights in the intersection of all appropriate intervals. Formally we have the following definition.

DEFINITION. *Let $T_1, \ldots, T_k$ be S-labeled trees. Assume that every edge is labeled by a pair of numbers $[a, b]$, where $a \leq b$. We call such trees* S-labeled interval labeled trees. *An* instantiation *T' of an S-labeled interval labeled tree T is an S-labeled weighted tree (as defined in section 6) where all nodes and edges of T' are equal to the nodes and edges of T and where every edge of T' is labeled by a number in the interval label of that edge in T.*

A maximum interval agreement subtree *of $T_1, \ldots, T_k$ ($MIWT(T_1, \ldots, T_k)$) is a maximum set $S' \subseteq S$ such that $MIWT(T_1, \ldots, T_k)$ is $MWT(T'_1, \ldots, T'_k)$, where T'_i is an instantiation of T_i for $i = 1, \ldots, k$. We refer to the problem of finding a maximum interval agreement subtree as* the MIWT problem.

Note that all previous agreement metrics are special cases of MIWT. In MIT, we assume all intervals are $[1, 1]$. In MWT, any edge weight a is interval $[a, a]$. In MHT, we assume all intervals are $[1, n-1]$. We presented the case separately for historical and methodological reasons.

The key observation for the MIWT problem is that Theorem 9 guarantees that a MIWT of k trees is also a homeomorphic agreement subtree of those trees (because it is an MWT of some instantiation of them). Once this observation is made the following algorithm follows naturally. Before presenting the algorithm, we need one more definition.

DEFINITION. *The* distance interval *between node x and y in an interval labeled tree is the interval $[z, w]$ where z is the sum of the smaller elements of each of the pairs on the path from x to y, and w is the sum of the larger elements.*

Algorithm for MIWT

We modify the MHT algorithm to return MIWT. The only necessary modification is the following:

1. For each homeomorphic agreement subtree T' handled by the algorithm do
   (a) for each tree T_i, $i = 1, \ldots, k$ do
   for every path in T_i corresponding to an edge in T' compute the distance interval of that path
   (b) { Every edge in T' now has k distance intervals associated with it. }
   for every edge in T' where the k distance intervals have a nonempty intersection, discard the edge and the entire subtree rooted to its head

end Algorithm.

THEOREM 10. *The above algorithm computes a MIWT of k trees in time $O(kn^{d+1} + n^{2d})$, where there is at least one tree of degree bounded by d.*

Proof. Correctness follows immediately from the observation that every MIWT is also a homeomorphic agreement subtree.

We need to prove the time bounds. The added modification can be implemented in time $O(kn)$ and needs to be added for every maximal decomposable set. Since there are $O(n^d)$ such sets, the asymptotic time complexity of the algorithm does not change, and is $O(kn^{d+1} + n^{2d})$. □

8. Future work. Although the $\mathcal{NP}$-completeness proof of the unbounded degree case prepares us for having the degree d as an exponent in our time complexity, it would be nice to reduce the time for finding the MWT to $O(c^d p(n, k))$, where p is a polynomial, rather than $O(n^d)$.

We presented here an approximation for the problem of finding the complement of the MHT. In [13] it was proven that the MHT problem for three trees with unbounded degree cannot be approximated within ratio n^ϵ for any constant $\epsilon \leq 1$.

For the case of bounded degree trees, the time was improved by [5].

From a graph theoretic aspect, it would be interesting to find a maximum agreement subtree in the sense of maximizing the number of edges, while using true contraction. The number is no less than n because every tree can be contracted to a star. However, again in [13] it was shown that this prolem is $\mathcal{NP}$-hard even for two trees.

An ambitious problem is to define "tree closeness" metrics and efficiently find the closest tree to a set of trees keeping all the leaves.

Acknowledgments. We wish to thank Mike Steel for his great help in refining this paper, Bob Robinson for kindly sending us reference material on the subject, and two anonymous referees who were very fast and insightful.

REFERENCES

[1] A. AMIR AND D. KESELMAN, *Maximum agreement subtree in a set of evolutionary trees - Metrics and efficient algorithms*, in Proc. 35th IEEE Symposium on Foundations of Computer Science, Santa Fe, NM, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 758–769.

[2] H. J. BANDELT AND A. DRESS, *Reconstructing the shape of a tree from observed dissimilarity data*, Adv. Appl. Math, 7 (1986), pp. 309–343.

[3] W. H. E. DAY, *Optimal algorithms for comparing trees with labeled leaves*, J. Classification, 2 (1985), pp. 7–28.

[4] M. FARACH, S. KANNAN, AND T. WARNOW, *A robust model for finding optimal evolutionary trees*, in Proc. 25th Annual ACM Symposium on the Theory of Computing, San Diego, CA, ACM, New York, 1993, pp. 137–145.

[5] M. Farach, T. M. Przytycka, and M. Thorup, *On the agreement of many trees*, in Proc. 3rd European Symposium on Algorithms, Corfo, Greece, 1995, pp. 381–393.
[6] M. Farach and M. Thorup, *Fast comparison of evolutionary trees*, in Proc. 5th ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, SIAM, Philadelphia, 1994, pp. 481–488.
[7] C. R. Finden and A. D. Gordon, *Obtaining common pruned trees*, J. Classification, 2 (1985), pp. 255–276.
[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.
[9] W. Goddard, E. Kubicka, G. Kubicki, and F. R. McMorris, *Agreement subtrees, metric and consensus for labeled binary trees*, DIMACS Workshop on Partitioning Data Sets, Rutgers University, New Brunswick, NJ, 1993.
[10] A. D. Gordon, *A measure of the agreement between rankings*, Biometrika, 66 (1979), pp. 7–15.
[11] A. D. Gordon, *On the assessment and comparison of classifications*, in Analyse de Données et Informatique, R. Tomassone, ed., INRIA, 1980, pp. 149–160.
[12] J. A. Hartigan, *Clustering algorithms*, John Wiley, New York, 1975.
[13] J. Hein, T. Jiang, L. Wang, and K. Zhang, *On the Complexity of Comparing Evolutionary Trees*, in Proc. 6th Symposium on Combinational Pattern Matching, Helsinki, Finland, 1995, pp. 177–190.
[14] E. Kubicka, G. Kubicki, and F. R. McMorris, *An algorithm to find agreement subtrees*, J. Classification, 1992.
[15] D. A. Neumann, *Faithful consensus methods for n-trees*, Bull. Math. Biol., 63 (1983), pp. 271–287.
[16] E. A. Smolenskii, Jurnal Vicisl. Mat. i Matem. Fiz, 2 (1962), pp. 371–372.
[17] M. Steel and T. Warnow, *Kaikoura tree theorems: Computing the maximum agreement subtree*, Inform. Proc. Lett., 48 (1993), pp. 77–82.
[18] M. A. Steel and D. Penny, *Distributions of tree comparison metrics - Some new results*, Syst. Biol., 42 (1993), pp. 126–141.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1670–1688, December 1997

006

THE UNION OF CONVEX POLYHEDRA IN THREE DIMENSIONS*

BORIS ARONOV†, MICHA SHARIR‡, AND BOAZ TAGANSKY§

Abstract. We show that the number of vertices, edges, and faces of the union of k convex polyhedra in 3-space, having a total of n faces, is $O(k^3 + kn \log k)$. This bound is almost tight in the worst case, as there exist collections of polyhedra with $\Omega(k^3 + kn\alpha(k))$ union complexity. We also describe a rather simple randomized incremental algorithm for computing the boundary of the union in $O(k^3 + kn \log k \log n)$ expected time.

Key words. combinatorial geometry, computational geometry, combinatorial complexity, convex polyhedra, geometric algorithms, randomized algorithms

AMS subject classifications. 52B10, 52B55, 65Y25, 68Q25, 68U05

PII. S0097539793250755

1. Combinatorial bounds.

Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ be a family of k convex polyhedra in 3-space, let n_i be the number of faces of P_i, and let $n = \sum_{i=1}^{k} n_i$. Put $U = \bigcup \mathcal{P}$. By the *combinatorial complexity* of a polyhedral set we mean the total number of its vertices, edges, and faces. Our main result is the following.

THEOREM 1.1. *The combinatorial complexity of the union U is $O(k^3 + kn \log k)$. This bound is almost tight in the worst case, since there are examples where the complexity of such a union is $\Omega(k^3 + kn\alpha(k))$.*

1.1. Background.

This result extends the known sharp bound of $\Theta(k^2 + n\alpha(k))$ on the complexity of the union of k convex polygons in the plane with a total of n edges [7]. It is interesting to note that in both cases the bounds depend only linearly on n.

Our result has several applications, mentioned below, to robot motion planning and to problems in geometric optimization. It is a natural special case of the problem of analyzing the complexity of the union of geometric objects, which is formulated for arrangements of more general curves and surfaces in two and three dimensions, respectively. This problem has received considerable attention recently and has been studied mostly in the plane. Several special cases have been identified where sharp complexity bounds can be established, such as the cases of "pseudodisks" [27] or of "fat" triangles [28]. In three dimensions, however, very few sharp bounds for the complexity of the union of objects are known. One such bound is for the union of n

* Received by the editors June 23, 1993; accepted for publication (in revised form) November 27, 1995. A preliminary version of this paper appeared in the Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science, (FOCS), IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 518–527.

http://www.siam.org/journals/sicomp/26-6/25075.html

† Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201 (aronov@ziggy.poly.edu). This author was supported by National Science Foundation grant CCR-92-11541.

‡ School of Mathematical Sciences, Tel Aviv University, Tel Aviv, 69978, Israel (sharir@math.tau.ac.il) and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012. This author was supported by National Science Foundation grant CCR-91-22103 and by grants from the U.S.–Israeli Binational Science Foundation, the German Israeli Foundation for Scientific Research and Development, and the Fund for Basic Research administered by the Israeli Academy of Sciences.

§ School of Mathematical Sciences, Tel Aviv University, Tel Aviv, 69978, Israel.

balls, where a $\Theta(n^2)$ worst-case bound is easy to establish. Another recent bound is for the union of n axis-parallel cubes in any dimension $d \geq 2$. The worst-case bound is $\Theta(n^{\lceil d/2 \rceil})$, and it improves to $\Theta(n^{\lfloor d/2 \rfloor})$ when the cubes have all the same size [11].

The union of geometric objects (or, rather, its complement) is among several important substructures in the arrangement of the objects, such as their lower (or upper) envelope, a single cell of the complement (a "hole" in the union), a subset of cells in the complement, and the collection of all nonconvex or otherwise "interesting" cells of the complement. Considerable progress has recently been made in the analysis of these substructures; see [5, 7, 22, 21, 23, 24, 33, 36].

We also obtain an efficient randomized incremental algorithm for computing the union of k convex polyhedra with a total of n faces, whose expected running time is $O(k^3 + kn \log k \log n)$, and is thus close to optimal in the worst case. Our algorithm computes the portion of the boundary of the union contained in each of the faces of the polyhedra, using a separate randomized incremental procedure for each face. The algorithm and its analysis are adapted from previously known techniques ([13, 16, 20, 30] and others), but it introduces a significant observation, namely that, even without a global randomized insertion mechanism (which is not applicable in our case), sharp expected complexity bounds can still be obtained. This may be of independent interest and may have further applications for efficient construction of other three- (or higher-)dimensional structures. Another complication arises since the objects that we add incrementally do not necessarily have constant complexity, so handling them requires a more complex version of the algorithm and of its analysis.

1.2. Analysis. We first simplify the analysis by assuming that the given polyhedra are in *general position*, meaning that no point is common to the boundaries of any four distinct polyhedra, no vertex of one polyhedron lies on the boundary of another, no two edges of distinct polyhedra meet, and no edge of a polyhedron meets the polygonal curve of intersection of the surfaces of any two other polyhedra. We claim that this assumption involves no loss of generality. Indeed, Lemmas 1.2 and 1.3 below, which are easily seen to hold also when the given polyhedra are not in general position, imply that it suffices to prove the asserted bound for the number of vertices of the union that are formed as the intersection of faces of three (or more) distinct polyhedra. However, it can be easily verified that, if we perturb the vertices of the given polyhedra by sufficiently small displacements, so as to move them into general position, the number of such union vertices does not decrease. Moreover, if the perturbation is sufficiently small, the number of edges and faces of the union also cannot decrease. We can then charge each edge or face of the union to an incident vertex in the perturbed collection and argue that no such vertex is charged more than a constant number of times. These considerations imply that it suffices to establish the bound of Theorem 1.1 for the number of *vertices* of the union of collections $\mathcal{P}$ in general position.

We begin with a derivation of a few simple auxiliary results.

LEMMA 1.2. *The sum of the numbers of vertices, edges, and faces of the pairwise intersections $P_i \cap P_j$, over all $1 \leq i < j \leq k$, is $O(kn)$.*

Proof. $P_i \cap P_j$ is a convex polyhedron bounded by at most $n_i + n_j$ faces, and thus has $O(n_i + n_j)$ vertices and edges. Summing this bound over all (i, j) with $1 \leq i < j \leq k$, we obtain $O(kn)$. □

Define the *arrangement* $\mathcal{A}(\mathcal{P})$ of the collection $\mathcal{P}$ as the decomposition of space into vertices, edges, faces, and three-dimensional cells induced by the faces of the polyhedra of $\mathcal{P}$; for more details on arrangements, see [4, 17, 34].

LEMMA 1.3. *The number of vertices of $\mathcal{A}(\mathcal{P})$, other than those formed as the intersection of faces of three or more distinct polyhedra, is $O(kn)$.*

Proof. Each such vertex is either a vertex of a polygon in $\mathcal{P}$, or a vertex of $P_i \cap P_j$, for some $i \neq j$. The claim now follows from Lemma 1.2. □

LEMMA 1.4 (Aronov, Bern, and Eppstein [1]). *The overall complexity of $\mathcal{A}(\mathcal{P})$ is $O(k^2 n)$, which is tight in the worst case.*

Proof. Each vertex of $\mathcal{A}(\mathcal{P})$ not counted in Lemma 1.3 is the intersection of an edge of some intersection $P_i \cap P_j$ with a face of another polyhedron P_ℓ. Since Lemma 1.2 implies that the total number of such edges is $O(kn)$, and each of them crosses the surface of another polyhedron at most twice, the upper bound follows. For an easy lower bound construction, see [1]. □

Proof of Theorem 1.1. We prove the theorem by induction on k. For $k \leq 3$ the claim follows trivially from Lemma 1.4. Recall that $\mathcal{A}(\mathcal{P})$ is the arrangement in $\mathbb{R}^3$ of the collection of the n facets of the polyhedra in $\mathcal{P}$. An m-face $f \in \mathcal{A}(\mathcal{P})$, for $m = 0, 1, 2$, is a *level-z* face if exactly z polyhedra in $\mathcal{P}$ contain f in their interior. Thus the faces of the union are precisely the level-0 faces.

A vertex $v \in \mathcal{A}(\mathcal{P})$ is said to be an *outer vertex* if it is incident to an edge of a polyhedron in $\mathcal{P}$. Otherwise v is called an *inner vertex*. By Lemma 1.3 the number of outer vertices is $O(kn)$. Thus our main goal is to bound the number of level-0 inner vertices. Denote by $C_z(\mathcal{P})$ the number of level-z inner vertices of $\mathcal{A}(\mathcal{P})$, and by $C_z(k, n)$ the maximum of $C_z(\mathcal{P})$ over all sets $\mathcal{P}$ of k convex polyhedra in general position, with a total of n facets.

The triple (f, e, e') is said to be a *special triple* if f is a level-1 2-face of $\mathcal{A}(\mathcal{P})$, e and e' are edges of f that are level-0 edges of $\mathcal{A}(\mathcal{P})$, and we can trace the boundary of f from e to e' without passing through any other level-0 edge. (Note that e and e' must both lie on the boundary of the unique polyhedron containing f in its interior and thus on the outer boundary component of f.) In this notation, the order of e and e' is immaterial, and we identify (f, e, e') with (f, e', e). Lemma 1.5 below, a main technical step of the proof, shows that the number of such special triples is $O(k^3 + kn \log k)$. We also define a *special triangle* to be any level-1 triangular face of $\mathcal{A}(\mathcal{P})$, with one level-0 edge and two level-1 edges.

Let v_0 be a level-0 inner vertex. Then v_0 is the intersection of three facets F_1, F_2, F_3 of three respective distinct polyhedra P_1, P_2, P_3. The vertex v_0 is incident to three level-0, six level-1, and three level-2 2-faces of $\mathcal{A}(\mathcal{P})$. Let f be one of the incident level-1 faces, say the one contained in $F_1 \cap P_3$ and lying outside P_2. There are two edges of f incident to v_0: a level-0 edge e_0 within $F_1 \cap F_3 \setminus P_2$, and a level-1 edge e_1 within $F_1 \cap F_2 \cap P_3$ (see Figure 1). Let $e_0, v_0, e_1, v_1, e_2, v_2, e_3$ denote the edges and vertices of the (outer) boundary of f in the order of their appearance along the boundary, starting from e_0 (see Figure 1). Note that we may have $e_3 = e_0$.

We now introduce a charging scheme, in which v_0 can be charged to a "nearby" vertex or other feature of the arrangement $\mathcal{A}(\mathcal{P})$. One of the following five cases must arise:

(i) v_1 is an outer vertex of $\mathcal{A}(\mathcal{P})$. In this case we charge one unit to v_1.

(ii) v_1 is a level-0 inner vertex, and thus (f, e_0, e_2) is a special triple. We charge one unit to (f, e_0, e_2).

(iii) v_1 is a level-1 inner vertex, and v_2 is not a level-0 inner vertex. We charge $1/3$ of a unit to v_1.

(iv) v_1 is a level-1 inner vertex, v_2 is a level-0 inner vertex, and $e_0 \neq e_3$. We charge one unit to the special triple (f, e_0, e_3).

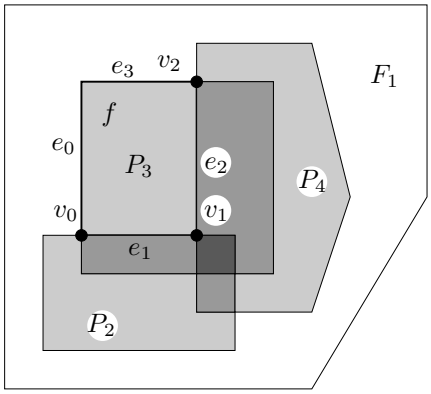


FIG. 1. *The level-*1 *face* f *used for the charging scheme*; *here case* (iv) *is illustrated.*

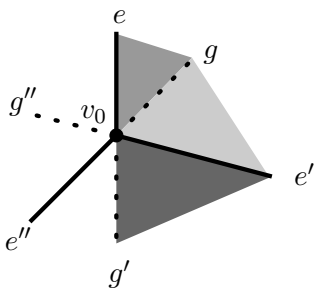


FIG. 2. *Many special triangles incident to* v_0.

(v) v_1 is a level-1 inner vertex, v_2 is a level-0 inner vertex, and $e_0 = e_3$. We charge $1/6$ of a unit to the vertex v_1 and $1/6$ of a unit to the special triangle $\triangle e_0 e_1 e_2$.

If we repeat the above procedure for each of the six level-1 2-faces incident to v_0, the vertex v_0 will receive at least 2 units of charge, at least $1/3$ for each 2-face. Moreover, we claim that v_0 can be incident to at most four special triangles (as in case (v)): Note that v_0 is incident to three level-0 edges e, e', e'', and to three level-1 edges g, g', g'', and each special triangle incident to v_0 is incident to one of these level-0 edges and to one of these level-1 edges. So suppose to the contrary that v_0 is incident to five or more special triangles. It follows that there are three consecutive special triangles incident to v_0, i.e., up to symmetric configurations, one special triangle is incident to e and g, one is incident to g and e', one is incident to e' and g', and e and g' must be collinear; see Figure 2. By definition, the other endpoints of e and g must lie in some original polyhedron facet F_4 not incident to v_0. Similarly, because of the general position assumption, the other endpoints of g and e' must both lie in F_4, and the same holds for the other endpoint of g'. This, however, is impossible, since the line segment $e \cup g'$ has both endpoints on F_4 and contains v_0 which is not on F_4. Hence, v_0 can be incident to at most four special triangles.

It follows that v_0 will receive at least $2 - 4 \times 1/6 = 4/3$ units of charge even if we do not charge the special triangles.

We repeat the charging scheme for all the level-0 inner vertices of $\mathcal{A}(\mathcal{P})$. Each outer vertex w that is charged in this scheme (in type (i) cases) has exactly two incident edges which lie in the intersection of two polyhedra facets. Then w may be

charged up to four times: it can be charged along those two intersection edges, at most twice along each edge, from the two incident level-1 faces, for the total of at most 4 units. Each special triple (f, e, e') may be charged up to four times, once for each vertex of e and e', for the total amount of at most 4 units of charge. Each level-1 inner vertex v is incident to three level-1 2-faces. Within each face it is charged at most $1/3$ of a unit (either in just one charging of type (iii) or in at most two chargings of type (v)). Thus the total charge to v is at most 1 unit. To summarize, in the overall charging scheme, every level-0 inner vertex receives at least $4/3$ units, every level-1 inner vertex pays at most 1 unit, and every outer vertex or special triple pays at most 4 units. This yields:

$$\frac{4}{3}C_0(\mathcal{P}) \le C_1(\mathcal{P}) + O(kn) + O(k^3 + kn\log k) = C_1(\mathcal{P}) + O(k^3 + kn\log k)\,. \tag{1}$$

Let $\mathcal{R} \subset \mathcal{P}$ be a random sample of $k-1$ polyhedra (that is, $\mathcal{R}$ is obtained by deleting at random one polyhedron from $\mathcal{P}$). Arguing as in several recent related works [11, 14, 36], we have

$$\begin{aligned}\frac{k-3+4/3}{k}C_0(\mathcal{P}) &\le \frac{k-3}{k}C_0(\mathcal{P}) + \frac{1}{k}C_1(\mathcal{P}) + \frac{1}{k}O(k^3 + kn\log k)\\ &= \mathbf{E}(C_0(\mathcal{R})) + O(k^2 + n\log k)\,,\end{aligned} \tag{2}$$

where $\mathbf{E}$ denotes expectation with respect to the random sample $\mathcal{R}$. Here we have used (1) and the fact that a level-0 inner vertex of $\mathcal{A}(\mathcal{P})$ remains a level-0 inner vertex of $\mathcal{A}(\mathcal{R})$ with probability $\frac{k-3}{k}$ (this happens if and only if none of the three incident polyhedra is deleted), a level-1 inner vertex of $\mathcal{A}(\mathcal{P})$ turns into a level-0 inner vertex of $\mathcal{A}(\mathcal{R})$ with probability $1/k$ (this happens if and only if the deleted polyhedron is the unique one containing the vertex), and no other vertex of $\mathcal{A}(\mathcal{P})$ can become a level-0 inner vertex in $\mathcal{A}(\mathcal{R})$. Using the induction hypothesis $C_0(k,n) \le ck^3 + ckn\log k$, for some absolute constant c, we obtain

$$\mathbf{E}(C_0(\mathcal{R})) \le \frac{1}{k}\sum_j C_0(k-1, n-n_j) \le c(k-1)^3 + \frac{c(k-1)^2}{k}n\log(k-1)\,.$$

Thus (2) becomes

$$\frac{k-5/3}{k}C_0(\mathcal{P}) \le c(k-1)^3 + \frac{c(k-1)^2}{k}n\log(k-1) + b(k^2 + n\log k)\,,$$

for an appropriate constant b. It is now easy to show that

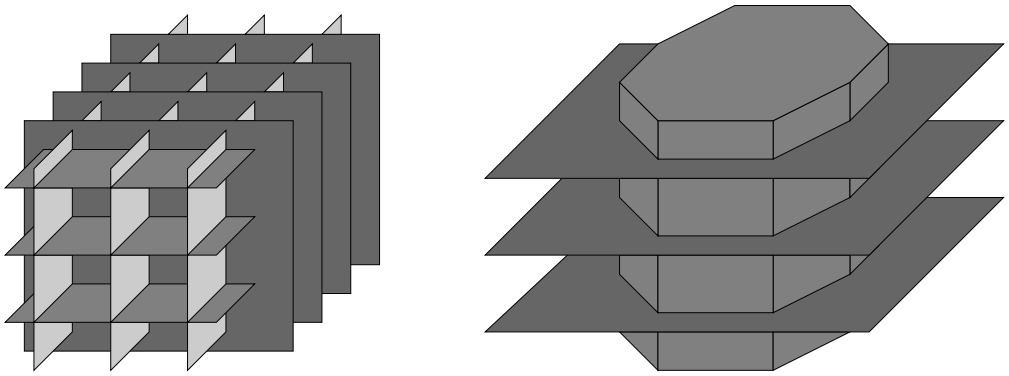
$$c(k-1)^3 + bk^2 \le \frac{k-5/3}{k}\cdot ck^3\,,$$

and that

$$\frac{c(k-1)^2}{k}n\log(k-1) + bn\log k \le \frac{k-5/3}{k}\cdot cnk\log k\,,$$

provided c is chosen sufficiently large. Indeed, the first inequality is trivial to enforce. The second one is implied by the inequality

$$c\left(k-2+\frac{1}{k}\right) + b \le c\left(k-\frac{5}{3}\right)\,,$$



(a) (b)

FIG. 3. *Lower bound construction.*

which is also trivial to enforce for $k > 3$. This completes the induction step and thus establishes the asserted upper bound.

As to the lower bound, $\Omega(k^3)$ is achieved by a gridlike arrangement of almost flat polyhedra (cf. Figure 3(a)). To obtain the second term, take a collection of $k/2$ convex polygons in the xy-plane, with a total of $n - ck/2 - k$ edges, for an appropriate constant c, such that the complexity of their union is $\Omega(n\alpha(k))$. Such a construction is described in [7]. Turn each planar polygon in this construction into a vertical prism, and cut the resulting collection of prisms by $k/2$ parallel horizontal flat polyhedra with c faces each. The union of the resulting collection of k convex polyhedra with a total of n faces has complexity $\Omega(kn\alpha(k))$. General position, if desired, can be achieved by a small perturbation of the planes containing the faces of the polyhedra, without reducing the complexity of the union. Figure 3(b) illustrates a simpler $\Omega(kn)$ construction with a single prism. Combining the two constructions, we obtain a lower bound of $\Omega(k^3 + kn\alpha(k))$, as claimed. □

To finish the proof, next we show the following.

LEMMA 1.5. *The number of special triples in an arrangement $\mathcal{A}(\mathcal{P})$ of k polyhedra with a total of n faces is $O(k^3 + kn \log k)$.*

Proof. Let f be a level-1 face of the arrangement $\mathcal{A}(\mathcal{P})$. Let e_f denote the number of edges of f that lie at level 0. (Note that all these edges must lie on the outer component of ∂f.) Clearly, the number of (unordered) special triples of the form (f, e', e'') is exactly e_f. However, to facilitate our inductive proof, we assign weight $w(f) = \max\{4e_f - 6, 0\}$ to f and proceed to estimate a new quantity, namely $\sum_f w(f)$, where the sum is taken over all level-1 faces f of $\mathcal{A}(\mathcal{P})$. (The reason for replacing e_f by $w(f)$ is, essentially, "induction loading," i.e., strengthening the statement in order to strengthen the induction hypothesis, thereby making the proof simpler.) We denote the maximum value of this quantity, over all collections $\mathcal{P}$ of k polyhedra in general position, with a total of n faces, by $C^{(1)}(k, n)$.

Before proceeding to bound $C^{(1)}$, we observe that it is an upper bound on the number of special triples (f, e, e'). Indeed, we need only consider level-1 faces f that have at least two level-0 edges on their boundary. For such faces f, we have $e_f \geq 2$, and $w(f) = 4e_f - 6 \geq e_f$, so indeed $\sum w(f)$, with the sum taken over all level-1 faces f, bounds the number of special triples in $\mathcal{A}(\mathcal{P})$.

We now estimate $C^{(1)}(k,n)$ by employing a slightly different induction scheme, as used in [2, 3, 5, 19]. That is, we remove a polyhedron P_i, add it back, and estimate the increase in the sum $\sum_f w(f)$, over all level-1 faces f of $\mathcal{A}(\mathcal{P})$, which neither lie on the boundary of P_i nor are contained in its interior. The argument is repeated for all $P_i \in \mathcal{P}$ and the resulting bounds on the increase in $\sum_f w(f)$ are added, to obtain a recurrence for $C^{(1)}$. We note that if f' is such a face in the full arrangement $\mathcal{A}(\mathcal{P})$, then, when P_i is removed, f' may expand, possibly merging with other faces, to form a bigger face f, which is a level-1 face in the reduced arrangement $\mathcal{A}_i = \mathcal{A}(\mathcal{P} \setminus \{P_i\})$.

Thus let f be such a face in $\mathcal{A}_i$. We assume that f is contained in a face F_j of some polyhedron P_j and in the interior of only one other polyhedron P_ℓ, where P_j and P_ℓ are distinct from P_i, and consider what may happen to f when P_i is reinserted. The portion $f \setminus P_i$ of f remains at the first level of the full arrangement, whereas $f \cap P_i$ lies at the second level and so should be ignored. $f \setminus P_i$ may be disconnected and consist of several subfaces $f_1, \ldots, f_u$. We are interested only in situations where $\sum_{q=1}^{u} w(f_q) > w(f)$, for only then does our count go up. Recall that e_f counts the number of level-0 edges of f, i.e., of edges of f on ∂P_ℓ, rather than all edges of f; in particular, we can assume that $e_f \geq 1$, for otherwise neither f nor $f \setminus P_i$ contribute anything to $C^{(1)}$.

Clearly, the cases where P_i contains f in its interior or avoids it altogether are uninteresting. If P_i avoids the outer boundary of f, there is no increase in weight (since all e_f level-0 edges of f lie on that boundary). Thus, from this point on, we assume that ∂P_i does meet the outer boundary of f.

We disregard subfaces f_q for which $w(f_q) = 0$; we are thus interested only in situations where

$$\sum_{w(f_q)>0} (4e_{f_q} - 6) > 4e_f - 6 \,. \tag{3}$$

Each of the at most e_f level-0 edges of f may be split by P_i into at most two subedges that may contribute to the left-hand side of (3), which is thus at most $4e_f + 4e^* - 6u^*$, where e^* is the number of level-0 edges that have been split, with both subedges appearing in positive-weight subfaces, and where u^* denotes the number of subfaces with positive weight. Thus the increase that f contributes to the overall sum of weights is at most $(4e_f + 4e^* - 6u^*) - (4e_f - 6) = 4e^* - 6u^* + 6$ (it can be smaller if $e_f = 1$ or if some of the e_f level-0 edges of f appear only in 0-weight subfaces). Let s be one of the e^* split edges, and let s_1 and s_2 be the two subedges into which s is split. If s_1 and s_2 lie in the same subface f_q of f, then the portion of ∂f_q between s_1 and s_2 must either contain a concave vertex of f_q that is also a vertex of $F_j \cap P_i$ or meet one of the islands (i.e., interior components of the boundary) of f that have become connected to the outer boundary by P_i (cf. the proofs of the Consistency Lemma and the Combination Lemma of [18], and Figure 4). In the former case we charge the splitting to any such concave (i.e., outer) vertex, and in the latter case we charge the splitting to such an island. It is easily seen that any such vertex or island is charged at most once, over all choices of P_i. Notice that the number of these islands, over all choices of a level-1 face $f \subset F_j$ and of P_i, for a fixed face F_j, is k, as it is at most the number of intersections $P_\ell \cap F_j$, for $1 \leq \ell \leq k$. Hence the total number of such charges is $O(kn)$. Similarly, by Lemma 1.3, the total number of concave vertices is also $O(kn)$. Thus the overall number of such edge splittings is $O(kn)$. Note that if an edge splitting of this kind occurs, then only one level-0 edge of ∂f is split by P_i, there is only one subface f_q with positive weight, and the weight increase is at most

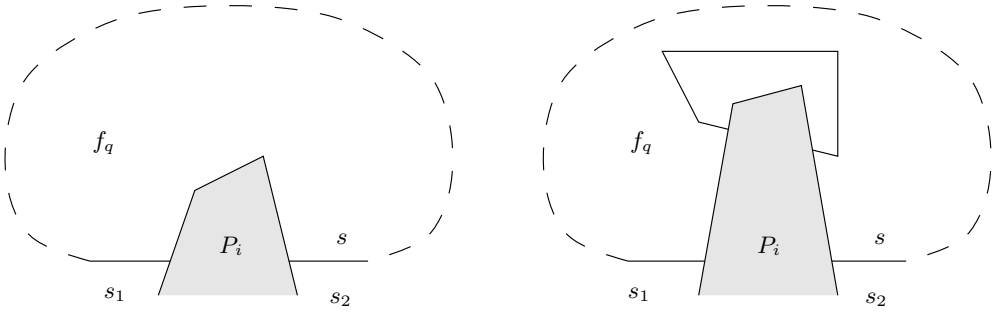


FIG. 4. *Two cases where the subedges of a split edge occur on the boundary of the same subface f_q.*

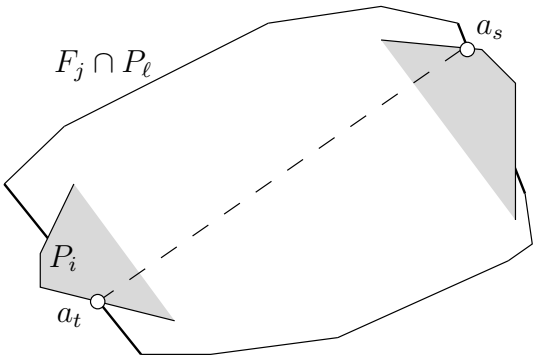


FIG. 5. *Analysis of edges split into different subfaces.*

4 (this follows from the convexity of P_i, F_j, and P_ℓ; see also Figure 4), so the total increase in all the faces in which splittings of this type occur, over all P_i, is $O(kn)$.

It therefore remains to consider the case where s_1 and s_2 lie in different subfaces of f, both having positive weight. Orient s so that f lies on its left, and suppose that s_1 precedes s_2 along s in this direction. Let a_s be the endpoint of s_2 closest to s_1. Label a_s by the subface of f incident to a_s and to s_2. If we trace the outer boundary of f in this direction, we obtain a cyclic sequence of points a_s; their labels form a corresponding cyclic sequence of positive-weight subfaces of f.

We claim that no subface can appear twice in this sequence. Indeed, consider $a_s, a_t \in \partial f$. By construction, $a_s, a_t \in P_i$, so the segment $a_s a_t$ lies in P_i and thus avoids the relative interior of any subface of f. On the other hand, a_s and a_t lie on the boundary of the convex polygon $F_j \cap P_\ell$ that contains f and thus also contains its subfaces. As $a_s a_t$ cuts this polygon in two and the subfaces of f incident to a_s and a_t lie locally on different sides of this segment, they must indeed be distinct subfaces. (Note that for both a_s and a_t the boundary of the incident subface lies locally in counterclockwise direction from the point along ∂f and thus along $\partial(F_j \cap P_\ell)$; see Figure 5.) This proves our claim.

Therefore the number e^* of level-0 edges s of f which are split into two subedges that end up in different positive-weight subfaces of f is at most u^*, the number of such subfaces.

To summarize, the increase that f contributes to our count (ignoring the increase caused by edges split into two subedges of the same subface) is at most $4u^* - 6u^* + 6 =$

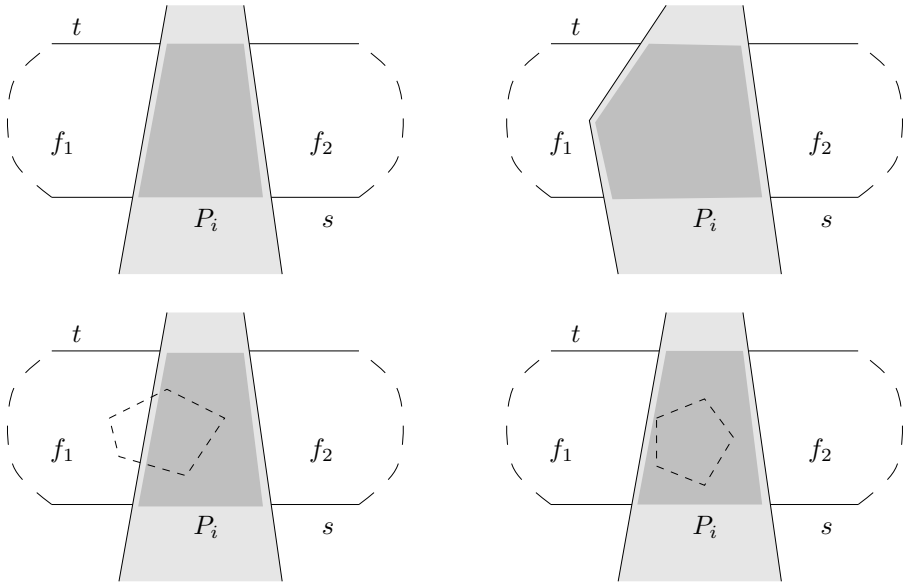


FIG. 6. *Possible configurations of the last kind of splitting. Only the top-left configuration cannot be charged to an outer vertex or to an island.*

$6-2u^*$. This is an increase only if $e^* = u^* \leq 2$; in fact, only the case $u^* = 2$ is relevant, because the preceding arguments give alternative ways of charging for the splittings when $u^* = 1$ (i.e., when there is only one subface with positive weight), and an increase can then occur only when $e^* = u^*$. Thus, reflecting over the preceding discussion, we conclude that it only remains to consider the case when $e^* = u^* = 2$, i.e., where there are two split edges s, t, and where each of the two subfaces in question, f_1, f_2, is bordered by a portion of s and by a portion of t. See Figure 6 for an illustration. Moreover, we can assume that the portions of $f \cap \partial P_i$ which appear along ∂f_1 and along ∂f_2 between s and t do not contain any vertex of $F_j \cap P_i$, and that they do not meet any island of f, because in such cases there are alternative ways of charging for the splitting, similar to those given above. By the same reasoning, we may assume that the shaded quadrilateral confined between s, t, f_1, and f_2, as shown in the bottom-right portion of Figure 6, does not fully contain an island of f, either.

In conclusion, we still need to account for the situations depicted in the top-left portion of Figure 6. That is, we want to count the number of *special quadrilaterals*, that are defined as follows. First, for any $1 \leq i \leq k$, put $U_i = \bigcup_{j \neq i} P_j$. Similarly, for any $1 \leq i < j \leq k$, let $U_{i,j} = \bigcup_{\ell \neq i,j} P_\ell$.

DEFINITION 1. *A quadrilateral Q is* special *if there exist distinct indices $i, j, \ell \in \{1, \ldots, k\}$ (note that the indices are permuted here, with respect to their usage in the above analysis) such that*

1. $Q \subset \partial P_\ell$,
2. $\mathrm{Int}(Q) \subset \mathrm{Int}(P_i \cap P_j)$,
3. *the vertices a, b, c, d of Q are vertices of ∂U,*
4. $Q \subset \partial U_{i,j}$,
5. $ab, cd \subset \partial P_i \cap \partial U_j$, *and*
6. $bc, ad \subset \partial P_j \cap \partial U_i$.

Figure 7 depicts the situation schematically.

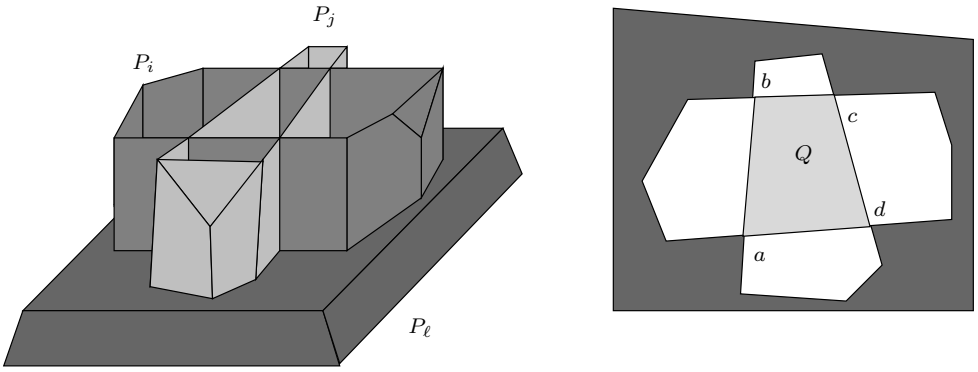


FIG. 7. *A special quadrilateral.*

If we denote the maximum number of such special quadrilaterals, over all collections $\mathcal{P}$ of k convex polyhedra with a total of n faces, by $C^{(2)}(k,n)$, apply the above analysis to each P_i in turn, and sum up the resulting inequalities, we obtain the recurrence

$$(k-2)C^{(1)}(k,n) = \sum_{i=1}^{k} C^{(1)}(k-1, n-n_i) + O(kn + C^{(2)}(k,n)) \,, \tag{4}$$

where the factor $k-2$ appears because, for each level-1 face f, $w(f)$ is counted every time a polyhedron $P \in \mathcal{P}$ is removed and reinserted, except when P is the polyhedron containing f on its boundary or the only polyhedron containing f in its interior.

Lemma 1.6 below provides an $O(k^3+kn)$ bound on $C^{(2)}(k,n)$. Hence, (4) becomes

$$(k-2)C^{(1)}(k,n) \le \sum_{i=1}^{k} C^{(1)}(k-1, n-n_i) + a(k^3+kn) \,,$$

for an appropriate constant a. We claim that the solution of this recurrence is $C^{(1)}(k,n) \le Ak^3 + Bkn\ln k$, where A and B are constants, which we proceed to prove by induction on k. Lemma 1.4 implies that $C^{(1)}(k,n) = O(k^2 n)$, as is easily checked, so choosing B sufficiently large will clearly make our solution valid for $k \le 10$, say. For $k > 10$, the induction hypothesis implies that

$$\begin{aligned}(k-2)C^{(1)}(k,n) &\le \sum_{i=1}^{k} (A(k-1)^3 + B(k-1)(n-n_i)\ln(k-1)) + a(k^3+kn) \\ &\le Ak(k-1)^3 + ak^3 + B(k-1)^2 n\ln(k-1) + akn \,.\end{aligned}$$

Hence it suffices to choose A and B so that

$$Ak(k-1)^3 + ak^3 \le A(k-2)k^3$$

and

$$B(k-1)^2 n\ln(k-1) + akn \le B(k-2)kn\ln k$$

for $k > 10$. The first inequality is equivalent to $ak^2 \le A(k^2-3k+1)$, which will hold if we choose, say, $A > 2a$. The second inequality is equivalent to

$$B\left(1-\frac{1}{k}\right)^2 \ln(k-1) + \frac{a}{k} < B\left(1-\frac{2}{k}\right)\ln k \,,$$

or to

$$\frac{B\ln(k-1)}{k^2}+\frac{a}{k}<B\left(1-\frac{2}{k}\right)\ln\frac{k}{k-1}\ .$$

Since $\ln\frac{k}{k-1}>\frac{1}{k}$ for $k>1$, this inequality is implied by

$$\frac{B\ln(k-1)}{k^2}+\frac{a}{k}<\frac{B}{k}\left(1-\frac{2}{k}\right)$$

or by

$$B\left(1-\frac{2+\ln(k-1)}{k}\right)>a\ ,$$

which holds for $k>10$ if we choose $B>2a$. This completes the proof of our assertion that $C^{(1)}(k,n)<Ak^3+Bkn\ln k$, for appropriate constants A and B. □

It remains to establish the promised upper bound on $C^{(2)}(k,n)$, which is the subject of the following lemma.

LEMMA 1.6. *The number of special quadrilaterals is* $O(k^3+kn)$.

Proof. Consider a special quadrilateral Q. Let $i=i(Q)$, $j=j(Q)$, $\ell=\ell(Q)$ be as in Definition 1. We restrict our attention to special quadrilaterals associated with a particular choice of P_i and P_j, without fixing P_ℓ. Let U' be the set obtained by forming the union of $K=P_i\cap P_j$ with $U_{i,j}$. In other words, we remove these two polyhedra from U and replace them by their intersection. Notice that $\partial Q\subset\partial U'$ and in fact $\partial Q=Q\cap\partial U'$.

A special quadrilateral Q is *trivial* if its boundary is contractible to a point in the closure of $\partial K\setminus U_{i,j}$. Such a contraction defines a "cap" (a topological disk) contained in ∂K and necessarily containing a vertex of K, to which we charge Q. It is easily checked that the caps corresponding to different trivial quadrilaterals form distinct connected components of $\partial K\setminus U_{i,j}$. From Lemma 1.2 we deduce that the number of the corresponding charges, over all choices of i,j, and ℓ, is only $O(kn)$.

We will count the number of those homotopy classes of closed curves in the closure of $\partial K\setminus U_{i,j}$ that contain boundaries of nontrivial quadrilaterals. (See, e.g., [35] for basic material concerning homotopy and related concepts.) By counting classes rather than quadrilaterals we err by a factor of at most 2, because a component of $\partial K\setminus U_{i,j}$, two of whose boundary curves can be continuously transformed into each other without leaving the component, has only two boundary components (since a topological disk with two or more holes has no homotopy-equivalent boundary components).

To carry out the proof, we need a slightly more general formulation of the problem, as follows. Consider a collection $\mathcal{B}=\{B_1,\ldots,B_p\}$ of p convex polyhedra and a collection $\mathcal{K}=\{K_1,\ldots,K_q\}$ of q pairwise-disjoint convex polyhedra. Assume further that no $\mathcal{B}$-polyhedron meets more than one of the $\mathcal{K}$-polyhedra. Let $K=\bigcup\mathcal{K}$ and $B=\bigcup\mathcal{B}$. Two closed curves in the closure $C=C(\mathcal{K},\mathcal{B})$ of $\partial K\setminus B$ are *equivalent* (relative to $(\mathcal{B},\mathcal{K})$) if they are homotopy-equivalent in C. (Here we assume that the components of $\partial K\setminus B$ are sufficiently separated so that a component of the closure is equal to the closure of a component. It is easily checked that in our case this condition holds originally, and that it is maintained inductively.) A closed curve is *trivial* (relative to $(\mathcal{B},\mathcal{K})$) if it is contractible in C to a point, i.e., equivalent to the trivial curve in C. A quadrilateral Q is *nontrivial* if ∂Q is a boundary component of C, ∂Q is not trivial, and Q is contained in the boundary of a $\mathcal{B}$-polyhedron and

avoids the interiors of all $\mathcal{B}$-polyhedra. Notice that this definition of nontriviality is a generalization of the one given above, with $\mathcal{B} = \mathcal{P} \setminus \{P_i, P_j\}$ and $\mathcal{K} = \{P_i \cap P_j\}$. Consider the intersection graph $G = G(\mathcal{B}, \mathcal{K})$ of the polyhedra in $\mathcal{B} \cup \mathcal{K}$, namely, the graph whose vertices are the given polyhedra and whose arcs connect pairs of polyhedra with nonempty intersection. Let $\delta = \delta_G$ be the vertex degree function in G. Let $\bar{\delta}(P) = \max\{\delta(P) - 1, 0\}$. We will prove that the number of homotopy classes of closed paths in C containing the boundary of a nontrivial quadrilateral is at most 2ν, where $\nu = \nu(\mathcal{B}, \mathcal{K}) = \sum_{t=1}^{q} \bar{\delta}(K_t)$. Notice that this claim implies the lemma, because one has, for the original choice of $\mathcal{B}$ and $\mathcal{K}$, $\nu(\mathcal{B}, \mathcal{K}) \leq \delta(P_i \cap P_j) \leq |\mathcal{B}| = k-2$. The argument is repeated for all $1 \leq i < j \leq k$ to yield an $O(k^3)$ bound on the number of nontrivial quadrilaterals.

We proceed by induction on the number of such homotopy classes. The base case, when no nontrivial quadrilaterals exist, is easy, as $\nu \geq 0$ by definition. We now proceed with the general induction step. Let Q be a nontrivial quadrilateral. Without loss of generality, assume that $Q \subset \partial B_1 \cap K_1$. Cut K_1 along Q into two convex portions, and shrink the portion (call it K_1'), that lies in the half-space H bounded by the plane spanned by Q and not containing B_1, slightly away from Q. Extend the other portion (call it K_1'') slightly towards K_1', so that it "pops up" above ∂B_1 but still avoids K_1' and meets only the polyhedra it met before the displacement. See Figure 8 for an illustration. Let $\mathcal{K}' = \mathcal{K} \setminus \{K_1\} \cup \{K_1', K_1''\}$ and $K' = \bigcup \mathcal{K}'$. We have constructed a new instance of the problem, namely $(\mathcal{B}, \mathcal{K}')$. We note that no polyhedron of $\mathcal{B}$ can intersect both portions of K_1 without meeting Q (or, rather, a slight displacement of Q away from B_1), and no polyhedron of $\mathcal{B}$ meets the displaced Q, by assumption. This implies that no $\mathcal{B}$-polyhedron intersects more than one $\mathcal{K}'$-polyhedron, so the new problem satisfies the same assumptions as the old one. We claim that the new problem is "smaller": By definition of a nontrivial quadrilateral, ∂Q cannot be homotopically shrunk to a point in C. On the other hand, Q is clearly trivial in $(\mathcal{B}, \mathcal{K}')$. Moreover, a nontrivial quadrilateral Q' not equivalent to Q in $(\mathcal{B}, \mathcal{K})$ remains nontrivial in $(\mathcal{B}, \mathcal{K}')$—this can be easily deduced by examining the shape of the components of $\partial K \setminus B$ and of $\partial K' \setminus B$ containing $\partial Q'$. However, it is possible for two distinct homotopy classes containing nontrivial quadrilaterals with respect to $(\mathcal{B}, \mathcal{K})$ to merge in $(\mathcal{B}, \mathcal{K}')$. This can happen if the connected component E of C containing Q was topologically equivalent to a disk with two holes, whose three boundary components (one of which is Q) are all nontrivial quadrilaterals; after the splitting at Q, E is transformed into two components, E' and E''. E'' is equivalent to a disk (in fact, $\partial E'' = \partial Q$) and thus does not give rise to a nontrivial quadrilateral, while E' is equivalent to a disk with a single hole and its two bounding quadrilaterals now become homotopic to each other, thus merging two previously distinct classes. This is the only possible merging of homotopy classes, as is easily checked; see Figure 9 for an illustration. Hence the number of classes containing nontrivial quadrilaterals increases by either one or two when we pass from $(\mathcal{B}, \mathcal{K}')$ back to $(\mathcal{B}, \mathcal{K})$.

On the other hand, since Q is nontrivial with respect to $(\mathcal{B}, \mathcal{K})$, ∂Q cannot be shrunk in C by moving along ∂K_1 into the half-space H defined above. Thus K_1' must meet at least one $\mathcal{B}$-polyhedron. Clearly, K_1'' meets B_1 itself. Hence, the degrees of K_1' and of K_1'' in $G' = G(\mathcal{B}, \mathcal{K}')$ are both positive. Thus

$$\bar{\delta}_{G'}(K_1') + \bar{\delta}_{G'}(K_1'') = \delta_{G'}(K_1') - 1 + \delta_{G'}(K_1'') - 1 = \delta_G(K_1) - 2 = \bar{\delta}_G(K_1) - 1.$$

We have used the fact that $\delta_{G'}(K_1') + \delta_{G'}(K_1'') = \delta_G(K_1)$, which follows by observing that the polyhedra of $\mathcal{B}$ that intersect K_1 are exactly those that intersect either K_1'

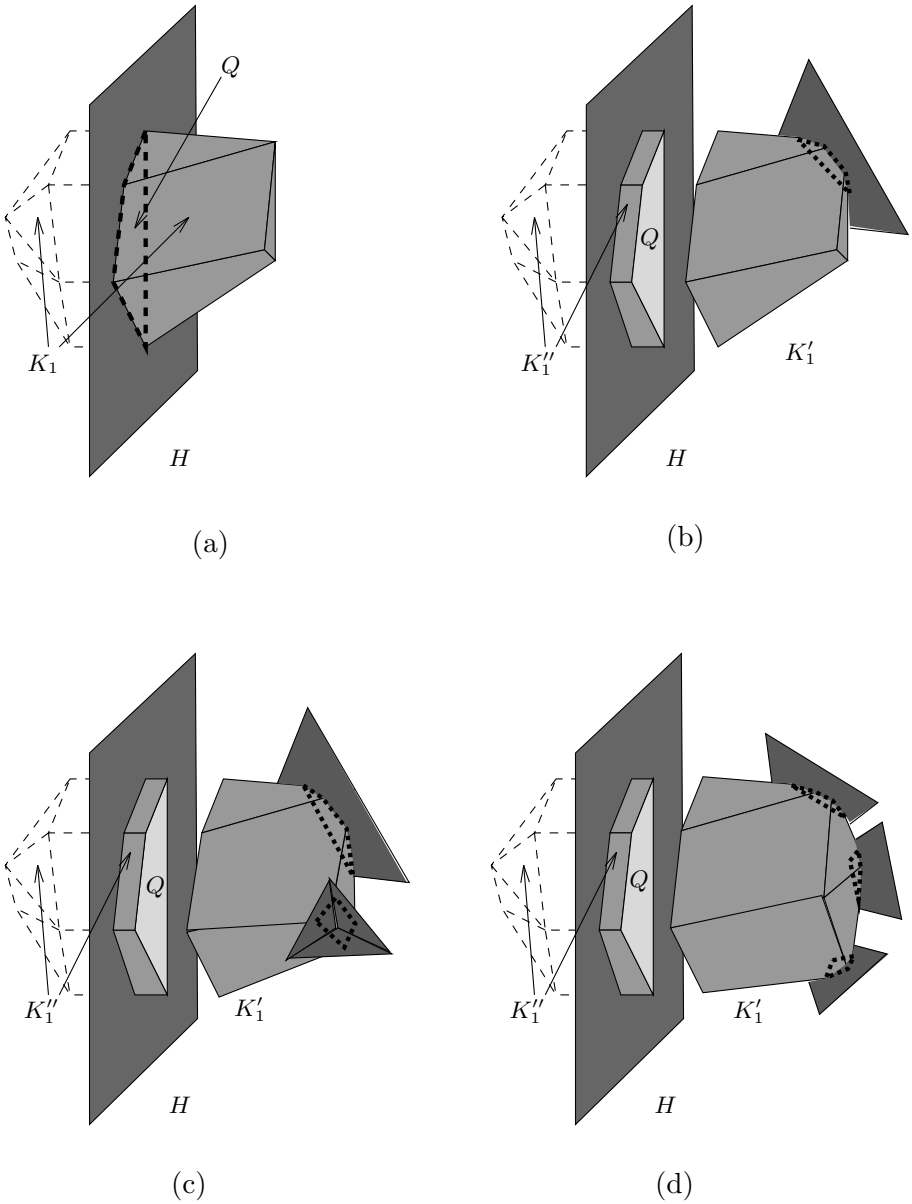


FIG. 8. *Cutting K_1 at a special quadrilateral Q, and the subsequent changes in the homotopy structure. In* (a)*, Q is trivial, so no cut needs to be made. In* (b)*, both Q and the quadrilateral "opposite" to Q become trivial after the cut. In* (c)*, Q becomes trivial, and the homotopy classes of the two adjacent quadrilaterals merge after the cut (this case is also illustrated in Figure* 9*). In* (d)*, Q becomes trivial, and no merges of the homotopy classes of the adjacent quadrilaterals occur.*

or K_1'', and that no polyhedron of $\mathcal{B}$ can intersect both these portions, as argued above. Therefore, $\nu(\mathcal{B},\mathcal{K}) = \nu(\mathcal{B},\mathcal{K}') + 1$, and the increase in 2ν bounds the increase in the number of homotopy classes containing nontrivial quadrilaterals, leading to the desired inequality. This completes the proof of the claim and of the lemma. □

Remarks. (1) The bound given by the last lemma is tight in the worst case. A family $\mathcal{P}$ of polyhedra with $\Omega(k^3)$ special quadrilaterals is given by the gridlike con-

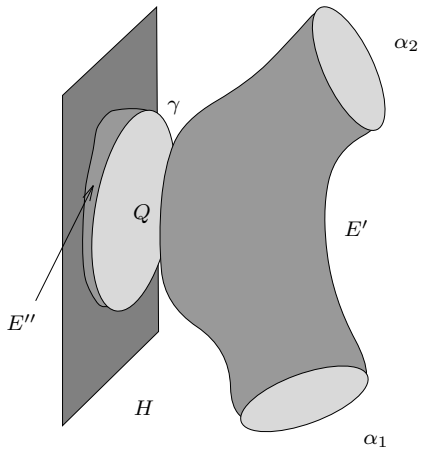


FIG. 9. *A situation where the number of classes decreases by* 2 *as we change* $(\mathcal{B}, \mathcal{K})$ *to* $(\mathcal{B}, \mathcal{K}')$*: The class* γ *containing* ∂Q *becomes trivial and classes* α_1 *and* α_2 *merge after* K_1 *is cut into* K_1' *and* K_1''.

struction in Figure 3(a). A family with $\Omega(kn)$ special quadrilaterals can be obtained as follows: First consider a set consisting of the following three polyhedra. Take P_1 to be any convex polyhedron with n faces. Draw an "X" on each face of P_1 (i.e., a pair of crossing segments) and then lift them slightly away from P_1. Pick one segment out of each pair and form their convex hull; this is polyhedron P_2. The third polyhedron P_3 is the convex hull of the remaining segments. It is easily checked that there is one special quadrilateral associated with each "X," so that this family of three polyhedra with $\Theta(n)$ faces has n special quadrilaterals. Now we repeat the construction but start with a polyhedron P with $\Theta(n/k)$ faces, draw a sufficiently small $k/2 \times k/2$ grid in each face away from the edges of P, lift it outward, and form k new polyhedra by taking convex hulls of collections of segments, one from each grid, thereby exhausting all segments and placing each segment in just one collection. It is easily checked that on each face of P we obtain $k^2/4$ special quadrilaterals, for a total of $\Omega(n/k) \times k^2/4 = \Omega(nk)$. The family consists of $k+1$ polyhedra with $\Theta(n/k) + k \times \Theta(n/k) = \Theta(n)$ faces. The claimed lower bound is attained by combining the two constructions.

(2) It is interesting to note that Lemma 1.6 is the only "source" of the term $O(k^3)$ in the bound for the complexity of the union. See section 3 which mentions an improved bound for an important special case.

We next describe an easy corollary of our main result. Let $\mathcal{P}$ be a collection of k convex polyhedra in general position with a total of n faces. A simple application of the probabilistic technique of [15, 32] yields the following.

THEOREM 1.7. *For any* $1 \le \lambda \le k-2$*, the total number of level-*j *vertices of* $\mathcal{A}(\mathcal{P})$*, for all* $j < \lambda$*, is* $O(k^3 + \lambda kn \log(k/\lambda))$.

Proof. Let $\mathcal{R}$ be a random sample of $r = \lfloor k/\lambda \rfloor$ polyhedra of $\mathcal{P}$. The analysis of [15, 32] implies that the number of inner vertices of $\mathcal{A}(\mathcal{P})$ at level $< \lambda$ is $O(\lambda^3 \mathbf{E}[C_0(\mathcal{R})])$, where $C_0(\mathcal{R})$ is, as above, the number of inner vertices of the union of the polyhedra of $\mathcal{R}$, and where $\mathbf{E}$ denotes expectation with respect to the choice of $\mathcal{R}$. By Theorem 1.1, $C_0(\mathcal{R}) = O(r^3 + rn_{\mathcal{R}} \log r)$, where $n_{\mathcal{R}}$ is the total number of faces of the polyhedra in $\mathcal{R}$. It thus follows that $\mathbf{E}[C_0(\mathcal{R})] = O(r^3 + r \log r \cdot \mathbf{E}[n_{\mathcal{R}}])$. Now $\mathbf{E}[n_{\mathcal{R}}] = O(n/\lambda)$, as any of the n faces of the polyhedra of $\mathcal{P}$ has probability $r/k \le 1/\lambda$ to appear as a face of a polyhedron of $\mathcal{R}$. Hence, the number of vertices

of the above kind that lie at level $< \lambda$ is

$$O\left(\lambda^3 r^3 + \lambda^3 r \log r \cdot \frac{n}{\lambda}\right) = O\left(k^3 + \lambda k n \log \frac{k}{\lambda}\right) .$$

Adding to this estimate the $O(kn)$ bound given in Lemma 1.3 for the number of outer vertices, the theorem follows. □

Remark. When $\lambda = k - 2$, all the vertices of $\mathcal{A}(\mathcal{P})$ are counted in the preceding theorem, and the above bound becomes $O(k^2 n)$, in agreement with Lemma 1.4.

A number of applications of the last result are described in [1]. It is applied there to a geometric pattern matching problem, seeking to determine the rigid translation of one set of points relative to another, which optimizes a certain distance function between the two sets. The paper also describes how the above bound can be used to bound the number of combinatorially distinct Euclidean minimum spanning trees that can be obtained by adding an extra point to a given point set.

2. Efficient construction of the union. We next describe an efficient randomized algorithm for constructing the union U of a collection $\mathcal{P}$ of polyhedra as above. The main idea of the algorithm is quite simple. In principle, we want to insert the given polyhedra one by one in random order and to maintain the union of the polyhedra that have been inserted so far. However, we actually apply the following somewhat modified strategy. First, we compute all pairwise intersections $P_i \cap P_j$, for $1 \le i < j \le k$, by repeated applications of the linear-time algorithm of Chazelle [12] for intersecting two convex polyhedra in 3-space. (A slower algorithm, such as that described in [29, 31], would also suffice for our purposes.) By Lemma 1.3, this takes $O(kn)$ time, and yields, for each face F of any polyhedron P_i, the collection $\mathcal{Q}_F$ of the convex polygons $Q_j = F \cap P_j$, for $j \ne i$. Clearly, the set $U_F = F \setminus \bigcup_{j \ne i} Q_j$ is the portion of F that appears on ∂U, so our goal is to compute the sets U_F, over all faces F. Reconstructing the boundary of U from this information is relatively straightforward, by gluing the portions U_F to each other in an appropriate manner.

Let F be a fixed face of some $P_i \in \mathcal{P}$. We choose a random order of the polyhedra in $\mathcal{P} \setminus \{P_i\}$ and insert the polygons Q_j, one by one, in the corresponding order, maintaining the complement of their union as we go. For this, we use the same technique as in [13, 16, 20, 30], which maintains a vertical decomposition of the complement into trapezoids and a "history dag" of all trapezoids that were ever created by the algorithm. Whenever a new polygon Q_j is inserted, we search through the dag for trapezoids intersected by ∂Q_j and then update the complement of the union by refining the trapezoidal decomposition to reflect the presence of Q_j and by discarding those trapezoids that end up inside Q_j. We refer the reader to the papers cited above for more details. We apply this procedure to every face F of each polyhedron of $\mathcal{P}$, and, as already mentioned, the union of the resulting regions U_F yields the boundary of the desired union U.

The cost of the algorithm is proportional to the number of trapezoids that are created during the execution of the algorithm, plus the initial cost of $O(kn)$ of computing all the collections $\mathcal{Q}_F$, plus the cost of tracing polygons through the history dag. This is more complicated than the standard technique because the number of children of a trapezoid in the dag need not be constant: when a trapezoid τ is "killed" by adding a polygon Q_j, it may be split into many overlapping trapezoids that become its children in the dag. However, when this happens, all but $O(1)$ of these subtrapezoids have a vertex of Q_j as one of their vertices. By Lemma 1.3, the total number of these vertices is only $O(kn)$. Using standard arguments, one can show that the expected

overall number of trapezoids τ that are ever created by the algorithm and contain such a vertex in their interior is $O(kn \log k)$. This allows us to trace the vertices of the inserted polygons Q_j, using a simple binary search mechanism, at a total expected cost of $O(kn \log k \log n)$. With this information available, if the polygon Q_j being inserted intersects an inner trapezoid τ of the dag, it is not difficult to find all the children of τ that it crosses in time proportional to their number. Hence the overall expected cost of tracing polygons through the dag is $O(kn \log k \log n)$ plus the sum of the weights of the trapezoids. Here the *weight* of a trapezoid $\tau \subseteq F$ is the number of polygons $Q_j \in \mathcal{Q}_F$ that intersect the relative interior of τ; each of them either fully contains τ, or is fully contained in τ, or its boundary crosses τ. A "canonical" trapezoid $\tau \subseteq F$ is one that occurs in the trapezoidal decomposition of the complement (within F) of the union of some subset of Q_F. Such a trapezoid is defined by between one and five polyhedra—P_i plus the at most four polyhedra that define its sides and corners in F. The following argument applies only to canonical trapezoids defined by exactly five polyhedra. It has to be repeated, with straightforward modifications, for trapezoids defined by one, two, three, or four polyhedra. Adapting the approach of the papers cited above, and using the Clarkson–Shor analysis technique [15], one easily shows the following.

Claim 1. The probability that a canonical trapezoid $\tau \subseteq F$ with weight w is created during the incremental construction is $1/\binom{w+4}{4}$.

Proof. When the algorithm processes F, the four other polyhedra defining τ have to be inserted before any of the w polyhedra intersecting $\mathrm{Int}(\tau)$ are inserted. □

Claim 2. The overall number T_w of canonical trapezoids with weight less than w, over all faces F of the given polyhedra, is $O(w^2k^3 + w^3kn \log \frac{k}{w})$.

Proof. We use the Clarkson–Shor analysis technique [15]. Since a canonical trapezoid τ is defined by five polyhedra (one of which contains the face F in which τ lies, and the other four define the four sides of τ, as in the papers cited above), it follows that $T_w = O(w^5 \mathbf{E}[T_1(\lceil k/w \rceil)])$, where $\mathbf{E}[T_1(\lceil k/w \rceil)]$ is the expected number of 0-weight trapezoids that arise for a random sample $\mathcal{R}$ of $\lceil k/w \rceil$ polyhedra from $\mathcal{P}$. Clearly, each such trapezoid lies on the boundary of the union $\bigcup \mathcal{R}$, and the expected number of such trapezoids is proportional to the expected complexity of $\bigcup \mathcal{R}$, namely to $O((k/w)^3 + (k/w) \cdot (n/w) \log(k/w))$ (see the proof of Theorem 1.7). The claim is now immediate. □

The expected running time of the algorithm, over all faces F, is thus

$$O\left(kn + \sum_{w=0}^{k-5} \frac{(w+1)t_w}{\binom{w+4}{4}}\right),$$

where t_w is the total number of trapezoids whose weight is exactly w. Since $t_w = T_{w+1} - T_w$ (with $T_0 = 0$), we can rewrite the above sum as

$$\begin{aligned}
&\sum_{w=0}^{k-5} \frac{T_{w+1} - T_w}{(w+4)(w+3)(w+2)} \\
&\quad = \sum_{w=1}^{k-5} T_w \left[\frac{1}{(w+3)(w+2)(w+1)} - \frac{1}{(w+4)(w+3)(w+2)}\right] \\
&\qquad + \frac{T_{k-4}}{(k-1)(k-2)(k-3)}
\end{aligned}$$

$$= O\left(kn + \sum_{w=1}^{k-5} \frac{w^2k^3 + w^3kn\log\frac{k}{w}}{(w+4)(w+3)(w+2)(w+1)}\right)$$

$$= O\left(kn + \sum_{w=1}^{k-5} \frac{k^3}{w^2} + \sum_{w=1}^{k-5} \frac{kn}{w}\log\frac{k}{w}\right)$$

$$= O(k^3 + kn\log^2 k)\ .$$

Hence we have the following.

THEOREM 2.1. *The union of a collection of k convex polyhedra in 3-space with a total of n faces can be computed in randomized expected time $O(k^3 + kn\log k\log n)$.*

Remark. It is worth noting that the algorithm fixes one polyhedron P_i (the one containing the face F), inserts the other polyhedra in random order, and repeats this over all P_i's. Thus the algorithm does not apply a single global insertion order, but this does not affect adversely its analysis.

The following theorem can be proved by an essentially identical argument.

THEOREM 2.2. *Consider a collection $\mathcal{P}$ of k 3-polyhedra with a total of n facets with the property that the union of any subset of k' polyhedra with a total of n' facets has complexity $O(k'n'\log k')$. Then the union of the whole collection can be computed in randomized expected time $O(kn\log k\log n)$.*

3. A motion planning application. A major application where the union of a collection of convex polyhedra in 3-space needs to be computed is that of planning translational motion for a convex polyhedron B in a polyhedral environment. Assume that B has p faces and that the obstacles (the complement of free space) can be represented as k convex polyhedra with pairwise-disjoint interiors, having a total of q faces. For each obstacle polyhedron A_i, for $i = 1, \ldots, k$, we form the *Minkowski sum* $P_i = A_i \oplus (-B)$, and the union U of the k resulting convex polyhedra P_i represents the portion of the *configuration space* of B where it collides with some obstacle; the complement of U is the *free configuration space* of B, which is what we want to compute. See [4, 5] for more details.

If A_i has q_i faces, then P_i has $O(pq_i)$ faces, so the total number of faces of the P_i's is $O(pq)$, where $q = \sum_{i=1}^{k} q_i$. Hence Theorem 1.1 implies that the combinatorial complexity of the free configuration space of B is $O(k^3 + kpq\log k)$. However, in this special case, Lemma 1.6 can be strengthened, using a different topological analysis, to show that there are only $O(kn)$ special quadrilaterals, which in turn implies that the complexity of the free configuration space is only $O(kpq\log k)$. This is proved in a companion paper [8]. Hence, by Theorem 2.2, this space can be constructed in $O(kpq\log k\log\,(pq))$ randomized expected time.

4. Conclusion. In this paper we have obtained almost tight bounds on the maximum complexity of the union of k convex polyhedra with a total of n faces in three dimensions, presented an efficient randomized algorithm for computing the boundary of the union, and mentioned several applications of these results.

The paper raises several open problems. First, we would like to tighten the small remaining gap between the upper and lower bounds for the complexity of the union. We suspect that the logarithmic factor appearing in the upper bound is just an artifact of our proof technique, and conjecture that the lower bound is tight.

Another open problem is to obtain sharp bounds for the complexity of a single component of the complement of the union of k convex polyhedra with a total of n

faces. The result of [5] implies that this complexity is $O(n^2 \log n)$, but we conjecture that the bound is close to $O(kn)$. The lower bound construction in the proof of Theorem 1.1 implies that the complexity of such a component can be $\Omega(nk\alpha(k))$ in the worst case.

It would also be interesting to extend our bounds to higher dimensions. Here is an easy lower bound construction, for any fixed dimension $d > 2$. Let d be even. Consider a planar collection of $k/(d/2)$ polygons with a total of $n/(d/2)$ edges, so that their union has $\Omega(k^2 + n\alpha(k))$ vertices on its boundary [7]. Consider a family of $d/2$ mutually orthogonal 2-flats in d-space. Place one copy of this configuration into each 2-flat and extend each polygon P into a prism in the remaining $d-2$ coordinates (namely, this prism is the Cartesian product of P and the $(d-2)$-space orthogonal to the 2-flat containing P). It is easily verified that the number of vertices of the union of the resulting k prisms, having a total of n facets, is at least $(\Omega(k^2 + n\alpha(k)))^{d/2} = \Omega(k^d + n^{d/2}\alpha^{d/2}(k))$. The construction for odd $d > 3$ is similar, with one of the two-dimensional constructions replaced by the three-dimensional lower bound construction described in section 1, for a total union complexity of $\Omega(k^d + kn^{\lfloor d/2 \rfloor}\alpha^{\lfloor d/2 \rfloor}(k))$. Notice that both bounds are tight at the extremes of the range of k, i.e., when $k = \Theta(1)$ or $k = \Theta(n)$.

Concerning an upper bound, under the assumptions of general position there is an easy argument (see Katona [25] and Kovalev [26] for a more general statement) that shows that the complement of the union of k convex polyhedra in d-space has at most $O(k^d)$ connected components. An upper bound of $O(n^d)$ on the total complexity, where n is the total number of polyhedra facets, is immediate by considering the arrangement induced by the hyperplanes spanned by the polyhedra facets, but this is likely to be a gross overestimate of the true complexity. Indeed, it is pointed out in [1] that the total complexity of the arrangement induced by the facets is $\Theta(k^{\lceil d/2 \rceil} n^{\lfloor d/2 \rfloor})$ in the worst case. How close these bounds are to the true worst-case complexity of the union is not known. Is the preceding lower bound tight or close to being tight?

There are also several algorithmic open problems. One is to obtain an efficient *deterministic* algorithm for constructing the union. Another one is to obtain an efficient technique for point location in the complement of the union of k polyhedra in 3-space with a total of n faces.

REFERENCES

[1] B. Aronov, M. Bern, and E. Eppstein, *Arrangements of polytopes with applications*, manuscript, 1993.

[2] B. Aronov, J. Matoušek, and M. Sharir, *On the sum of squares of cell complexities in hyperplane arrangements*, J. Combin. Theory Ser. A, 65 (1994), pp. 311–321.

[3] B. Aronov, M. Pellegrini, and M. Sharir, *On the zone of a surface in a hyperplane arrangement*, Discrete Comput. Geom., 9 (1993), pp. 177–188.

[4] B. Aronov and M. Sharir, *Triangles in space, or building (and analyzing) castles in the air*, Combinatorica, 10 (1990), pp. 137–173.

[5] B. Aronov and M. Sharir, *Castles in the air revisited*, Discrete Comput. Geom., 12 (1994), pp. 119–150.

[6] B. Aronov and M. Sharir, *The union of convex polyhedra in three dimensions*, Proc. 34th Annual IEEE Sympos. Found. Comput. Sci., IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 518–527.

[7] B. Aronov and M. Sharir, *The common exterior of convex polygons in the plane*, Comput. Geom. Theory Appl., 8 (1997), pp. 139–149.

[8] B. Aronov and M. Sharir, *On translational motion planning in three dimensions*, SIAM J. Comput., 26 (1997), pp. 1785–1803.

[9] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec, *Applications of random sampling to on-line algorithms in computational geometry*, Discrete Comput. Geom., 8 (1992), pp. 51–71.

[10] J.-D. Boissonnat and K. Dobrindt, *On-line randomized construction of the upper envelope of triangles and surface patches in* $\mathbb{R}^3$, Comput. Geom. Theory Appl., to appear; J. Algorithms, to appear.

[11] J.-D. Boissonnat, M. Sharir, B. Tagansky, and M. Yvinec, *Voronoi diagrams in higher dimensions under certain polyhedral convex distance functions*, in Proc. 11th ACM Symp. on Comput. Geom., ACM, New York, 1995, pp. 79–88.

[12] B. Chazelle, *An optimal algorithm for intersecting three-dimensional convex polyhedra*, SIAM J. Comput., 21 (1992), pp. 671–696.

[13] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Snoeyink, *Computing a face in an arrangement of line segments,* SIAM J. Comput., 22 (1993), pp. 1286–1302.

[14] L. P. Chew, K. Kedem, M. Sharir, B. Tagansky, and E. Welzl, *Voronoi diagrams of lines in three dimensions under a polyhedral convex distance function*, in Proc. 6th ACM-SIAM Symp. on Discrete Algorithms, SIAM, Philadelphia, 1995, pp. 197–204.

[15] K. Clarkson and P. Shor, *Applications of random sampling in computational geometry* II, Discrete Comput. Geom., 4 (1989), pp. 387–421.

[16] M. de Berg, K. Dobrindt, and O. Schwarzkopf, *On lazy randomized incremental construction*, Discrete Comput. Geom., 14 (1995), pp. 261–286.

[17] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.

[18] H. Edelsbrunner, L. Guibas, and M. Sharir, *The complexity and construction of many faces in arrangements of lines and of segments*, Discrete Comput. Geom., 5 (1990), pp. 161–196.

[19] H. Edelsbrunner, R. Seidel, and M. Sharir, *On the zone theorem in hyperplane arrangements,* SIAM J. Comput., 22 (1993), pp. 418–429.

[20] L. Guibas, D. Knuth, and M. Sharir, *Randomized incremental construction of Voronoi and Delaunay diagrams*, Algorithmica, 7 (1992), pp. 381–413.

[21] D. Halperin and M. Sharir, *A near-quadratic algorithm for planning the motion of a polygon in a polygonal environment*, Discrete Comput. Geom., 16 (1996), pp. 121–134.

[22] D. Halperin and M. Sharir, *New bounds for lower envelopes in three dimensions with applications to visibility in terrains*, Discrete Comput. Geom., 12 (1994), pp. 313–326.

[23] D. Halperin and M. Sharir, *Arrangements and their applications in robotics*: *Recent developments*, in The Algorithmic Foundations of Robotics, K. Goldberg, D. Halperin, J.C. Latombe, and R. Wilson, eds., A.K. Peters, Boston, MA, 1995, pp. 495–511.

[24] D. Halperin and M. Sharir, *Almost tight upper bounds for the single cell and zone problems in three dimensions*, Discrete Comput. Geom., 14 (1995), pp. 385–410.

[25] G. O. Katona, *On a problem of L. Fejes Tóth*, Stud. Sci. Math. Hung., 12 (1977), pp. 77–80.

[26] M. D. Kovalev, *Svoisto vypuklykh mnozhestv i ego prilozhenie,* Mat. Zametki, 44 (1988), pp. 89–99.

[27] K. Kedem, R. Livne, J. Pach, and M. Sharir, *On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles*, Discrete Comput. Geom., 1 (1986), pp. 59–71.

[28] J. Matoušek, J. Pach, M. Sharir, S. Sifrony, and E. Welzl, *Fat triangles determine linearly many holes*, SIAM J. Comput., 23 (1994), pp. 154–169.

[29] K. Mehlhorn, *Data Structures and Algorithms* 3: *Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, Heidelberg, 1984.

[30] N. Miller and M. Sharir, *Efficient randomized algorithms for constructing the union of fat triangles and of pseudo-disks*, manuscript, 1991.

[31] F. Preparata and M. Shamos, *Computational Geometry*: *An Introduction*, Springer-Verlag, New York, 1985.

[32] M. Sharir, *On k-sets in arrangements of curves and surfaces*, Discrete Comput. Geom., 6 (1991), pp. 593–613.

[33] M. Sharir, *Almost tight upper bounds for lower envelopes in higher dimensions*, Discrete Comput. Geom., 12 (1994), pp. 327–345.

[34] M. Sharir and P.K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, Cambridge, New York, Melbourne, 1995.

[35] E.H. Spanier, *Algebraic Topology*, Springer-Verlag, New York, Heidelberg, Berlin, 1966.

[36] B. Tagansky, *A new technique for analyzing substructures in arrangements*, Discrete Comput. Geom., 16 (1996), pp. 455–479.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1689–1713, December 1997

007

STAR UNFOLDING OF A POLYTOPE WITH APPLICATIONS*

PANKAJ K. AGARWAL†, BORIS ARONOV‡, JOSEPH O'ROURKE§, AND CATHERINE A. SCHEVON¶

Abstract. We introduce the notion of a *star unfolding* of the surface $\mathcal{P}$ of a three-dimensional convex polytope with n vertices, and use it to solve several problems related to shortest paths on $\mathcal{P}$.

The first algorithm computes the edge sequences traversed by shortest paths on $\mathcal{P}$ in time $O(n^6\beta(n)\log n)$, where $\beta(n)$ is an extremely slowly growing function. A much simpler $O(n^6)$ time algorithm that finds a small superset of all such edge sequences is also sketched.

The second algorithm is an $O(n^8 \log n)$ time procedure for computing the *geodesic diameter* of $\mathcal{P}$: the maximum possible separation of two points on $\mathcal{P}$ with the distance measured along $\mathcal{P}$.

Finally, we describe an algorithm that preprocesses $\mathcal{P}$ into a data structure that can efficiently answer the queries of the following form: "Given two points, what is the length of the shortest path connecting them?" Given a parameter $1 \le m \le n^2$, it can preprocess $\mathcal{P}$ in time $O(n^6 m^{1+\delta})$, for any $\delta > 0$, into a data structure of size $O(n^6 m^{1+\delta})$, so that a query can be answered in time $O((\sqrt{n}/m^{1/4})\log n)$. If one query point always lies on an edge of $\mathcal{P}$, the algorithm can be improved to use $O(n^5 m^{1+\delta})$ preprocessing time and storage and guarantee $O((n/m)^{1/3}\log n)$ query time for any choice of m between 1 and n.

Key words. convex polytopes, geodesics, shortest paths, star unfolding

AMS subject classifications. 52B10, 52B55, 68Q25, 68U05

PII. S0097539793253371

1. Introduction.

The widely studied problem of computing shortest paths in Euclidean space amidst polyhedral obstacles arises in planning optimal collision-free paths for a given robot. In two dimensions, the problem has been thoroughly explored and a number of efficient algorithms have been developed; see, e.g., [SS86, Wel85, Mit93, HS93]. However, the problem becomes significantly harder in three dimensions. Canny and Reif [CR87] have shown it to be NP-hard, and the fastest available algorithm runs in singly exponential time [RS89, Sha87]. This has motivated researchers to develop efficient approximation algorithms [Pap85, Cla87, CSY94, HS95] and to study interesting special cases [MMP87, Sha87]. One of the most widely studied special cases is computing shortest paths on the surface of a convex polytope [SS86, MMP87, Mou90]; this problem was originally formulated by H. Dudeney in 1903; see [Gar61, p. 36]. Sharir and Schorr presented an $O(n^3 \log n)$ algorithm

* Received by the editors August 6, 1993; accepted for publication (in revised form) November 27, 1995. A preliminary version of this paper appeared in *Proceedings of the* 2*nd Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Comput. Sci. 447, Springer-Verlag, Berlin, 1990, pp. 251–263 [AAOS90]. Part of the work was carried out while the first two authors were at the Courant Institute of Math. Sci., New York University, and later at DIMACS, an NSF Science and Technology Center, and while the fourth author was at the Dept. of Computer Science, Johns Hopkins University, Baltimore, MD.

http://www.siam.org/journals/sicomp/26-6/25337.html

† Department of Computer Science, Box 90129, Duke University, Durham, NC 27708-0129 (pankaj@euclid.cs.duke.edu). The work of this author was supported by NSF grants CCR-91-06514 and STC88-09648.

‡ Department of Computer Science, Polytechnic University, Brooklyn, NY 11201 (aronov@ziggy.poly.edu). The work of this author was partially supported by an AT&T Bell Laboratories Ph.D. Scholarship and by NSF grants CCR-92-11541 and STC88-09648.

§ Department of Computer Science, Smith College, Northampton, MA 01063 (orourke@cs.smith.edu). The work of this author was supported by NSF grants CCR-91-22169 and CCR-88-2194.

¶ AT&T Bell Laboratories, P. O. Box 636, Murray Hill, NJ 07974 (schevon@mail.med.upenn.edu).

for this problem; this algorithm was subsequently improved by Mitchell, Mount, and Papadimitriou [MMP87] to $O(n^2 \log n)$ and then by Chen and Han to $O(n^2)$ [CH90].

In this paper we consider three problems involving shortest paths on the surface $\mathcal{P}$ of a convex polytope in $\mathbb{R}^3$. A shortest path on $\mathcal{P}$ is uniquely identified by its endpoints and by the sequence of edges it encounters. Sharir [Sha87] proved that no more than $O(n^7)$ distinct sequences of edges are actually traversed by the shortest paths on $\mathcal{P}$. This bound was subsequently improved to $\Theta(n^4)$ [Mou85, SO88]. Sharir also gave an $O(n^8 \log n)$ time algorithm to compute an $O(n^7)$ size superset of shortest-path edge sequences. However, computing the exact set of shortest-path edge sequences seems to be very difficult. Schevon and O'Rourke [SO89] presented an algorithm that computes the exact set of all shortest-path edge sequences and also identifies, in logarithmic time, the edge sequences traversed by all shortest paths connecting a given pair of query points lying on edges of $\mathcal{P}$. The sequences can be explicitly generated, if necessary, in time proportional to their length. Their algorithm, however, requires $O(n^9 \log n)$ time and $O(n^8)$ space.[1]

In this paper we propose two edge-sequence algorithms. The first is a simple $O(n^6)$ algorithm to compute a superset of shortest-path edge sequences, thus improving the result of [Sha87]; it is described in section 5. The second computes the exact set of shortest-path edge sequences in $O(n^6 \beta(n) \log n)$ time, where $\beta(\cdot)$ is an extremely slowly growing function. This second algorithm significantly improves the previously mentioned $O(n^9 \log n)$ algorithm. The computation of the collection of all shortest-path edge sequences on a polytope is an intermediate step of several algorithms [Sha87, OS89] and is of interest in its own right.

The second problem studied in this paper is that of computing the *geodesic diameter* of $\mathcal{P}$, i.e., the maximum distance along $\mathcal{P}$ between any two points on $\mathcal{P}$. O'Rourke and Schevon [OS89] gave an $O(n^{14} \log n)$ time procedure for determining the geodesic diameter of $\mathcal{P}$. In [AAOS90], we presented a simpler and faster algorithm whose running time is $O(n^{10})$. Here we improve this to $O(n^8 \log n)$.

The third problem involves answering queries of the following form: "Given $x, y \in \mathcal{P}$, determine the distance between x and y along $\mathcal{P}$." Given a parameter $1 \leq m \leq n^2$, we present a method for preprocessing $\mathcal{P}$, in $O(n^6 m^{1+\delta})$ time, into a data structure of size $O(n^6 m^{1+\delta})$ for any $\delta > 0$, so that a query can be answered in time $O((\sqrt{n}/m^{1/4}) \log n)$. If x is known to lie on an edge of the polytope, the preprocessing and storage requirements are reduced to $O(n^5 m^{1+\delta})$ and the query time becomes $O((n/m)^{1/3} \log n)$ for $1 \leq m \leq n$. Constants of proportionality in the above bounds depend on the choice of δ.

Our algorithms are based on a common geometric concept, the *star unfolding*. Let $x \in \mathcal{P}$ be a point such that the shortest path from x to every vertex of $\mathcal{P}$ is unique. Intuitively, the star unfolding of $\mathcal{P}$ with respect to this point is obtained by removing these n shortest paths from $\mathcal{P}$ and embedding the remaining surface isometrically in the plane. Remarkably, the star unfolding is isometric to a simple planar polygon and the structure of shortest paths emanating from x on $\mathcal{P}$ corresponds to a certain Voronoi diagram in the plane [AO92]. Together with relative stability of the combinatorial structure of the unfolding as x moves within a small neighborhood on $\mathcal{P}$, these properties facilitate the construction of efficient algorithms for the above

[1] A preliminary version of this algorithm [SO88] erroneously claimed a time complexity of $O(n^7 2^{\alpha(n)} \log n)$; this claim was corrected in [SO89]. Hwang, Chang, and Tu [HCT89] suggested a more efficient procedure for solving the same problem, but the key claim (their second Lemma 6) has yet to be convincingly established [Sch89, Chapter 5].

three problems. Although all of the algorithms have high polynomial time complexity as a function of n, they are not so inefficient in relation to the worst-case number of edge sequences, $\Theta(n^4)$.

Chen and Han [CH90] independently discovered the star unfolding and used it for computing the shortest-path information from a single fixed point on the surface of a polytope. The nonoverlap of the unfolding [AO92], however, was not known at the time of their work.

This paper is organized as follows. In section 2, we formalize our terminology and list some basic properties of shortest paths. Section 3 defines the star unfolding and establishes some of its properties. Section 4 sketches an efficient algorithm to compute a superset of all possible shortest-path edge sequences, and in section 5 we present an algorithm for computing the exact set of these sequences; both algorithms are based on the star unfolding. In section 6 we again use the notion of star unfolding to obtain a faster algorithm for determining the geodesic diameter of a convex polytope. Section 7 deals with shortest-path queries. Section 8 contains some concluding remarks and open problems.

2. Geometric preliminaries. We begin by reviewing the geometry of shortest paths on convex polytopes.

Let $\mathcal{P}$ be the surface of a polytope with n vertices. We refer to vertices of $\mathcal{P}$ as *corners*; the unqualified terms *face* and *edge* are reserved for faces and edges of $\mathcal{P}$. We assume that $\mathcal{P}$ is triangulated. This does not change the number of faces and edges of $\mathcal{P}$ by more than a multiplicative constant but does simplify the description of our algorithms.

2.1. Geodesics and shortest paths. A path π on $\mathcal{P}$ that cannot be shortened by a local change at any point in its relative interior is referred to as a *geodesic*. Equivalently, a geodesic on the surface of a convex polytope is either a subsegment of an edge, or a path that (1) does not pass through corners, though it may possibly terminate at them, (2) is straight near any point in the interior of a face, and (3) is transverse to every edge it meets in such a fashion that it would appear straight if one were to "unfold" the two faces incident on this edge until they lie in a common plane; see, for example, Sharir and Schorr [SS86]. The behavior of a geodesic is thus fully determined by its starting point and initial direction. In the following discussion we disregard the geodesics lying completely within a single edge of $\mathcal{P}$. Given the sequence of edges a geodesic crosses and its starting and ending points, the geodesic itself can be obtained by laying out, in order, the faces that it visits in the plane so that adjacent faces share an edge and lie on opposite sides of it, and then by connecting the (images of) the two endpoints with a straight-line segment. In particular, the sequence of traversed edges together with the endpoints completely determine the geodesic.

Trivially, every shortest path along $\mathcal{P}$ is a geodesic and no shortest path meets a face or an edge more than once. We call the length of a shortest path between two points $p, q \in \mathcal{P}$ the *geodesic distance* between p and q, and we denote it by $d(p, q)$. The following additional properties of shortest paths are crucial for our analysis.

LEMMA 2.1 (see Sharir and Schorr [SS86]). *Let π_1 and π_2 be distinct shortest paths emanating from x. Let $y \in \pi_1 \cap \pi_2$ be a point distinct from x. Then either one of the paths is a subpath of the other, or neither π_1 nor π_2 can be extended past y while remaining a shortest path.*

COROLLARY 2.2. *Two shortest paths cross at most once.*

LEMMA 2.3. *If π_1, π_2 are two distinct shortest paths connecting $x, y \in \mathcal{P}$, each of the two connected components of $\mathcal{P} \setminus (\pi_1 \cup \pi_2)$ contains a corner.*

Proof. First, Lemma 2.1 implies that removal of $\pi_1 \cup \pi_2$ splits $\mathcal{P}$ into exactly two components. If one of the two components of $\mathcal{P} \setminus (\pi_1 \cup \pi_2)$ contained no corners, π_1 and π_2 would have to traverse the same sequence of edges and faces. However, there exists at most one geodesic connecting a given pair of points and traversing a given sequence of edges and faces. □

2.2. Edge sequences and sequence trees. A *shortest-path edge sequence* is the sequence of edges intersected by some shortest path π connecting two points on $\mathcal{P}$, in the order met by π. Such a sequence is *maximal* if it cannot be extended in either direction while remaining a shortest-path edge sequence; it is *half-maximal* if no extension is possible at one of the two ends. It has been shown by Schevon and O'Rourke [SO88] that the maximum total number of half-maximal sequences is $\Theta(n^3)$.

Observe that every shortest-path edge sequence σ is a prefix of some half-maximal sequence, namely, the one obtained by extending σ maximally at one end. Thus an exhaustive list of $O(n^3)$ half-maximal sequences contains, in a sense, all the shortest-path edge-sequence information of $\mathcal{P}$. More formally, given an arbitrary collection of edge sequences emanating from a fixed edge e, let the *sequence tree* Σ of this set be the tree with all distinct nonempty prefixes of the given sequences as nodes, with the trivial sequence consisting solely of e as the root, and such that σ is an ancestor of σ' in the tree if and only if σ is a prefix of σ' [HCT89]. The $\Theta(n^3)$ bound on the number of half-maximal sequences implies that the collection of $O(n)$ sequence trees obtained by considering all shortest-path edge sequences from each edge of $\mathcal{P}$ has, in turn, a total of $\Theta(n^3)$ leaves and $\Theta(n^4)$ nodes in the worst case.

2.3. Ridge trees and the source unfolding. The shortest paths emanating from a fixed source $x \in \mathcal{P}$ cover the surface of $\mathcal{P}$ in a way that can be naturally represented by "unfolding" the paths to a planar layout with respect to x. This unfolding, the "source unfolding," has been studied since the turn of the century. We will define it precisely in a moment. A second way to organize the paths in the plane is the "star unfolding," to be defined in section 3. This is not quite as natural and is of more recent lineage. Our algorithms will be built around the star unfolding, but some of the arguments do refer to the source unfolding as well.

Given two points x, y on $\mathcal{P}$, $y \in \mathcal{P}$ is a *ridge point* with respect to x if there is more than one shortest path between x and y. Ridge points with respect to x form a *ridge tree* T_x embedded on $\mathcal{P}$,[2] whose leaves are corners of $\mathcal{P}$, and whose internal vertices have degree at least three and correspond to points of $\mathcal{P}$ with three or more distinct shortest paths to x. In a degenerate situation where x lies on the ridge tree for some corner p, then p will not be a leaf of T_x, but rather will become a degree-2 vertex in T_x; so in general not all corners will appear as leaves of T_x. We define a *ridge* as a maximal subset of T_x consisting of points with exactly two distinct shortest paths to x and containing no corners of $\mathcal{P}$. These are the "edges" of T_x. Ridges are open geodesics [SS86]; a stronger characterization of ridges is given in Lemma 2.4. Fig. 1 shows two views of a ridge tree on a pyramid.

We will refer to a point $y \in \mathcal{P}$ as a *generic* point if it is not a ridge point with respect to any corner of $\mathcal{P}$. The maximal connected portion of a face (resp., an edge) of $\mathcal{P}$ consisting entirely of generic points will be called a *ridge-free region* (resp., an *edgelet*); see Fig. 2.

If we cut $\mathcal{P}$ along the ridge tree T_x and isometrically embed the resulting set in

[2] For smooth surfaces (Riemannian manifolds), the ridge tree is known as the "cut locus" [Kob67].

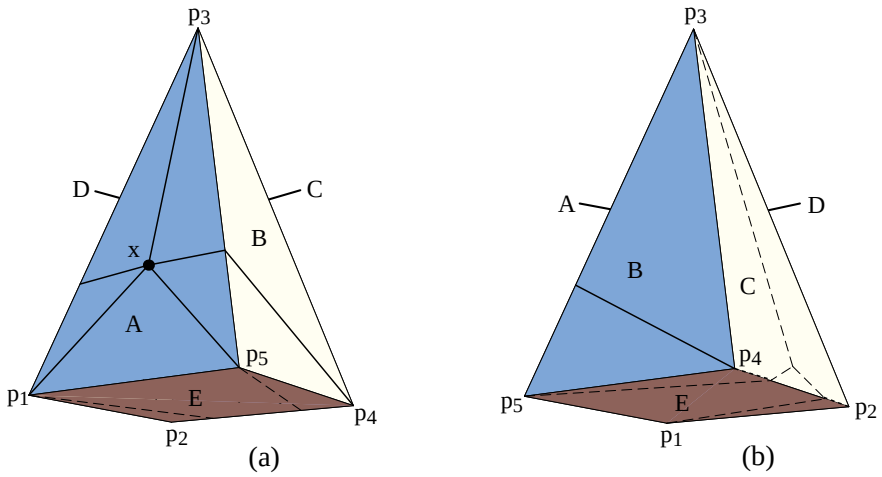


FIG. 1. *Pyramid, front* (a) *and side* (b) *views. Shortest paths to five vertices from source* x *are shown solid; the ridge tree is dashed. Coordinates of vertices are* $(\pm 1, \pm 1, 0)$ *for* p_1, p_2, p_4, p_5*, and* $p_3 = (0, 0, 4)$*;* $x = (0, 3/4, 1)$*. The ridges incident to* p_2 *and* p_4 *lie nearly on the edge* p_2p_4*.*

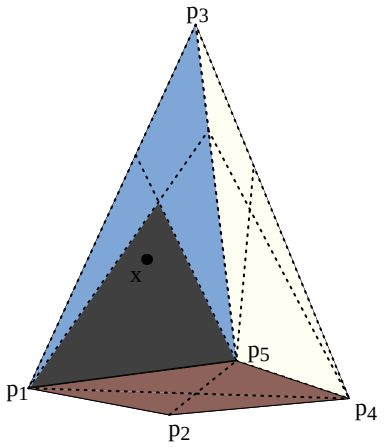


FIG. 2. *Ridge-free regions for Fig.* 1. $T_{p_1} \cup \cdots \cup T_{p_5}$ *are shown dashed (e.g.,* T_{p_3} *is the "X" on the bottom face). The ridge-free region containing* x *is shaded darker.*

$\mathbb{R}^2$, we obtain the *source unfolding* of [OS89].[3] In the source unfolding, the ridges lie on the boundary of the unfolding, while x lies at its "center," which results in a star-shaped polygon [SS86]; see Fig. 3. Let a *peel* be the closure of a connected component of the set obtained by removing from $\mathcal{P}$ both the ridge tree T_x and the shortest paths from x to all corners. A peel is isometric to a convex polygon [SS86]. Each peel's boundary consists of x, the shortest paths to two "consecutive" corners of $\mathcal{P}$, p and p', and the unique path in T_x connecting p to p'. A peel can be thought of as the collection of all the shortest paths emanating from x between xp and xp'. (The peel between xp_1 and xp_5 is shaded in Fig. 3.)

[3] The same object is called $U(\mathcal{P})$ in [SS86], "planar layout" in [Mou85], and "outward layout" in [CH90]. For Riemannian manifolds, it is the "exponential map" [Kob67].

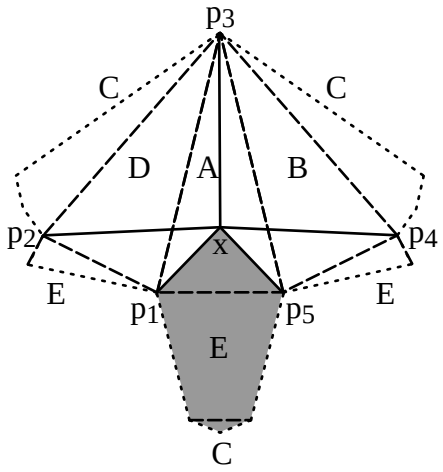


FIG. 3. *Source unfolding for the example in Fig.* 1. *Shortest paths to vertices are solid, polytope edges are dashed, and ridges are dotted. One peel is shaded.*

We need to strengthen the characterization of ridges from geodesics to shortest paths, in order to exploit Corollary 2.2. This characterization seems to be new.

LEMMA 2.4. *Every ridge of the ridge tree T_x, for any point $x \in \mathcal{P}$, is a shortest path.*

Proof. An edge π of the ridge tree is an (open) geodesic consisting of points that have two different shortest paths to x [SS86]. Suppose π is not a shortest path. Then, since it is composed of segments, it contains shortest paths, and in particular, a shortest path π' delimited by two points $a, b \in \pi$ such that there is another shortest path π'' connecting them. Refer to Fig. 4. By Lemma 2.1, $\pi' \cap \pi'' = \{a, b\}$. Let α_1 and α_2 be the two shortest paths from x to a, and let β_1 and β_2 be the two shortest paths from x to b. Notice that, by Lemma 2.1, π', α_1, α_2, β_1, β_2 do not meet except at the endpoints. In particular, we can relabel these paths so that α_2 and β_2 lie in the same connected component of $\mathcal{P} - (\alpha_1 \cup \beta_1 \cup \pi')$. There are two cases to consider.

Case 1. $x \notin \pi''$. Thus, by Lemma 2.1, π'' does not meet α_1 or α_2 except at a. Similarly, π'' does not meet β_1 or β_2 except at b. Thus, without loss of generality, we can assume that π'' lies in the portion Δ of $\mathcal{P}$ bounded by π', α_1, and β_1, and not containing α_2 or β_2. Since α_1, β_1 are shortest paths from x to a and b, respectively, their relative interiors do not intersect T_x. Moreover, π' does not contain a vertex of T_x, so Δ does not contain any corner of $\mathcal{P}$, as each corner is a vertex of T_x. On the other hand, paths $\pi', \pi'' \subset \Delta$ are distinct shortest paths connecting a to b, so by Lemma 2.3 each of the two connected components of $\mathcal{P} \setminus (\pi' \cup \pi'')$ has to contain a corner of $\mathcal{P}$. However, one of these components is entirely contained in Δ—a contradiction.

Case 2. $x \in \pi''$. As π'' and α_1 can be viewed as emanating from a and having x in common, and π'' extends past x, Lemma 2.1 implies that α_1 is a prefix of π''. Similarly, α_2 is a prefix of π'', contradicting distinctness of α_1 and α_2. □

Remark. Case 2 in the above proof is vacuous if x is a corner, which is the case in our applications of this lemma.

As defined, T_x is a tree with n vertices of degree less than 3 and thus has $\Theta(n)$ vertices and edges. However, the worst-case combinatorial size of T_x jumps from $\Theta(n)$

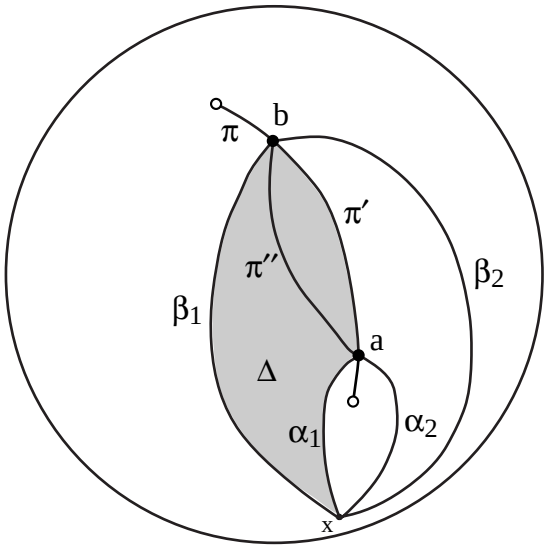


FIG. 4. *Illustration of the proof of Lemma* 2.4. *Here x is the source, and π a geodesic ridge, with $\pi' \subseteq \pi$ a shortest path. The region Δ cannot contain any vertices of $\mathcal{P}$.*

to $\Theta(n^2)$ if one takes into account the fact that a ridge is a shortest path comprised of as many as $\Theta(n)$ line segments on $\mathcal{P}$ in the worst case—and it is possible to exhibit a ridge tree for which the number of ridge-edge incidences is indeed $\Omega(n^2)$ [Mou85]. For simplicity we assume that ridges intersect each edge of $\mathcal{P}$ transversely.

3. Star unfolding. In this section we introduce the notion of the *star unfolding* of $\mathcal{P}$ and describe its geometric and combinatorial properties. Working independently, both Chen and Han [CH90, CH91] and Rasch [Ras90] have used the same notion, and in fact the idea can be found in Aleksandrov's work [Ale58, p. 226], [AZ67, p. 171].

3.1. Geometry of the star unfolding. Let $x \in \mathcal{P}$ be a generic point so that there is a unique shortest path connecting x to each corner of $\mathcal{P}$. These paths are called *cuts* and are comprised of *cut points* (see Fig. 1). If $\mathcal{P}$ is cut open along these cuts the result is a two-dimensional complex that we call the *star unfolding* S_x. If isometrically embedded in the plane, the star unfolding corresponds to a simple polygon. That the star unfolding can be embedded in the plane without overlap is by no means a straightforward claim; it was first established in [AO92] as the following lemma.

LEMMA 3.1 (see Aronov and O'Rourke [AO92]). *If viewed as a metric space with the natural definition of interior metric, S_x is isometric to a simple polygon in the plane (with the internal geodesic metric).*

The polygonal boundary ∂S_x consists entirely of edges originating from cuts. The vertices of S_x derive from the corners of $\mathcal{P}$ and from the source x. An example is shown in Fig. 5. More complex examples will be shown in Fig. 7.

The cuts partition the faces of $\mathcal{P}$ into subfaces, which map to what we call the *plates* of S_x. Since we assume the faces of $\mathcal{P}$ to be triangles, each plate is a compact convex polygon with at most six edges; see Fig. 6(b). We consider these plates to be the faces of the two-dimensional complex S_x. We assume that the complex carries with it labeling information consistent with $\mathcal{P}$.

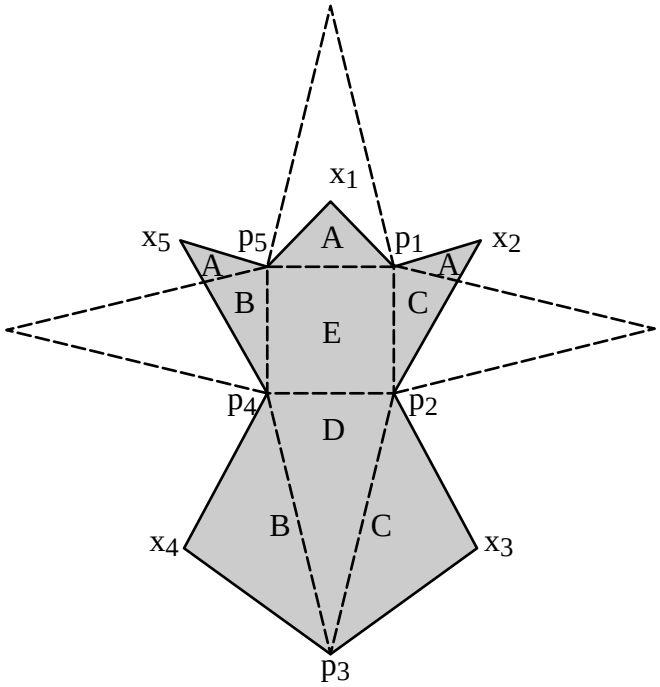


FIG. 5. *Construction of the star unfolding corresponding to Fig.* 1. *S_x is shaded. The superimposed dashed edges show the "natural" unfolding obtained by cutting along the four edges incident to p_3. The A, B, C, D, and E labels indicate portions of S_x derived from those faces.*

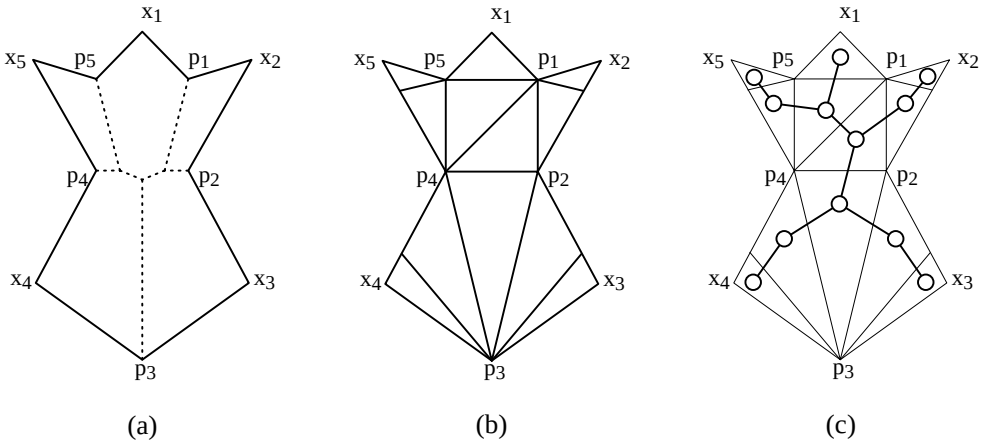


FIG. 6. (a) *Ridge tree,* (b) *plates, and* (c) *pasting tree corresponding to Fig.* 5.

Somewhat abusing the notation, we will freely switch between viewing S_x as a complex and as a simple polygon embedded in the plane. In particular, a path $\pi \subset S_x$ will be referred to as a *segment* if it corresponds to a straight-line segment in this embedding. Note that every segment in S_x is a shortest path in the intrinsic metric of the complex, but not every shortest path in S_x is a segment, as some shortest paths in S_x might bend at corners.

For $p \in \mathcal{P}$, let $U(p)$ be the set of points in S_x to which p maps; U is the "unfolding" map (with respect to x). $U(p)$ is a single point whenever p is not a cut point. A non-corner point $y \in \mathcal{P}$ distinct from x and lying on a cut has exactly two unfolded images in S_x. The corners of $\mathcal{P}$ map to single points. $X = U(x) = \{x_1, \ldots, x_n\}$ is a set of n distinct points in S_x. If $|U(p)| = 1$, then, with a slight abuse of notation, we use $U(p)$ to denote the unique image of p as well. We can extend the definition of the map U to sets in a natural way by putting $U(Q) = \bigcup_{q \in Q} U(q)$. In particular, we have the following lemma.

LEMMA 3.2 (see Sharir and Schorr [SS86]). *For a point $y \in \mathcal{P}$, any shortest path π from x to y maps to a segment $\pi^* \subset S_x$ connecting an element of $U(y)$ to an element of $U(x)$.*

There is also a view of S_x that relates it to the source unfolding: the star unfolding is just an "inside-out" version of the source unfolding, in the following sense. The star unfolding can be obtained by stitching peels together along ridges; see Fig. 6(a). The source unfolding is obtained by gluing them along the cuts; compare this with Fig. 3. (A "peel" was defined in section 2.3 as a subset of $\mathcal{P}$, but by slightly abusing the terminology we also use this term to refer to the corresponding set of points in the source or star unfolding.)

We next define the *pasting tree* Π_x as the graph whose nodes are the plates of S_x, with two nodes connected by an arc if the corresponding plates share an edge in S_x; see Fig. 6(c). For a generic point x, Π_x is a tree with $O(n^2)$ nodes, as it is the dual of a convex partition of a simple polygon without Steiner points. (If x were a ridge point of some corner, S_x would not be connected and Π_x would be a forest.) Π_x has only n leaves corresponding to the triangular plates incident to the n images of x in S_x. By Lemma 3.2, any shortest path from x to $y \in \mathcal{P}$ corresponds to a simple path in Π_x, originating at one of the leaves. Thus, the $O(n^3)$ edge sequences corresponding to the simple paths that originate from leaves of Π_x include all shortest-path edge sequences emanating from x. In fact, there are $O(n^2)$ maximal edge sequences in this set, one for each pair of leaves. This relation between Π_x and the shortest-path sequences is crucial in our sequence algorithms described in sections 4 and 5.

In the following sections we will need the concept of the "kernel" of a star unfolding. Number the corners $p_1, \ldots, p_n$ in the order in which cuts emanate from x. Number the n source images (elements of $X = U(x)$) so that ∂S_x is the cycle $x_1 p_1 x_2 \ldots p_n x_1$ comprised of $2n$ segments (see Fig. 5). The kernel is a subset of S_x; here it is more convenient to view S_x as a simple polygon. Consider the polygonal cycle $p_1 p_2 \ldots p_n p_1$. We claim that it is the boundary of a simple polygon fully contained in S_x. Indeed, each line segment $p_i p_{i+1}$[4] is fully contained in the peel sandwiched between $x_{i+1} p_i$ and $x_{i+1} p_{i+1}$. Thus, the line segments $p_i p_{i+1}$ are segments in S_x, in the sense defined above, and indeed form a simple cycle. The simple n-gon bounded by this cycle is referred to as the *kernel* K_x of the star unfolding S_x. An equivalent way of defining K_x is by removing from S_x all triangles $\triangle p_{i-1} x_i p_i$ for $i = 1, \ldots, n$. As with S_x, we will alternate between viewing K_x as a complex and as a simple polygon in the plane. Fig. 7 illustrates the star unfolding and its kernel for several randomly generated polytopes.[5] Note that neither set is necessarily star-shaped.

The main property of the kernel that we will later need is described in the following lemma.

[4] Here and thereafter $p_{n+1} = p_1$, $p_0 = p_n$, $x_{n+1} = x_1$, $x_0 = x_n$.

[5] The unfoldings were produced with code written by Julie DiBiase and Stacia Wyman of Smith College.

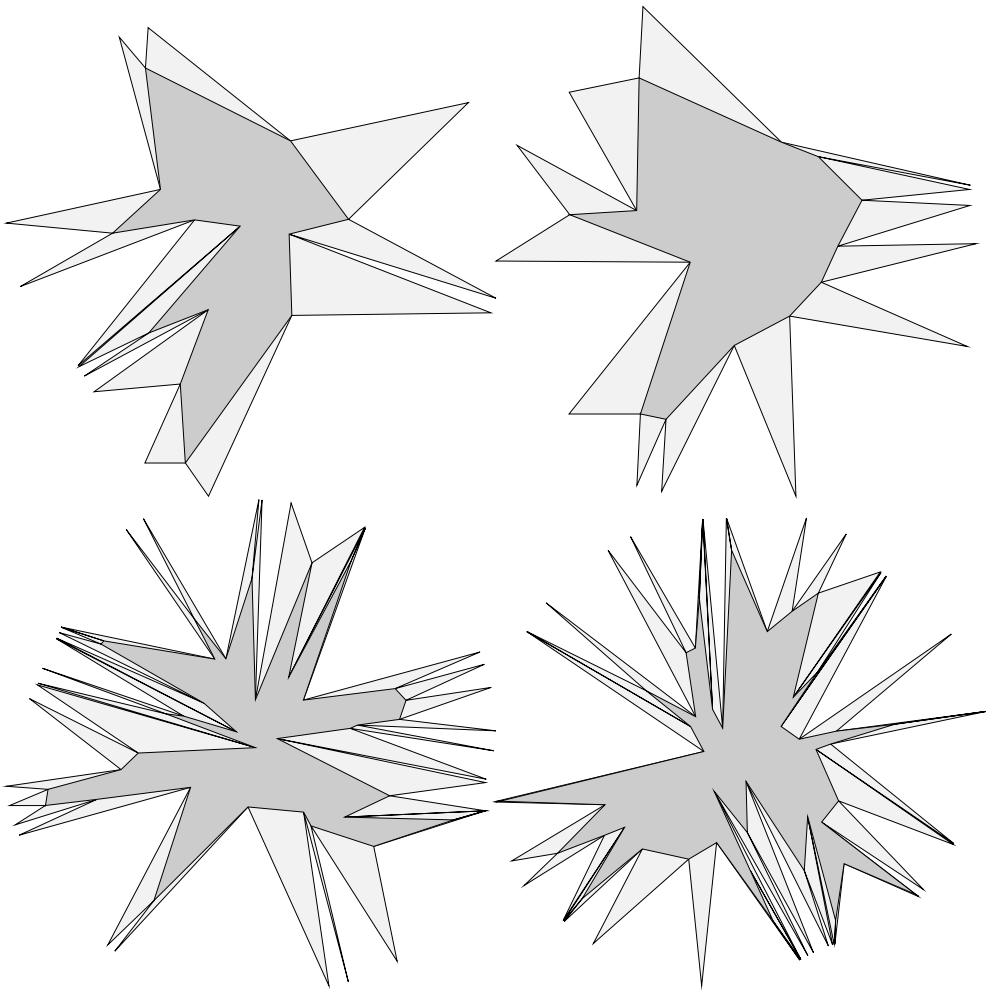


FIG. 7. *Four star unfoldings:* $n = 13, 13, 36, 42$ *vertices, left to right, top to bottom. The kernel is shaded darker in each figure.*

LEMMA 3.3. *The image of the ridge tree is completely contained within the kernel, which is itself a subset of the star unfolding:* $U(T_x) \subset K_x \subset S_x$.

Proof. Since K_x can be defined by subtraction from S_x, $K_x \subset S_x$ is immediate. The ridge tree T_x can be thought of as the union of the peel boundaries that do not come from cuts. As peels are convex, these boundaries remained when we removed triangles $\triangle p_{i-1}x_ip_i$ from S_x to form K_x. □

Aronov and O'Rourke [AO92] proved the following theorem.

THEOREM 3.4. $U(T_x)$ *is exactly the restriction of the planar Voronoi diagram of the set* $X = U(x)$ *of source images to within* K_x *or, equivalently, to within* S_x.

3.2. Structure of the star unfolding. We now describe the combinatorial structure of S_x. A *vertex* of S_x is an image of x, of a corner of $\mathcal{P}$, or of an intersection of an edge of $\mathcal{P}$ with a cut. An *edge* of S_x is a maximal portion of an image of a cut or an edge of $\mathcal{P}$ delimited by vertices of S_x. It is easy to see that S_x consists of $\Theta(n^2)$ plates in the worst case, even though its boundary is formed by only $2n$ segments, i.e.,

the images of the cuts. We define the *combinatorial structure* of S_x as the 1-skeleton of S_x, i.e., the graph whose nodes and arcs are the vertices and edges of S_x, labeled in correspondence with the labels of S_x, which are in turn derived from labels on $\mathcal{P}$. The combinatorial structure of a star unfolding has the following crucial property.

LEMMA 3.5. *Let x and y be two noncorner points lying in the same ridge-free region or on the same edgelet. Then S_x and S_y have the same combinatorial structure.*

Proof. Let f be the face containing the segment xy in its interior. The case when xy is part of an edge is similar.

As the shortest paths from any point $z \in xy$ to the corners are pairwise disjoint except at z (cf. Lemma 2.1) and z is confined to f, the combinatorial structure of S_z is uniquely determined by

(1) the circular order of the cuts around z, and
(2) the sequence of edges and faces of $\mathcal{P}$ met by each of the cuts.

We will show that (1) and (2) are invariants of S_z as long as $z \in xy$ does not cross a ridge or an edge of $\mathcal{P}$. First, the set of points of f, for which some shortest path to a fixed corner p traverses a fixed edge sequence, is convex—it is simply the intersection of f with the appropriate peel with respect to p—implying invariance of (2).

Now suppose the circular order of the cuts around z is not the same for all $z \in xy$. The initial portions of the cuts, as they emanate from any z, cannot coincide, as distinct cuts are disjoint except at z. Hence there can be a change in this circular order only if one of the vectors pointing along the initial portion of the cuts changes discontinuously at some intermediate point $z' \in xy$. However, this can happen only if z' is a ridge point with respect to a corner, and is therefore a contradiction. □

This lemma holds under more general conditions. Namely, instead of requiring that xy be free of ridge points, it is sufficient to assume that the number of distinct shortest paths connecting z to any corner does not change as z varies along xy (this number is larger than 1 if xy is a portion of a ridge).

LEMMA 3.6. *Under the assumptions of Lemma* 3.5*, K_x is isometric to K_y; i.e., they are congruent simple polygons.*

Proof. K_x is determined by the order of corners on $\partial K_x = p_1p_2, \ldots, p_np_1$ and, for each i, by the choice of the shortest path p_ip_{i+1}, if there are two or more such paths. The ordering is fixed once combinatorial structure of S_x is determined. The choice of the shortest path connecting p_i to p_{i+1} is determined by the constraint that $\triangle p_{i-1}x_ip_i$ is free of corners. □

Let R be a ridge-free region. By the above lemma, S_x can be embedded in the plane in such a way that the images of the corners of $\mathcal{P}$ are fixed for all $x \in R$, while the images of x in S_x move as x varies in $R \subseteq \mathcal{P}$. This guarantees that $K_y = K_x$ for all $x, y \in R$. This is illustrated in Fig. 8. In what follows, we will assume such an embedding of S_x and use K_R to denote K_x for all points $x \in R$. Similarly, define K_ε for an edgelet ε.

3.3. The number of different unfoldings. For the algorithms described in this paper, it will be important to bound the number of different possible combinatorial structures of star unfoldings, as we vary the position of source point x, and to compute these unfoldings efficiently (more precisely, compute their combinatorial structure plus some metric description, parameterized by the exact position of the source), as the source moves on the surface of the polytope. Two variants of this problem will be needed. In the first, we assume that the source is placed on an edge of $\mathcal{P}$, and in the second the source is placed anywhere on $\mathcal{P}$. In view of Lemma 3.5,

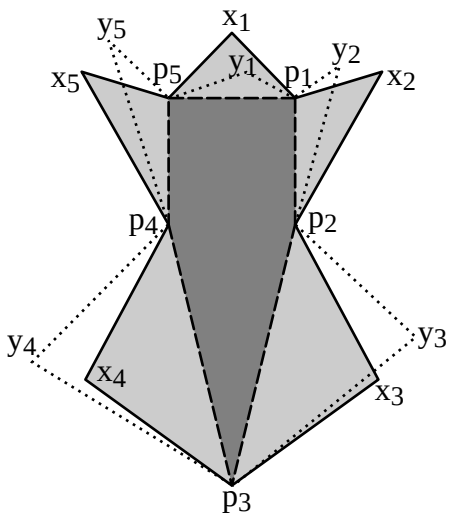


FIG. 8. *The star unfolding when the source x moves to y inside a ridge-free region. The unfolding S_x (Fig. 5) is shown lightly shaded; S_y is shown dotted. Their common kernel $K_x = K_y$ is the central dark region.*

it suffices to bound the number of edgelets and ridge-free regions, respectively.

LEMMA 3.7. *In the worst case, there are $\Theta(n^3)$ edgelets and they can be computed in $O(n^3 \log n)$ time.*

Proof. Each edge can meet a ridge of the ridge tree of a corner at most once, since ridges are shortest paths (recall that we assume that no ridge overlaps an edge—removal of this assumption does not invalidate our argument but only adds a number of technical complications). This gives an upper bound of $n \times O(n) \times O(n)$ on the number of edge-ridge intersections and therefore on the number of edgelets. An example of a convex polytope with $\Omega(n^3)$ edgelets is relatively easy to construct by modifying the lower bound construction of Mount [Mou90].

To compute the edgelets, we construct ridge trees from every corner in $n \times O(n^2)$ time by n applications of the algorithm of Chen and Han [CH90]. The edgelets are now computed by sorting the intersections with ridges along each edge. □

LEMMA 3.8. *In the worst case, there are $\Theta(n^4)$ ridge-free regions on $\mathcal{P}$. They can be computed in $\Theta(n^4)$ time.*

Proof. The overlay of n ridge trees, one from each corner of $\mathcal{P}$, produces a subdivision of $\mathcal{P}$ in which every region is bounded by at least three edges (cf. Fig. 2). Thus, by Euler's formula, the number of regions in this subdivision is proportional to the number of its vertices, which we proceed to estimate.

By Lemma 2.4 ridges are shortest paths and therefore two of them intersect in at most two points (cf. Corollary 2.2) or overlap. In the latter case no new vertex of the subdivision is created, so we restrict our attention to the former. In particular, as there are $n \times O(n) = O(n^2)$ ridges, the total number of their intersection points is $O(n^4)$. Refining this partition further by adding the edges of $\mathcal{P}$ does not affect the asymptotic complexity of the partition, as ridges intersect edges in a total of $O(n^3)$ points (cf. Lemma 3.3). This establishes the upper bound.

It is easily checked that there are $\Omega(n^4)$ ridge-free regions in Mount's example of a polytope with $\Omega(n^4)$ shortest-path edge sequences [Mou90]. Hence there are $\Theta(n^4)$

ridge-free regions on $\mathcal{P}$ in the worst case.

The ridge-free regions can be computed by calculating the ridge tree for every corner and overlaying the trees in each face of $\mathcal{P}$. The first step takes $O(n^3)$ time, while the second step can be accomplished in time $O((r+n^3)\log n) = O(n^4 \log n)$, where r is the number of ridge-free regions in $\mathcal{P}$, using the line-sweep algorithm of Bentley and Ottmann [BO79]. If computing the ridge-free regions is a bottleneck, the last step can be improved to $O(n^4)$ by using a significantly more complicated algorithm of Chazelle and Edelsbrunner [CE92]. (See also [CS89, Ba95].) □

3.4. Encoding ridge trees. In section 3.1, we proved that the combinatorial structure of S_x is the same for all points x in a ridge-free region. As x moves in a ridge-free region, the ridge tree T_x changes continuously (in the Hausdorff metric) as a subset of $\mathcal{P}$. In this subsection, we prove an upper bound on the number of different combinatorial structures of T_x as the source point x varies over a ridge-free region or an edgelet. In fact, we are interested not so much in counting the number of distinct ridge trees as we are in representing all possible ridge trees compactly to, for example, extract all vertices that ever occur in the ridge trees. Apart from being interesting in their own right, these results are needed in the algorithms described in sections 5–7.

Let R be a ridge-free region, and let x be a point in R. By Theorem 3.4, T_x is the Voronoi diagram $\mathcal{V}_x$ of the set $X = U(x)$ of images of x clipped to lie within K_R. Since ridge vertices do not lie on ∂S_x, all changes in T_x, as x varies in R, can be attributed to changes in $\mathcal{V}_x$. Thus it suffices to distinguish distinct combinatorial structures of Voronoi diagrams $\mathcal{V}_x$, $x \in R$. Here, by "combinatorial complexity" we mean an enumeration of vertices, edges, and regions of the Voronoi diagram, together with incidence relations between them.

Let $X = U(x) = \{x_1, \ldots x_n\}$. For each $x_i = (x_{i1}, x_{i2})$, let

$$f_i(y) = d(x_i, y) = \sqrt{(x_{i1} - y_1)^2 + (x_{i2} - y_2)^2}\,,$$

where (y_1, y_2) are the coordinates of a generic point y in the plane. Let

$$f(y) = \min_{1 \le i \le n} f_i(y)$$

be the *lower envelope* of the f_i's. Then $\mathcal{V}_x$ is the same as (the 1-skeleton of) the orthogonal projection of the graph of $f(y)$ onto the y_1y_2-plane, labeled with the name(s) of the function(s) attaining the minimum at each point.

We introduce an orthogonal coordinate system in R and let x have coordinates (s,t) in this system. Then positions of x_i are linear functions of s, t of the form

$$\begin{pmatrix} x_{i1} \\ x_{i2} \end{pmatrix} = \begin{pmatrix} a_i \\ b_i \end{pmatrix} + \begin{pmatrix} \cos\theta_i & \sin\theta_i \\ -\sin\theta_i & \cos\theta_i \end{pmatrix} \begin{pmatrix} s \\ t \end{pmatrix}, \tag{1}$$

where (a_i, b_i) are coordinates of x_i when x is at the origin of the (s,t) coordinate system in R, and θ_i defines the orientation of the ith image of R in the plane.

We now regard f and f_i's as 4-variate functions of s, t, y_1, y_2, and denote by M_R the (labeled) projection of the graph of f onto the (s, t, y_1, y_2)-plane. All possible combinatorial structures of $\mathcal{V}_x$, as x varies over the entire plane, are obviously encoded in M_R, as the diagram for $x = (\alpha, \beta)$ is simply the (1-skeleton) of the cross section of M_R by the 2-flat $s = \alpha$, $t = \beta$. Let

$$g_i(s, t, y_1, y_2) = (f_i(y_1, y_2))^2 - (s^2 + t^2 + y_1^2 + y_2^2)\,. \tag{2}$$

Using (1) we obtain

$$g_i(s,t,y_1,y_2) = C_i^{(0)} + C_i^{(1)}y_1 + C_i^{(2)}y_2 + C_i^{(3)}s + C_i^{(4)}t + C_i^{(5)}sy_1 + C_i^{(6)}sy_2 + C_i^{(7)}ty_1 + C_i^{(8)}ty_2, \tag{3}$$

where, for each i, the C_i's are constants that depend solely on a_i, b_i, and θ_i. Let $g(s,t,y_1,y_2)$ denote the lower envelope of the g_i's. Since $g(s,t,y_1,y_2) = (f(s,t,y_1,y_2))^2 - (s^2+t^2+y_1^2+y_2^2)$, the projection of the graph of g is the same as M_R. Let

$$v_1 = sy_1,\ v_2 = sy_2,\ v_3 = ty_1,\ v_4 = ty_2 \tag{4}$$

and set

$$\bar{g}_i(s,t,y_1,y_2,v_1,v_2,v_3,v_4) = C_i^{(0)} + C_i^{(1)}y_1 + C_i^{(2)}y_2 + C_i^{(3)}s + C_i^{(4)}t + C_i^{(5)}v_1 + C_i^{(6)}v_2 + C_i^{(7)}v_3 + C_i^{(8)}v_4. \tag{5}$$

Then every face of the graph of g is the intersection of the lower envelope of $\bar{g}_i$'s with the surface defined by equation (4). Since each $\bar{g}_i$ is an 8-variate linear function, by the upper bound theorem for convex polyhedra, the graph of their lower envelope has $O(n^4)$ faces of all dimensions (and in fact can be triangulated by using $O(n^4)$ simplices). Hence the number of faces in M_R is also $O(n^4)$. Using the algorithm of Brönnimann, Chazelle, and Matoušek [BCM94], all the faces of this lower envelope, and thus all the edges and vertices of M_R, can be computed in $O(n^4)$ time. (Using the triangulation mentioned above, one could compute a representation of all faces of M_R at the same time by intersecting each simplex of the triangulation with the surface (4). Our algorithms will need only the edges and vertices of M_R, however.) Putting everything together, we conclude with the following lemma.

LEMMA 3.9. *All the ridge trees for source points lying in a ridge-free region R can be encoded in a single lower envelope M_R whose combinatorial complexity is $O(n^4)$. Moreover, the edges and vertices of M_R can be computed in time $O(n^4)$.*

Remark. The only reason for assuming in the above analysis that x stays away from the boundary of R was to ensure that the vertices of the Voronoi diagram avoid the boundary of K_R. However, it is easy to verify that when x is allowed to vary over the closure of R, Voronoi vertices never cross the boundary of K_R but may touch it in limiting configurations. Thus the same analysis applies in that case as well.

If the source point moves along an edgelet ε rather than in a ridge-free region, we can obtain a bound on the number of different combinatorial structures of ridge trees by setting $t = 0$ in (1). Proceeding in the same way as above, each g_i now becomes

$$g_i(s,y_1,y_2) = C_i^{(0)} + C_i^{(1)}y_1 + C_i^{(2)}y_2 + C_i^{(3)}s + C_i^{(5)}sy_1 + C_i^{(6)}sy_2\,.$$

Again, we define g as the lower envelope of g_i's, and the subdivision M_ε as the labeled projection of the graph of the lower envelope g. Let $v_1 = sy_1, v_2 = sy_2$, and set

$$\bar{g}_i(s,y_1,y_2,v_1,v_2) = C_i^{(0)} + C_i^{(1)}y_1 + C_i^{(2)}y_2 + C_i^{(3)}s + C_i^{(5)}v_1 + C_i^{(6)}v_2\,. \tag{6}$$

Since $\bar{g}_i$ is now a 5-variate linear function, by the upper bound theorem, the number of faces in M_ε is $O(n^3)$. A similar bound was proved earlier in [GMR91]. Since the lower envelope of $\bar{g}_i$'s can be computed in $O(n^3)$ time [BCM94], the vertices and edges of M_ε can also be computed in time $O(n^3)$. Hence we obtain the following.

LEMMA 3.10. *All the ridge trees that occur as the source point moves on an edgelet ε can be represented as an envelope M_ε of n trivariate functions; its complexity is $O(n^3)$, and its edges and vertices can be computed in $O(n^3)$ time.*

Remark. Only a portion of M_R (resp., M_ε) is relevant to our analysis of ridge trees. Recall that it encodes the diagrams $\mathcal{V}_x$ for all x. However, we are interested only in $x \in R$ (resp., $x \in \varepsilon$) and not all of $\mathcal{V}_x$ but just the portion contained in K_R (resp., K_ε). Hence only those points $(s, t, y_1, y_2) \in M_R$ (resp., $(s, y_1, y_2) \in M_\varepsilon$) are relevant for which $(s, t) \in R$ (resp., $s \in \varepsilon$) and $(y_1, y_2) \in K_R$ (resp., $(y_1, y_2) \in K_\varepsilon$). When we make use of the information stored in M_R and M_ε at a later point in the computation, we will have to "filter out" irrelevant features. This issue is addressed later in section 5.

4. Edge sequences superset. In this section we describe an $O(n^6)$ algorithm for constructing a superset of the shortest-path edge sequences, which is both more efficient and conceptually simpler than previously suggested procedures, and which produces a smaller set of sequences.

Observe that all shortest-path edge sequences are realized by pairs of points lying on edges of $\mathcal{P}$—any other shortest path can be contracted without affecting its edge sequence so that its endpoints lie on edges of $\mathcal{P}$. Let x be a generic point lying on an edgelet ε. As mentioned in section 3.1, the pasting tree Π_x contains all shortest-path edge sequences that emanate from x. Moreover, by Lemma 3.5 Π_x is independent of choice of x in ε; therefore we will use Π_ε to denote Π_x for any point $x \in \varepsilon$. The set of $O(n^3)$ pasting trees $\{\Pi_\varepsilon \mid \varepsilon \text{ is an edgelet}\}$, each of size $O(n^2)$, contains an implicit representation of a set of $O(n^6)$ sequences ($O(n^5)$ of which are maximal in this set), which includes all shortest-path edge sequences that emanate from generic points.

ALGORITHM 1. *Sequence trees.*

```
for each edge e of P do
    Σ_e = ∅.
    for each edgelet endpoint v ∈ e do
        Compute shortest-path edge sequences Σ_v emanating from v.
        Σ_e = Σ_e ∪ Σ_v.

    for each edgelet ε ⊂ e do
        Compute Π_ε.
        for each maximal sequence σ ∈ Π_ε do
            Σ_e = Σ_e ∪ {σ}.

    T_e = the trivial sequence tree consisting of just e.
    for each sequence σ ∈ Σ_e do
        Traverse σ, augmenting T_e.
        Stop if σ visits the same edge twice.

    T_e is the sequence tree containing shortest-path edge sequences
        emanating from e.
```

Hence we can compute a superset of shortest path edge sequences in three steps: First, partition the points on the edges of $\mathcal{P}$ into $O(n^3)$ edgelets in time $O(n^3)$ as described in Lemma 3.7. Second, compute shortest-path edge sequences from the endpoints of each edgelet, using Chen and Han's shortest-path algorithm. Next, compute the star unfolding from a point in each edgelet, again using the shortest-

path algorithm. The total time spent in the last two steps is $O(n^5)$. Finally, this representation of edge sequences is transformed into $O(n)$ sequence trees, one for each edge (cf. section 2.2); see Algorithm 1 for pseudocode. For each pasting tree Π_ε, we separately traverse Π_ε from each of its leaves, so we spend $O(n^3)$ time per pasting tree. Hence the total time spent is $O(n^6)$. We thus obtain Theorem 4.1.

THEOREM 4.1. *Given a convex polytope in $\mathbb{R}^3$ with n vertices, one can construct, in time $O(n^6)$, $O(n)$ sequence trees that store a set of $O(n^6)$ edge sequences, which include all shortest-path edge sequences of $\mathcal{P}$.*

Remarks. (i) Note that our algorithm uses nothing more complex than the algorithm of Chen and Han for computing shortest paths from a fixed point, plus some sorting and tree traversals. It achieves an improvement over previous algorithms mainly by reorganizing the computation around the star unfolding.

(ii) The sequence-tree representation for just the shortest-path edge sequences is smaller by a factor of n^2 than our estimate on the size of the set produced by Algorithm 1 (cf., section 2.2), but computing it efficiently seems difficult. In addition, it is not clear how far the actual output of our algorithm is from the set of all shortest-path edge sequences. We have a sketch of a construction for a class of polytopes that force our algorithm to produce $\Omega(n^5)$ *non*–shortest-path edge sequences.

5. Exact set of shortest-path edge sequences. In this section, we present an $O(n^3\beta(n)\log n)$ algorithm for computing the exact set of maximal shortest-path edge sequences emanating from an edgelet. Here $\beta(\cdot)$ is an extremely slowly growing function asymptotically smaller than $\log^* n$. Running this algorithm for all edgelets of $\mathcal{P}$, the exact set of maximal shortest-path edge sequences can be computed in time $O(n^6\beta(n)\log n)$, which is a significant improvement over Schevon and O'Rourke's $O(n^9\log n)$ algorithm [SO89].

Let ε be an edgelet. For the purposes of this section we consider S_x embedded in the plane so that $K_x = K_\varepsilon$ does not move as x varies along ε. So on the one hand, K_ε is a fixed simple n-gon in the plane and on the other hand it is a complex constructed of $O(n^2)$ convex pieces of faces of $\mathcal{P}$. By analogy with S_x, we call these pieces *plates of* K_ε. A plate of K_ε is fully contained in a plate of S_x for any $x \in \varepsilon$. This latter plate may change its shape as x moves in ε but always corresponds to the same node of the pasting tree $\Pi_x = \Pi_\varepsilon$. Moreover, there is at most one plate of K_x in a plate of S_x, as a plate of K_x is obtained from a convex set (a plate of S_x) contained in a simple polygon (S_x) by cutting off n triangles ($\triangle x_i p_{i-1} p_i$, for $i = 1, \ldots, n$), each by a chord ($p_{i-1}p_i$).

We are interested in computing the set Σ_ε of those shortest-path edge sequences (corresponding to paths emanating from points on ε) which are maximal over all points in ε. In other words, given a sequence $\sigma \in \Sigma_\varepsilon$, there is a point $x \in \varepsilon$ and a shortest path starting from x that traverses σ, and there is no point on ε from which there is a shortest path that traverses an edge sequence that is an extension of σ. Recall that each sequence in Σ_ε corresponds to a path in the pasting tree Π_ε, originating from one of its leaves. Each leaf of Π_ε corresponds to a triangular plate incident to one of the n images of x. Let $\Sigma_{\varepsilon,i} \subseteq \Sigma_\varepsilon$ denote the set of edge sequences that originate from the ith leaf of Π_ε. If a sequence $\sigma \in \Sigma_{\varepsilon,i}$ is realized by a shortest path π emanating from $x \in \varepsilon$, then π leaves x between xp_{i-1} and xp_i, and it corresponds to a segment in S_x emanating from x_i, the ith image of x. The area swept by all of these segments (for a fixed x) is exactly the ith peel $P_{x,i}$. $P_{x,i}$ consists of the triangle $\triangle x_i p_{i-1} p_i$ and the "remainder" $\hat{P}_{x,i}$, which is the portion of the ith peel that lies in $K_x = K_\varepsilon$. If we concentrate on *maximal* shortest paths contained in $P_{x,i}$, it is sufficient to consider

those paths that end in $\hat{P}_{x,i}$, as any path that ends in $\triangle x_i p_{i-1} p_i$ can be extended to a point in $\hat{P}_{x,i}$ while remaining a shortest path. Put $C_i = \bigcup_{x \in \varepsilon} \hat{P}_{x,i}$; see Fig. 9a.

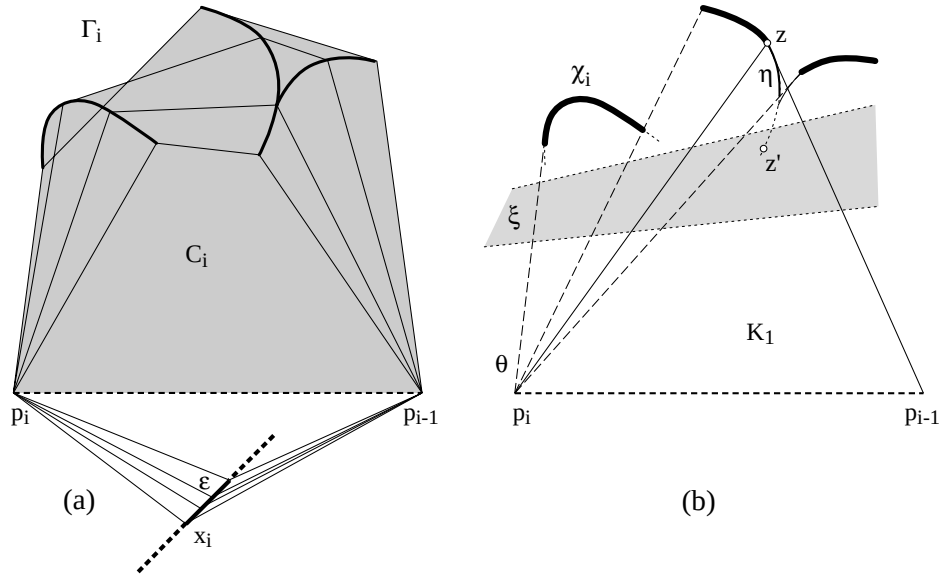


FIG. 9. (a) *C_i is the union of the peel remainders $\hat{P}_{x,i}$. Γ_i comprise the arcs traced by the ridge vertices (shown dark).* (b) *γ_i is the θ-monotone upper envelope (solid) of Γ_i with respect to p_i; several key radial lines from p_i are shown (dashed). γ_i is a superset of the "outer" envelope χ_i (dark); here the arc $\eta \in \gamma_i \setminus \chi_i$. A covering triangle $\triangle z p_{i-1} p_i$ whose apex z covers $z' \in (\cup \Gamma_i) \cap \xi$ is shown (solid).*

LEMMA 5.1. *Let $p, p' \in C_i$ be two points lying in the same plate of K_ε. Suppose $p \in \hat{P}_{x,i}$ and $p' \in \hat{P}_{x',i}$ for some $x, x' \in \varepsilon$. Then the edge sequences corresponding to the segments $x_i p$ and $x_i' p'$ are the same. The edge sequences are different if $p, p' \in C_i$ are contained in different plates of K_ε.*

Proof. Let v be the node of Π_ε corresponding to the plate of K_ε that contains p and p'. Since there is a unique path from the ith leaf to v in Π_ε, the lemma follows. Different plates of K_ε, as observed above, correspond to different nodes of Π_ε, and thus to different sequences of edges, essentially by definition of Π_ε. □

By the above lemma, each point of C_i *determines* a unique shortest-path edge sequence of $\Sigma_{\varepsilon,i}$, and all points of C_i lying in the same plate of K_ε determine the same shortest-path edge sequence of $\Sigma_{\varepsilon,i}$. We mark a node of Π_ε if the corresponding plate of K_ε intersects C_i. Let $\Pi_{\varepsilon,i}$ be the minimal subtree of Π_ε rooted at the ith leaf and containing all marked nodes. Then there is a one-to-one correspondence between the sequences of $\Sigma_{\varepsilon,i}$ and the paths in $\Pi_{\varepsilon,i}$ from its root to its leaves.

LEMMA 5.2. *Let σ be an edge sequence in $\Sigma_{\varepsilon,i}$ realized by a shortest path originating from point $x \in \varepsilon$. Then there is vertex p of $\hat{P}_{x,i}$ such that the segment $x_i p$ realizes the sequence σ.*

Proof. By Lemma 5.1, σ is realized by all points in the intersection of a unique plate ξ of K_ε with C_i. Choose a point $p \in \hat{P}_{x,i} \cap \xi$ such that $|x_i p|$ is maximum among all such points, where $|\ldots|$ denotes the Euclidean length of a segment. (Notice that the maximum must be achieved, for otherwise there is a shortest path from x_i whose sequence extends that of σ.) If p is not a vertex of $\hat{P}_{x,i}$, then we can choose a point

$p' \in \hat{P}_{x,i} \cap \xi$ in a sufficiently small neighborhood of p such that $|x_i p'| > |x_i p|$, which is a contradiction. Hence we can assume that p is a vertex of $\hat{P}_{x,i}$, as desired. □

In view of this lemma, we can restrict our attention to the points of C_i that correspond to the vertices of $\hat{P}_{x,i}$, for any $x \in \varepsilon$. Recall that each vertex of $\hat{P}_{x,i}$ is the image of a ridge vertex. A typical ridge vertex v is incident to three open peels c_j, c_k, c_ℓ; if v has degree more than three and exists at more than just a discrete set of positions of $x \in \varepsilon$, replace the triple of incident peels with a larger tuple in the following discussion. As x moves along ε, the vertex traces an algebraic curve $v = v(x)$ in K_ε. Let a *lifetime* of a ridge vertex v be a maximal connected interval $\varepsilon' \subseteq \varepsilon$ for which $x \in \varepsilon'$ implies that v is a vertex of T_x. Let Γ_i be the set of arcs traced out by ridge vertices that appear on the boundary of $\hat{P}_{x,i}$ during their lifetimes (Fig. 9a); set $n_i = |\Gamma_i|$. It can be verified that the arcs in Γ_i corresponding to a ridge vertex, defined by the triple c_j, c_k, c_ℓ, are the projections onto the xy-plane of those edges of the subdivision M_ε, defined in section 3.4, along which g_j, g_k, g_ℓ simultaneously appear on the lower envelope g. (As we mentioned in section 3.4, M_ε may contain "irrelevant" features. In particular, we must first truncate each aforementioned arc so that it corresponds to positions of the source on ε. Second, we must verify that the Voronoi diagram vertex corresponding to the arc indeed yields a ridge vertex. It is sufficient to check, for a single point of the curve traced out by the vertex as x ranges over ε, that it lies inside K_ε, as a ridge vertex cannot leave K_ε. This is easily accomplished by one point-location query per arc.)

We have previously observed that the maximal sequences of $\Pi_{\varepsilon,i}$ are necessarily realized by points on $\partial C_i \setminus p_{i-1}p_i$. In particular, points that lie in the interior of C_i can be safely disregarded. We will now apply a similar procedure to points of $\cup\Gamma_i$. If we introduce polar coordinates with p_i as the origin, each arc $\eta_j \in \Gamma_i$ can be regarded as a univariate (partial) function $r = \eta_j(\theta)$ (split η_j into a constant number of θ-monotone arcs if it is not θ-monotone; this is possible since η_i is a portion of an algebraic curve of small degree). Consider the graph of the upper envelope γ_i of the functions $\eta_j(\theta)$ (Fig. 9b). Since each arc in Γ_i is algebraic of constant degree, the upper envelope γ_i has $O(n_i\beta(n_i))$ breakpoints [ASS89]; here $\beta(k) = 2^{\alpha^s(k)}$, s is a constant depending on the maximum degree of arcs in Γ_i, and $\alpha(\cdot)$ is the inverse Ackermann function. Using a divide-and-conquer approach, the upper envelope can be computed in time $O(n_i\beta(n_i)\log n_i)$; see [SA95].

We will now show that tracing γ_i through K_ε is sufficient for computing $\Pi_{\varepsilon,i}$—there is no need to examine the entire $\cup\Gamma_i$.

LEMMA 5.3. *Each sequence in $\Sigma_{\varepsilon,i}$ is determined by a point on γ_i.*

Proof. We will show in fact that points on a subset χ_i of γ_i suffice to determine all sequences. Say a point z "covers" a point $z' \neq z$ if the triangle $\triangle z p_{i-1} p_i$ contains z'. Call the set of points of $\cup\Gamma_i$ not covered by any point of $\cup\Gamma_i$ its *outer envelope* χ_i. It is evident that $\chi_i \subseteq \gamma_i$; see Fig. 9b.

Suppose there is a sequence $\sigma \in \Sigma_{\varepsilon,i}$ not realized by any point on χ_i. Let ξ be the plate of K_ε corresponding to σ. Thus some arc of Γ_i meets ξ, but χ_i does not. Hence every point of $(\cup\Gamma_i) \cap \xi$ is covered by a point of χ_i. Pick a point $z \in \chi_i$ that covers some point $z' \in (\cup\Gamma_i) \cap \xi$; see Fig. 9b. By the choice of z', there is an $x \in \varepsilon$ such that the segment xz' corresponds to a shortest path on $\mathcal{P}$. Consider $K_\varepsilon \setminus \xi$. It consists of two or three simple polygons, one of which, say K_1, is incident to $p_{i-1}p_i$ (the case when ξ touches $p_{i-1}p_i$ is slightly different and can be handled by an easier argument). We claim that z does not lie in K_1. Indeed, $\triangle z' p_{i-1} p_i$ is such that both $p_{i-1}z'$ and $p_i z'$ enter ξ and remain there. As $\triangle z p_{i-1} p_i$ contains $\triangle z' p_{i-1} p_i$, removal

of ξ separates K_ε, and $z \notin \xi$ and both $p_{i-1}z$ and p_iz must cross ξ and exit it. As $z \in \chi_i$, there is an $x' \in \varepsilon$ so that the segment $x'z$ is (the image of) a shortest path. However, since $z \notin K_1$, this path crosses ξ, so the edge sequence it traverses is an extension of σ, contradicting maximality of σ.

We have shown that each sequence is realized by a point of χ_i and, therefore, of γ_i. □

If a plate of K_ε is intersected by an edge of γ_i, we call the corresponding node of Π_ε *visited* by γ_i. The above lemma implies that the minimal subtree containing the node corresponding to x_i and all nodes visited by γ_i is the same as $\Pi_{\varepsilon,i}$. It is thus sufficient to determine the nodes of Π_ε visited by γ_i.

ALGORITHM 2. *Exact edge sequences.*

```
Form edgelets.
for each edgelet ε do
    Compute M_ε.
    Γ := Set of projections of edges in M_ε.
    Eliminate irrelevant features from Γ.
    for each image x_i do
        Γ_i := Arcs of Γ at which g_i appears on the lower envelope.
        Compute upper envelope γ_i of Γ_i.
        Compute and triangulate K_ε[p_i].
        Compute the connected arcs ξ_1, ..., ξ_u.
        for each △ in the triangulation do
            Compute Π_△ := Π_ε ∩ △.
            Compute Π^1_△, Π^2_△.
            for each arc ξ_j ⊂ △ do
                Find nodes of Π^1_△, Π^2_△ visited by ξ_j.
```

Note that γ_i is θ-monotone with respect to p_i, and therefore γ_i lies in the portion $K_\varepsilon[p_i]$ of K_ε visible from p_i. Compute $K_\varepsilon[p_i]$, in linear time [EA81], and triangulate $K_\varepsilon[p_i]$ into a linear number of triangles all incident to p_i. Now partition γ_i into connected portions $\xi_1, \ldots, \xi_u$, each fully contained in one of these triangles. This can be done in time proportional to the number of breakpoints in γ_i and the number of triangles involved and produces at most $O(n)$ extra arcs, as γ_i is θ-monotone (with respect to p_i) and thus crosses each segment separating consecutive triangles in at most one point. Since each triangle $\triangle$ is fully contained in K_ε and thus encloses no images of a vertex of $\mathcal{P}$, the set of plates of Π_ε met by $\triangle$ corresponds to a subtree $\Pi_\triangle$ of Π_ε of linear size, with at most one vertex of degree 3 and all remaining vertices of degree at most 2. Hence $\Pi_\triangle$ can be covered by two simple paths $\Pi^1_\triangle, \Pi^2_\triangle \subseteq \Pi_\triangle$, and they can be computed in linear time. For each $\xi_j \subset \triangle$, we determine the furthest node that ξ_j reaches in $\Pi^1_\triangle, \Pi^2_\triangle$ by binary search. Recall that ξ_j consists of a number of algebraic arcs of constant degree. An intersection between ξ_j and a plate of K_ε can be detected in $O(1)$ time per such arc, so the binary search requires only $O(\log n)$ time per arc. The total time spent is thus $O(n_i\beta(n)\log n + n^2)$ over all triangles of $K_\varepsilon[p_i]$. Repeating this procedure over all n leaves of Π_ε, the total time spent in computing Σ_ε is $O(n^3\beta(n)\log n)$, as $\sum_i n_i = O(n^3)$ by Lemma 3.10. The above processing is repeated for each of the $O(n^3)$ edgelets ε. This completes the description of the algorithm. It is summarized in Algorithm 2.

THEOREM 5.4. *The exact set of all shortest-path edge sequences on the surface*

of a 3-polytope on n vertices can be computed in $O(n^6\beta(n)\log n)$ time, where $\beta(n) = o(\log^* n)$ is an extremely slowly growing function.

6. Geodesic diameter. In this section we present an $O(n^8 \log n)$ time algorithm for computing the geodesic diameter of $\mathcal{P}$. As mentioned in the introduction, this question was first investigated by O'Rourke and Schevon [OS89] who presented an $O(n^{14} \log n)$ time algorithm for computing it. Their algorithm relies on the following proposition.

LEMMA 6.1 (see O'Rourke and Schevon [OS89]). *If a pair of points $x, y \in \mathcal{P}$ realizes the diameter of $\mathcal{P}$, then either x or y is a corner of $\mathcal{P}$, or there are at least five distinct shortest paths between x and y.*

Lemma 6.1 suggests the following strategy for locating all diametral pairs. We first dispose of the possibility that either x or y is a corner in $n \times O(n^2) = O(n^3)$ time just as in [OS89]. Next, we fix a ridge-free region R and let M_R be the subdivision defined in section 3.4. We need to compute all pairs of points $x \in \mathrm{cl}\,(R)$ and $y^* \in K_x$ such that there are at least five distinct shortest paths between x and y, with $U(y) = y^*$. By a result of Schevon [Sch89], such a pair x, y can be a diametral pair only if it is the only pair, in a sufficiently small neighborhood of x and y, with at least five distinct shortest paths between them. Such a pair of points corresponds to a vertex of M_R. Hence we use the following approach.

We first compute, in $O(n^4)$ time, all ridge-free regions of $\mathcal{P}$ (cf. Lemma 3.8). Next, for each ridge-free region R, we compute K_R, vertices of M_R, and $f(v)$ for all vertices of M_R (recall that $f(v)$ is the shortest distance from v to any source image; cf. section 3.3). Next, for each vertex $v = (s, t, y_1, y_2)$ of M_R, we determine whether (s, t) lies in the closure of R and $(y_1, y_2) \in K_R$. If the answer to both of these questions is "yes," we add v to the list of candidates for diametral pairs. (This step is exactly the elimination of "irrelevant features" mentioned at the end of section 3.4. Once the two conditions are verified, we know that (s, t) and (y_1, y_2) correspond to actual points x, y on $\mathcal{P}$ and $f(v)$ is exactly $d(x, y)$.)

Finally, among all diametral candidate pairs, we choose a pair that has the largest geodesic distance. See Algorithm 3 for the pseudocode.

For each ridge-free region R, K_R can be computed in time $O(n^2)$ and preprocessed for planar point location in additional $O(n \log n)$ time using the algorithm of Sarnak and Tarjan [ST86]. (Once again, recall that we treat K_R as a simple polygon and use (y_1, y_2)-coordinate system there.) By Lemma 3.9, vertices of M_R and $f(v)$, for all vertices of M_R, can be computed in time $O(n^4)$. We spend $O(\log n)$ time for point location at each vertex of M_R, so the total time spent is $O(n^8 \log n)$.

THEOREM 6.2. *The geodesic diameter of a convex polytope in $\mathbb{R}^3$ with n vertices can be computed in time $O(n^8 \log n)$.*

7. Shortest-path queries. In this section we discuss the preprocessing needed to support queries of the following form: "Given $x, y \in \mathcal{P}$, determine $d(x, y)$." We assume that each face ϕ of $\mathcal{P}$ has its own coordinate system (e.g., a vertex of ϕ is regarded as the origin and the two edges of ϕ incident to it are regarded as the two axes), and that a point $p \in \mathcal{P}$ is specified by the face ϕ containing p and by the ϕ-coordinates of p. Two variants of the query problem are considered: (1) no assumption is made about x and y, and (2) x is assumed to lie on an edge of $\mathcal{P}$.

Our data structure is based on the following observations. Let $x, y \in \mathcal{P}$ be two query points. Suppose $x = (s, t)$ is a generic point lying in a ridge-free region R and

ALGORITHM 3. *Geodesic diameter.*

```
for each corner c of P do
    Construct the ridge tree T_c with respect to c.
    for each vertex v of T_c do
        Add d(c, v) to the list of diameter candidates.
Compute the ridge-free regions.
for each ridge-free region R do
    Compute M_R and f(v) for all vertices v ∈ M_R.
    Compute S_x for some x ∈ R.
    Construct K_R = K_x.
    Preprocess K_R for point location queries.
    Preprocess cl(R) for point location queries.
    for each vertex v = (s, t, y_1, y_2) of M_R do
        if (s, t) ∈ cl(R) and (y_1, y_2) ∈ K_x then
            Add f(v) = d((s, t), (y_1, y_2)) to the list of diameter candidates.
Find a diametral candidate pair with the maximum geodesic distance.
```

$y^* = (y_1, y_2)$ is an image of y in S_x. If y^* lies in the kernel K_R, then

$$d(x,y) = f(s,t,y_1,y_2) = \left(g(s,t,y_1,y_2) + (s^2+t^2+y_1^2+y_2^2)\right)^{1/2},$$

where

$$g(s,t,y_1,y_2) = \min_{1\le i\le n} \bar{g}_i(s,t,y_1,y_2,v_1,v_2,v_3,v_4),$$

as defined in equations (2), (4), and (5). Let H_R be the set of hyperplanes in $\mathbb{R}^9$ corresponding to the graphs of $\bar{g}_i$'s (cf. equation (5))

(7)
$$h_i : v_5 = C_i^{(0)} + C_i^{(1)} y_1 + C_i^{(2)} y_2 + C_i^{(3)} s + C_i^{(4)} t + C_i^{(5)} v_1 + C_i^{(6)} v_2 + C_i^{(7)} v_3 + C_i^{(8)} v_4.$$

Then, computing the value of $g(s,t,y_1,y_2)$ is the same as determining the first hyperplane of H_R intersected by the vertical ray emanating from the point

$$(s,t,y_1,y_2,sy_1,sy_2,ty_1,ty_2,-\infty)$$

in the positive v_5-direction. The desired value is $(v_5 + s^2 + t^2 + y_1^2 + y_2^2)^{1/2}$, where v_5 is the v_5-coordinate of the intersection point.

On the other hand, if $y^* \notin K_x$, then it lies in one of the triangles $\triangle p_{i-1}x_ip_i$ and $d(x,y) = |x_iy^*|$. For a ridge-free region R, let κ_R denote the preimage of ∂K_R on $\mathcal{P}$, i.e., $U(\kappa_R) = \partial K_R$. The following lemma is crucial in answering queries when $y \notin U^{-1}(K_R)$.

LEMMA 7.1. *Let R be a ridge-free region or an edgelet, let ϕ be a face of $\mathcal{P}$, and let Δ be a connected component of $\phi \setminus \kappa_R$ whose image is not contained in K_R. Then the sequence of edges traversed by the shortest-path $\pi(x,y)$ is independent of the choice of $x \in R$ and $y \in \Delta$.*

Proof. For the sake of contradiction, suppose there are two points $y, y' \in \Delta$ such that the sequences of edges traversed by $\pi(x,y')$ and $\pi(x,y'')$ are distinct. Then there

must exist a point $y \in y'y''$ with two shortest paths to x—to obtain such a point, move y from one end of $y'y''$ to the other and observe that the shortest path from x to y changes continuously and maintains the set of edges of $\mathcal{P}$ that it meets, *except* at points y with more than one shortest path to x. Thus $y \in T_x$, so $U(y) \subset U(T_x) \subset K_R$. However, the segment $y'y'' \subset \Delta$ as Δ is convex, implying $U(y) \subset U(\Delta) \subset S_x \setminus K_R$, which is a contradiction.

Similarly, if $x', x'' \in R$ are such that the paths connecting these two points to $y \in \Delta$ traverse different edge sequences, there must exist $x \in x'x''$, which is connected to y by two shortest paths, again forcing y onto T_x and yielding a contradiction. The lemma follows easily. □

Data structure Based on the above observations, we can preprocess $\mathcal{P}$ as follows. We partition every face ϕ of $\mathcal{P}$ into ridge-free regions in time $O(n^4)$ (see Lemma 3.8), and preprocess the resulting subdivision of ϕ for planar point-location queries using any standard algorithm [ST86]. The queries would use ϕ-coordinates. The total time spent in this step is $O(n^4 \log n)$.

Let R be a fixed ridge-free region. We construct the following data structures for R. Choose an arbitrary point $x \in R$. Compute $K_x = K_R$ and the connected components of $\phi \setminus \kappa_R$ for each face ϕ of $\mathcal{P}$. Again, we preprocess the resulting subdivision of each face for planar point-location queries in ϕ-coordinates. We label each component Δ of $\phi \setminus \kappa_R$ as to whether it lies in $U^{-1}(K_R)$. If it does not, we choose a point $y \in \Delta$ and compute the edge sequence σ for the shortest path from x to y. By Lemma 7.1, σ is the same for all pairs $x \in R$ and $y \in \Delta$. We also compute the transformation, corresponding to the edge sequence σ, which maps the ϕ-based coordinates of points in Δ to ϕ'-coordinates of the face of $\mathcal{P}$ containing R. This corresponds to laying out in the plane the faces prescribed by σ from ϕ' to ϕ so that $d(x, y)$ becomes the length of the straight-line segment connecting x and y. All transformations for regions $\Delta \not\subset U^{-1}(K_R)$ can be computed in $O(n^2)$ time by a single depth-first traversal of the shortest-path sequence tree from x, computed by the algorithm of Chen and Han. If, on the other hand, Δ lies in $U^{-1}(K_R)$, the exact sequence of edges traversed by a shortest path from $x \in R$ to $y \in \Delta$ depends on the choice of x and y; the structure for determining it is described below. (Note that such sets Δ correspond exactly to "plates of K_R" as in section 5.) However, in this case any $y \in U^{-1}(K_R)$ has a unique image $y^* \in K_R$, so for each $\Delta \subset U^{-1}(K_R)$ we compute the coordinate transformation U from the ϕ-coordinates, where ϕ is the face of $\mathcal{P}$ containing Δ, to the coordinates in the planar embedding of K_R (they were referred to as (y_1, y_2)-coordinates in section 3.4). The sequences σ and the coordinate transformations U, for all $\Delta \subset U^{-1}(K_R)$, can be computed in $O(n^2)$ time, by performing a depth-first search on Π_x (each node of Π_x corresponds to a connected component Δ).

Next, let H_R be the set of hyperplanes defined in (7). We preprocess H_R into a data structure, so that the first hyperplane of H intersected by a vertical ray emanating from a point with $v_5 = -\infty$ can be computed efficiently. Matoušek and Schwarzkopf [MS93] (also see [AM92]) have proposed such a data structure, which, given a parameter $n \le u \le n^4$, can preprocess H_R, in time $O(u^{1+\delta})$, into a data structure of size $O(u^{1+\delta})$, so that a ray-shooting query can be answered in time $O(\frac{n}{u^{1/4}} \log n)$. This completes the description of the data structures for R. We construct these data structures for each ridge-free region R.

Since there are $O(n^4)$ ridge-free regions, the total time spent in constructing the data structures is $O(n^4(n^2 + u^{1+\delta}))$.

Answering a query. Let $x, y \in \mathcal{P}$ be a query pair. Let ϕ_x, ϕ_y be the faces of $\mathcal{P}$ containing x and y, respectively. Assume first that x is a generic point. By locating x in the point location data structure for ϕ_x, we identify in $O(\log n)$ time the ridge-free region R that contains x. Next, we determine the connected component Δ of $\phi_y \setminus \kappa_R$ that contains y by point location in ϕ_y. If $\Delta \cap U^{-1}(K_R) = \emptyset$, we can use the transformation stored at Δ to compute $d(x, y)$ in $O(1)$ time. If $\Delta \subset U^{-1}(K_R)$, using the second data structure we compute the first hyperplane h of H hit by the ray emanating from $(a_x, b_x, a_y, b_y, a_x a_y, a_x b_y, b_x a_y, b_x b_y, -\infty)$ in the $+v_5$-direction, where (a_x, b_x) and (a_y, b_y) are the coordinates of x and y, respectively. The coordinates of x are in the ϕ_x-coordinate system and the coordinates of y are in the coordinates system associated with the unfolding of K_R—the coordinate transformation from ϕ_y to (y_1, y_2) is stored at Δ. Once we know h, $d(x, y) = (g(a_x, b_x, a_y, b_y) + a_x^2 + a_y^2 + b_x^2 + b_y^2)^2$ can be computed in $O(1)$ time. The total time required is $O((n/u^{1/4}) \log n)$.

Finally, if x is not a generic point then, as mentioned in the remark following Lemma 3.9, we can use the data structures of any of the ridge-free regions whose boundaries contain x. It is easy to see by a continuity argument that all shortest paths from such a point are encoded equally well in the data structures of all of the ridge-tree regions touching x.

Hence setting $u = n^2 m$, we can conclude with the following theorem.

THEOREM 7.2. *Given a polytope $\mathcal{P}$ in $\mathbb{R}^3$ with n vertices and a parameter $1 \leq m \leq n^2$, one can construct, in time $O(n^6 m^{1+\delta})$ for any $\delta > 0$, a data structure of size $O(n^6 m^{1+\delta})$, so that $d(x, y)$ for any two points $x, y \in \mathcal{P}$ can be computed in time $O((\sqrt{n}/m^{1/4}) \log n)$. Constants of proportionality depend on δ.*

If x always lies on an edge, then H is a set of hyperplanes in $\mathbb{R}^6$, so the query time of the analogous vertical ray-shooting data structure in six dimensions is $O(n/u^{1/3} \log n)$ for $n \leq u \leq n^2$. Moreover, we have to construct only $O(n^3)$ different data structures, one for each edgelet, so we can conclude with the following theorem.

THEOREM 7.3. *Given a polytope $\mathcal{P}$ in $\mathbb{R}^3$ with n vertices and a parameter $1 \leq m \leq n$, one can construct, in time $O(n^5 m^{1+\delta})$ for any $\delta > 0$, a data structure of size $O(n^5 m^{1+\delta})$, so that for any two points $x, y \in \mathcal{P}$ such that x lies on an edge of $\mathcal{P}$ one can compute $d(x, y)$ in time $O((n/m)^{1/3} \log^2 n)$.*

8. Discussion and open problems. We have shown that use of the star unfolding of a polytope leads to substantial improvements in the time complexity of three problems related to shortest paths on the surface of a convex polytope: finding edge sequences, computing the geodesic diameter, and distance queries. Moreover, the algorithms are not only theoretical improvements, but also, we believe, conceptual simplifications. This demonstrates the utility of the star unfolding.

We conclude by mentioning some open problems:

1. Can one obtain an upper bound on the number of different combinatorial structures of ridge trees better than $O(n^4)$? Such an improvement would yield a similar improvement in the time complexities of diameter and exact shortest-path edge sequences algorithms.
2. Can one answer a shortest-path query faster if *both* x and y lie on some edge of $\mathcal{P}$? This special case is important for planning paths among convex polyhedra (see Sharir [Sha87]).

Acknowledgment. We thank the referees for comments that led to significant improvements in the presentation of this paper.

REFERENCES

[AAOS90] P. K. Agarwal, B. Aronov, J. O'Rourke, and C. Schevon, *Star unfolding of a polytope with applications*, in Proc. of 2nd Annual Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comput. Sci. 447, Springer-Verlag, Berlin, 1990, pp. 251–263.

[Ale58] A. D. Aleksandrov, *Konvexe Polyeder*, Math. Lehrbucher und Monographien, Akademie-Verlag, Berlin, 1958.

[AM92] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, SIAM J. Comput., 22 (1993), pp. 794–806.

[AO92] B. Aronov and J. O'Rourke, *Nonoverlap of the star unfolding*, Discrete Comput. Geom., 8 (1992), pp. 219–250.

[ASS89] P. K. Agarwal, M. Sharir, and P. Shor, *Sharp upper and lower bounds on the length of general Davenport-Schinzel sequences*, J. Comb. Theory Ser. A, 52 (1989), pp. 228–274.

[AZ67] A. D. Aleksandrov and V. A. Zalgaller, *Intrinsic Geometry of Surfaces*, American Mathematical Society, Providence, RI, 1967. (Translation of the 1962 Russian original.)

[Ba95] I. Balaban, *An optimal algorithm for finding segments intersections*, in Proc. 11th Annual ACM Sympos. Comput. Geom., ACM, New York, 1995, pp. 211–219.

[BO79] J. L. Bentley and T. A. Ottmann, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., C-28 (1979), pp. 643–647.

[BCM94] H. Brönnimann, B. Chazelle, and J. Matoušek, *Product range spaces, sensitive sampling, and derandomization*, in Proc. 34th Annual IEEE Sympos. Found. Comput. Sci., IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 400–409.

[CR87] J. Canny and J. Reif, *New lower bound techniques for robot motion planning problems*, in Proc. 28th IEEE Symp. Found. Comput. Sci., IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 49–60.

[CE92] B. Chazelle and H. Edelsbrunner, *An optimal algorithm for intersecting line segments in the plane*, J. Assoc. Comput. Mach., 39 (1992), pp. 1–54.

[CH90] J. Chen and Y. Han, *Shortest paths on a polyhedron*, in Proc. 6th Annual ACM Sympos. Comput. Geom., ACM, New York, 1990, pp. 360–369.

[CH91] J. Chen and Y. Han, *Storing shortest paths for a polyhedron*, in Advances in Computing and Information—ICCI '91 Internat. Conf. Proc., Springer-Verlag, Berlin, 1991, pp. 169–80.

[CSY94] J. Choi, J. Sellen, and C.-K. Yap, *Approximate Euclidean shortest path in* 3*-space*, in Proc. 10th Annual ACM Sympos. Comput. Geom., ACM, New York, 1994, pp. 41–48.

[Cla87] K. Clarkson, *Approximation algorithms for shortest path motion planning*, in Proc. 19th Annual ACM Sympos. Theory Comput., ACM, New York, 1987, pp. 56–65.

[CS89] K. Clarkson and P. Shor, *Applications of random sampling in computational geometry*, II, Discrete Comput. Geom., 4 (1989), pp. 387–421.

[EA81] H. El Gindy and D. Avis, *A linear algorithm for computing the visibility polygon from a point*, J. Algorithms, 2 (1981), pp. 186–197.

[Gar61] M. Gardner, *The* 2*nd Scientific American Book of Mathematical Puzzles and Diversions*, Simon and Schuster, New York, 1961.

[GMR91] L. Guibas, J. S. B. Mitchell, and T. Roos, *Voronoi diagrams of moving points in the plane*, in Proc. 17th Internat. Workshop Graph-Theoret. Concepts Comput. Sci., Lecture Notes in Comput. Sci. 570, Springer-Verlag, Berlin, 1991, pp. 113–125.

[HS93] J. Hershberger and S. Suri, *Efficient computation of Euclidean shortest paths in the plane*, in Proc. 34th Annual IEEE Sympos. Found. Comput. Sci., IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 508–517.

[HS95] J. Hershberger and S. Suri, *Practical methods for approximating shortest paths on a convex polytope in* R^3, in Proc. 6th Annual ACM–SIAM Sympos. Discrete Algorithms, SIAM, Philadelphia, 1995, pp. 447–456.

[HCT89] Y.-H. Hwang, R.-C. Chang, and H.-Y. Tu, *Finding all shortest path edge sequences on a convex polyhedron*, in Proc. 1st Workshop Algorithms Data Struct., Lecture Notes in Comput. Sci 382, Springer-Verlag, Berlin, 1989, pp. 251–266.

[Kob67] S. Kobayashi, *On conjugate and cut loci*, in Studies in Global Geometry and Analysis, S. S. Chern, ed., Mathematical Association of America, Providence, RI, 1967, pp. 96–122.

[Mou85] D. Mount, *On Finding Shortest Paths on Convex Polyhedra*, Technical Report 1495,

Dept. of Comput. Sci., Univ. of Maryland, 1985.

[Mit93] J. S. B. Mitchell, *Shortest paths among obstacles in the plane*, in Proc. 9th Annual ACM Sympos. Comput. Geom., ACM, New York, 1993, pp. 308–317.

[MMP87] J. Mitchell, D. Mount, and C. Papadimitriou, *The discrete geodesic problem*, SIAM J. Comput., 16 (1987), pp. 647–668.

[Mou90] D. M. Mount, *The number of shortest paths on the surface of a polyhedron*, SIAM J. Comput., 19 (1990), pp. 593–611.

[MS93] J. Matoušek and O. Schwarzkopf, *Ray shooting in convex polytopes*, Discrete Comput. Geom., 10 (1993), pp. 215–232.

[OS89] J. O'Rourke and C. Schevon, *Computing the geodesic diameter of a 3-polytope*, in Proc. 5th Annual ACM Sympos. Comput. Geom., ACM, New York, 1989, pp. 370–379.

[Pap85] C. Papadimitriou, *An algorithm for shortest paths motion in three dimensions*, Inform. Process. Lett., 20 (1985), pp. 259–263.

[Ras90] R. Rasch, *Shortest Paths Along a Convex Polyhedron*, Diploma thesis, Univ. of Saarland, Saarbrücken, Germany, 1990.

[RS89] J. Reif and J. Storer, *Shortest paths in Euclidean space with polyhedral obstacles*, J. Assoc. Comput. Mach., 41 (1994), pp. 1013–1019.

[Sch89] C. Schevon, *Algorithms for Geodesics on Polytopes*, Ph.D. thesis, Johns Hopkins Univ., Baltimore, MD, 1989.

[Sei81] R. Seidel, *A Convex Hull Algorithm Optimal for Point Sets in Even Dimensions*, Technical Report 81/14, Dept. Comput. Sci., Univ. of British Columbia, Vancouver, 1981.

[Sha87] M. Sharir, *On shortest paths amidst convex polyhedra*, SIAM J. Comput., 16 (1987), pp. 561–572.

[SO88] C. Schevon and J. O'Rourke, *The number of maximal edge sequences on a convex polytope*, in Proc. 26th Allerton Conf. Commun. Control Comput., Univ. Illinois at Urbana-Champaign, IL, 1988, pp. 49–57.

[SO89] C. Schevon and J. O'Rourke, *An Algorithm for Finding Edge Sequences on a Polytope*, Technical Report JHU-89/03, Dept. Comput. Sci., Johns Hopkins Univ., Baltimore, MD, 1989.

[SS86] M. Sharir and A. Schorr, *On shortest paths in polyhedral spaces*, SIAM J. Comput., 15 (1986), pp. 193–215.

[ST86] N. Sarnak and R. E. Tarjan, *Planar point location using persistent search trees*, Commun. Assoc. Comput. Mach., 29 (1986), pp. 609-679.

[SA95] M. Sharir and P. K. Agarwal, *Davenport–Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, New York, 1995.

[Wel85] E. Welzl, *Constructing the visibility graph for n line segments in $O(n^2)$ time*, Inform. Process. Lett., 20 (1985), pp. 167–171.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1714–1732, December 1997

008

COMPUTING ENVELOPES IN FOUR DIMENSIONS WITH APPLICATIONS*

PANKAJ K. AGARWAL†, BORIS ARONOV‡, AND MICHA SHARIR§

Abstract. Let $\mathcal{F}$ be a collection of n d-variate, possibly partially defined, functions, all algebraic of some constant maximum degree. We present a randomized algorithm that computes the vertices, edges, and 2-faces of the lower envelope (i.e., pointwise minimum) of $\mathcal{F}$ in expected time $O(n^{d+\varepsilon})$ for any $\varepsilon > 0$. For $d = 3$, by combining this algorithm with the point-location technique of Preparata and Tamassia, we can compute, in randomized expected time $O(n^{3+\varepsilon})$, for any $\varepsilon > 0$, a data structure of size $O(n^{3+\varepsilon})$ that, for any query point q, can determine in $O(\log^2 n)$ time the function(s) of $\mathcal{F}$ that attain the lower envelope at q. As a consequence, we obtain improved algorithmic solutions to several problems in computational geometry, including (a) computing the width of a point set in 3-space, (b) computing the "biggest stick" in a simple polygon in the plane, and (c) computing the smallest-width annulus covering a planar point set. The solutions to these problems run in randomized expected time $O(n^{17/11+\varepsilon})$, for any $\varepsilon > 0$, improving previous solutions that run in time $O(n^{8/5+\varepsilon})$. We also present data structures for (i) performing nearest-neighbor and related queries for fairly general collections of objects in 3-space and for collections of moving objects in the plane and (ii) performing ray-shooting and related queries among n spheres or more general objects in 3-space. Both of these data structures require $O(n^{3+\varepsilon})$ storage and preprocessing time, for any $\varepsilon > 0$, and support polylogarithmic-time queries. These structures improve previous solutions to these problems.

Key words. lower envelopes, point location, ray shooting, closest pair

AMS subject classifications. 68Q20, 68Q25, 68R05, 68U05

PII. S0097539794265724

1. Introduction.

Let $\mathcal{F} = \{f_1, \ldots, f_n\}$ be a collection of n d-variate, possibly partially defined, functions, all algebraic of some constant maximum degree b (and if they are partially defined, their domains of definition are also described each by a constant number of polynomial equalities and inequalities of maximum degree b). Abusing the notation slightly, we will not distinguish between a function and its graph. The *lower envelope* $E_{\mathcal{F}}$ of $\mathcal{F}$ is defined as

$$E_{\mathcal{F}}(\mathbf{x}) = \min_i f_i(\mathbf{x}),$$

where the minimum is taken over all functions of $\mathcal{F}$ that are defined at $\mathbf{x}$. The *minimization diagram* $M_{\mathcal{F}}$ of $\mathcal{F}$ is the decomposition of $\mathbb{R}^d$ into maximal connected regions (of any dimension), called *cells* (or *faces*), so that within each cell the same subset of functions appears on the envelope $E_{\mathcal{F}}$. There is a natural subdivision of

* Received by the editors April 6, 1994; accepted for publication (in revised form) November 27, 1995. Work on this paper by the first author was supported by NSF grant CCR-93–01259 and by an NYI award. Work on this paper by the second author was supported by NSF grant CCR-92-11541. Work by the third author was supported by NSF grant CCR-91-22103, by a Max Planck Research Award, and by grants from the U.S.–Israeli Binational Science Foundation, the German–Israeli Foundation for Scientific Research and Development, and the Fund for Basic Research administered by the Israeli Academy of Sciences.

http://www.siam.org/journals/sicomp/26-6/26572.html

† Department of Computer Science, Box 90129, Duke University, Durham, NC 27708-0129 (pankaj@euclid.cs.duke.edu).

‡ Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201-3840 (aronov@ziggy.poly.edu).

§ School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel (sharir@math.tau.ac.il) and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012.

$E_{\mathcal{F}}$ into cells, as well, where a cell of the lower envelope is defined as the portion $E_{\mathcal{F}}$ that projects onto cell of $M_{\mathcal{F}}$. As the correspondence is 1-1, we will abuse the terminology and not make a distinction between the two kinds of cells. (A more detailed definition, treating also the case of partially defined functions, is given in [Sha].) The *combinatorial complexity* of $M_{\mathcal{F}}$ and of $E_{\mathcal{F}}$ is the number of faces of all dimensions in $M_{\mathcal{F}}$ and $E_{\mathcal{F}}$.

Recently, there has been a significant progress in the analysis of the combinatorial complexity of lower envelopes of multivariate functions [HS, Sha]. In particular, it was shown in [Sha] that the maximum complexity of $M_{\mathcal{F}}$ is $O(n^{d+\varepsilon})$, for any $\varepsilon > 0$, where the constant of proportionality depends on ε, d, and b. This result almost settles a major open problem and has already led to many applications [ASa, HS, Sha]. However, less progress has been made on the corresponding algorithmic problem, which calls for the efficient construction of the lower envelope of such a collection $\mathcal{F}$, in time $O(n^{d+\varepsilon})$, for any $\varepsilon > 0$. The ideal output of an algorithm that computes the envelope is a data structure of size $O(n^{d+\varepsilon})$, which can return, for a given point $\mathbf{x} \in \mathbb{R}^d$, the identity of the function(s) attaining the envelope at $\mathbf{x}$ in logarithmic (or polylogarithmic) time. Weaker solutions might provide just an enumeration of the cells of $M_{\mathcal{F}}$ and adjacency structure representing all pairs of cells that touch each other.

Sharir [Sha] presented an algorithm for computing the lower envelope of bivariate functions having the properties listed above, which runs in time $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$. Other algorithms with a similar performance are given in [BD, dBDS]. The simplest solution to this problem, involving a deterministic divide-and-conquer algorithm, was recently presented in [ASS]. All these algorithms facilitate point-location queries of the sort described above. Unfortunately, these methods fail in higher dimensions. For example, the technique of [Sha] relies on the existence of a *vertical decomposition* of the minimization diagram M_R of a sample R of r functions of $\mathcal{F}$, into a small number (that is, $O(r^{d+\varepsilon})$) of cells of *constant description complexity*. Such decompositions exist (and are easy to compute) for $d = 2$, but their existence in higher dimensions is still an open problem.

In this paper we present a randomized algorithm with $O(n^{d+\varepsilon})$ expected time, for any $\varepsilon > 0$, for computing the vertices, edges, and 2-dimensional faces of the lower envelope of n d-variate functions having the properties assumed above. We can use this algorithm to compute the entire lower envelope of a collection $\mathcal{F}$ of n trivariate functions, which has all the desired characteristics; in particular, it preprocesses $\mathcal{F}$, in expected time $O(n^{3+\varepsilon})$, into a data structure of size $O(n^{3+\varepsilon})$ that, for a query point q, can compute $E_{\mathcal{F}}(q)$, and also the function(s) attaining the envelope at q, in $O(\log^2 n)$ time. The algorithm bypasses the problem of having to construct small-size vertical decomposition by applying the technique of Preparata and Tamassia [PT] for point location in certain types of 3-dimensional subdivisions. This allows us to use a coarser decomposition of the minimization diagram, whose size is close to cubic.

Several recent papers [AST, CEGSb, MS] have studied a variety of geometric problems whose solution calls for the construction of, and searching in, lower or upper envelopes in 4-space. These applications fall into two main categories: *preprocess-for-queries* problems, which call for the construction of such an envelope, to be queried repeatedly later; and *batched* problems, where all the queries are known in advance and the goal is to produce the answers to all of them efficiently. We will present some of these applications, as listed in the abstract, where our new algorithm for computing lower envelopes in four dimensions leads to improved solutions.

The paper is organized as follows. In section 2 we present a general efficient randomized technique, which we believe to be of independent interest, for computing all the 0, 1, and 2-dimensional features of lower envelopes in any dimension. Then, in section 3, we apply this algorithm to the case of trivariate functions. This gives us an initial representation of the lower envelope of such a collection, which we then augment by additional features, resulting in a representation suitable for the application of the Preparata–Tamassia technique. In sections 4 and 5 we present applications of our result to a variety of problems in computational geometry, as listed in the abstract.

After the original submission of this paper, Agarwal and Sharir [ASb] observed that, for some of the problems that are solved here, the size of the vertical decomposition of the relevant minimization diagrams is small. This leads to slightly improved solutions to these problems. See also the remark at the end of section 5.1.

2. Lower envelopes in arbitrary dimension. In this section, we present a randomized technique for computing a partial description of lower envelopes of d-variate functions, for $d \geq 2$, whose expected running time is $O(n^{d+\varepsilon})$, for any $\varepsilon > 0$. This technique only computes the vertices, edges, and 2-faces of the envelope, which is sufficient for the full construction of envelopes in 4-space, as will be explained in section 3.

Let $\mathcal{F} = \{f_1, \ldots, f_n\}$ be a collection of (partial) d-variate functions in $\mathbb{R}^{d+1}$, satisfying the conditions described in the introduction. We assume that the functions of $\mathcal{F}$ are in general position; see [Sha] for more details and for a discussion of this assumption. To compute the lower envelope $E_{\mathcal{F}}$ of $\mathcal{F}$, in the above partial sense, we proceed as follows. Let Σ be the family of all $(d-1)$-subsets of $\mathcal{F}$. We fix a subset $\sigma = \{f_1, \ldots, f_{d-1}\} \in \Sigma$, and let

$$\Pi^\sigma = \{\mathbf{x} \in \mathbb{R}^d \mid f_1(\mathbf{x}) = \cdots = f_{d-1}(\mathbf{x})\}.$$

Since the f_i's are assumed to be in general position, Π^σ is a 2-dimensional surface (or a surface patch). For the sake of simplicity, we assume that Π^σ is connected and x_1x_2-monotone (i.e., any $(d-2)$-flat orthogonal to the x_1x_2-plane meets Π^σ in at most one point); otherwise, we decompose it into a constant number of connected portions so that each portion is an x_1x_2-monotone surface patch, and work with each patch separately. For each $i \geq d$, let

$$\gamma_i = \{\mathbf{x} \in \Pi^\sigma \mid f_i(\mathbf{x}) = f_1(\mathbf{x}) = \cdots = f_{d-1}(\mathbf{x})\};$$

γ_i is a 1-dimensional curve, which partitions Π^σ into two (not necessarily connected) regions, K_i^+ and K_i^-, where

$$K_i^+ = \{\mathbf{x} \in \Pi^\sigma \mid f_i(\mathbf{x}) > f_1(\mathbf{x}) = \cdots = f_{d-1}(\mathbf{x})\},$$
$$K_i^- = \{\mathbf{x} \in \Pi^\sigma \mid f_i(\mathbf{x}) < f_1(\mathbf{x}) = \cdots = f_{d-1}(\mathbf{x})\}.$$

Then the intersection $Q^\sigma = \bigcap_{i \geq d} K_i^+$ is the portion of Π^σ over which the envelope $E_{\mathcal{F}}$ is attained by the functions of σ. The algorithm will compute the regions Q^σ, over all choices of $(d-1)$-tuples σ of functions, thereby yielding the vertices, edges, and 2-faces of $M_{\mathcal{F}}$ (because of the general position assumption, any such feature must show up as a feature of at least one of the regions Q^σ). The algorithm actually computes the vertices, edges, and 2-faces of a refinement of $M_{\mathcal{F}}$, but the faces of $M_{\mathcal{F}}$ of dimension at most 2 can be retrieved from them in a straightforward manner. We omit the easy details.

We compute Q^σ using a randomized incremental approach, similar to the ones described in [CEGSS, dBDS, Mua, Mub, SA]. Since the basic idea is by now fairly standard, we give only a brief overview of the algorithm and refer the reader to [CEGSS, SA] for details. We first compute the set $\Gamma^\sigma = \{\gamma_i \mid d \le i \le n\}$.[1] Next, we add the curves γ_i one by one in a random order and maintain the intersection of the regions K_i^+ for the curves added so far. Let $(\gamma_d, \gamma_{d+1}, \ldots, \gamma_n)$ be the (random) insertion sequence, and let Q_i^σ denote the intersection of $K_d^+, \ldots, K_i^+$. We construct and maintain the "vertical decomposition" $\tilde{Q}_i^\sigma$ of Q_i^σ. This is defined as the partitioning of each 2-face ϕ of Q_i^σ into "pseudotrapezoids," obtained by drawing, from each vertex of ϕ and from each locally x_1-extreme point on $\partial\phi$, a curve within ϕ obtained by intersecting ϕ with the hyperplane $x_1 = const$, and by extending it (in both directions if necessary) until it meets $\partial\phi$ again. Each pseudotrapezoid is *defined* by at most four curves of Γ^σ; conversely, any four or fewer curves of Γ^σ define a constant number of pseudotrapezoidal cells, namely, those formed along Π^σ when only these curves are inserted. (Note that this construction is well defined since Π^σ is an x_1x_2-monotone surface.) In the $(i+1)$st step we add K_{i+1}^+ and compute $\tilde{Q}_{i+1}^\sigma$ from $\tilde{Q}_i^\sigma$, using the technique described in [CEGSS].

The analysis of the expected running time of the algorithm proceeds along the same lines as described in [CEGSS, SA]. We define the *weight*, $w(\tau)$, of a pseudotrapezoid τ, defined by the arcs of Γ^σ, to be the number of functions f_i, for $i = d, d+1, \ldots, n$, excluding the up to four functions whose intersections with Π^σ *define* τ, such that $f_i(\mathbf{x}) < f_1(\mathbf{x}) = \cdots = f_{d-1}(\mathbf{x})$ for some point $\mathbf{x} \in \tau$.

As shown in [CEGSS], the cost of the above procedure is proportional to the number of pseudotrapezoids that are created during the execution of the algorithm, plus the sum of their weights, plus an overhead term of $O(n^d)$ needed to prepare the collections of curves γ_i over all 2-dimensional intersection manifolds Π^σ. The analysis given below deals only with pseudotrapezoids that are defined by exactly four such functions (plus the $d-1$ functions defining Π^σ), and easy and obvious modifications are necessary to handle all other pseudotrapezoids.

Let T^σ denote the set of pseudotrapezoids (or "cells" for brevity) defined by four arcs of Γ^σ, and let $T = \bigcup_{\sigma\in\Sigma} T^\sigma$. Each cell in T is defined by $d-1+4 = d+3$ functions of $\mathcal{F}$. In what follows we implicitly assume that the specification of a cell $\tau \in T$ includes the $d+3$ functions defining τ, where there is a clear distinction between the first $d-1$ functions (constituting the set σ) and the last four functions (defining the four curves that generate τ along Π^σ). For an integer $k \ge 0$ and a subset $R \subseteq \mathcal{F}$, let $T_k(R) \subseteq T$ (resp., $T_{\le k}(R) \subseteq T$) denote the set of cells, defined by $d+3$ functions of R, as above, with weight k (resp., at most k). Let $N_k(R) = |T_k(R)|$ and

$$N_k(r) = \max_R N_k(R)\,,$$

where the maximum is taken over all subsets R of $\mathcal{F}$ of size r. Similarly, we define $N_{\le k}(R)$ and $N_{\le k}(r)$. Since each cell of $T_0(R)$ lies on the lower envelope of R, it follows that $N_0(r) = O(r^{d+\varepsilon})$. Adapting the analysis technique of Clarkson and Shor [ClS], we have the following lemma.

[1] We need to assume an appropriate model of computation, in which any of the various primitive operations required by the algorithm can be performed in constant time. For example, we can assume the model used in real computational algebraic geometry [HRR], where each algebraic operation involving a constant number of polynomials of constant maximum degree can be performed exactly, using rational arithmetic in constant time.

LEMMA 2.1. *The probability that a pseudotrapezoidal cell $\tau \in T_k(\mathcal{F})$ is created during the incremental construction of Q^σ, where $\sigma \in \Sigma$ is the tuple for which Π^σ contains τ, is $1/\binom{k+4}{4}$.*

Proof. For τ to be created, it is necessary and sufficient that the four curves of Γ^σ defining τ appear in the random insertion order before any of the curves corresponding to the k functions that contribute to the weight of τ, and this probability is easily seen to be $1/\binom{k+4}{4}$. □

LEMMA 2.2. *For any $0 \le k \le n-d-3$ and for any $\varepsilon > 0$,*

$$N_{\le k}(n) = O((k+1)^{3-\varepsilon} n^{d+\varepsilon}).$$

Proof. We use a variant of the probabilistic analysis technique of Clarkson and Shor [ClS]. If we choose a random sample $R \subseteq \Gamma$ of $r = \lfloor n/(k+1) \rfloor$ functions of $\mathcal{F}$, then a cell $\tau \in T_k(\Gamma)$ is in $T_0(R)$ if all $d+3$ functions, $f_1, \ldots, f_{d+3}$, defining τ are chosen in R, and none of the remaining k functions f_i, such that $f_i(\mathbf{x}) \le f_1(\mathbf{x})$ for some $\mathbf{x} \in \tau$, are chosen in R. The Clarkson–Shor technique implies that

$$\begin{aligned} N_{\le k}(n) &= O((k+1)^{d+3} N_0(\lfloor n/(k+1) \rfloor)) \\ &= O((k+1)^{d+3} (n/(k+1))^{d+\varepsilon}) \\ &= O((k+1)^{3-\varepsilon} n^{d+\varepsilon}), \end{aligned}$$

for any $\varepsilon > 0$. □

For a cell $\tau \in T$, let A_τ be the event that $\tau \in \tilde{Q}_i^\sigma$, for the tuple $\sigma \in \Sigma$ for which T^σ contains τ, and for some $d \le i \le n$. The expected running time of the algorithm, over all choices of $(d-1)$-tuples of functions, is thus proportional to

$$\begin{aligned} \sum_{\tau \in T} \Big[(w(\tau)+1) \cdot \Pr[A_\tau] \Big] + O(n^d) &= \sum_{k \ge 0} \sum_{\tau \in T_k(\mathcal{F})} \Big[(k+1) \cdot \Pr[A_\tau] \Big] + O(n^d) \\ &= \sum_{k=0}^{n-d-3} \frac{(k+1) N_k(\mathcal{F})}{\binom{k+4}{4}} + O(n^d), \end{aligned}$$

where the last inequality follows from Lemma 2.1. Since

$$N_k(\mathcal{F}) = N_{\le k}(\mathcal{F}) - N_{\le (k-1)}(\mathcal{F}),$$

we obtain, using Lemma 2.2,

$$\begin{aligned} &\sum_{k=0}^{n-d-3} \frac{(k+1) N_k(\mathcal{F})}{\binom{k+4}{4}} \\ &= N_0(\mathcal{F}) + 24 \sum_{k=1}^{n-d-3} \frac{N_{\le k}(\mathcal{F}) - N_{\le (k-1)}(\mathcal{F})}{(k+4)(k+3)(k+2)} \\ &= O(n^{d+\varepsilon}) + 24 \sum_{k=1}^{n-d-4} N_{\le k}(n) \left[\frac{1}{(k+4)(k+3)(k+2)} - \frac{1}{(k+5)(k+4)(k+3)} \right] \\ &\qquad + \frac{24 N_{\le n-d-3}(n)}{(n-d+1)(n-d)(n-d-1)} \\ &= O\left(n^{d+\varepsilon} + \sum_{k=1}^{n-d-4} \frac{k^{3-\varepsilon} n^{d+\varepsilon}}{(k+5)(k+4)(k+3)(k+2)} \right) \qquad \text{(using Lemma 2.2)} \end{aligned}$$

$$= O\left(n^{d+\varepsilon} \cdot \sum_{k=1}^{n-d-4} \frac{1}{k^{1+\varepsilon}}\right) = O(n^{d+\varepsilon}) .$$

We thus obtain the following result.

THEOREM 2.3. *The vertices, edges, and 2-faces of the lower envelope of n (partial) d-variate functions, satisfying the conditions stated in the introduction, can be computed in randomized expected time $O(n^{d+\varepsilon})$, for any $\varepsilon > 0$.*

Remark. The preceding analysis shows that, for $d \geq 2$, the expected running time of our algorithm is in fact $O(n^d + \sum_{k=1}^{n} k^{d-1}\varphi(\lfloor n/k \rfloor))$, where $\varphi(r)$ is an upper bound on the complexity of the lower envelope of any subset of S of size at most r. For example, the vertices, edges, and 2-faces of the lower envelope of n simplices in $\mathbb{R}^{d+1}$ can be computed in $O(n^d \alpha(n) \log n)$ expected time, because the complexity of the lower envelope of n d-simplices is $O(n^d \alpha(n))$ [PS].

3. Envelopes in four dimensions. We next apply the results of the preceding section to obtain an efficient algorithm for constructing the lower envelope of a collection $\mathcal{F}$ of n trivariate functions, satisfying the assumptions made above in the following strong sense: One can preprocess $\mathcal{F}$ in randomized expected time $O(n^{3+\varepsilon})$, for any $\varepsilon > 0$, into a data structure of size $O(n^{3+\varepsilon})$ that supports queries of the form: given a point $w \in \mathbb{R}^3$, compute the function(s) attaining $E_{\mathcal{F}}$ at w; each query can be answered in $O(\log^2 n)$ time.

To achieve this we first apply the algorithm summarized in Theorem 2.3 to compute the vertices, edges, and 2-faces of the minimization diagram $M_{\mathcal{F}}$ of $\mathcal{F}$. We next partition each cell of $M_{\mathcal{F}}$ into "monotone" subcells, in the sense of Lemma 3.1, to obtain a refinement $M'_{\mathcal{F}}$ of $M_{\mathcal{F}}$. This refinement satisfies the requirements of the point-location method due to Preparata and Tamassia [PT], which we use to produce the data structure representing the lower envelope in the above manner.

We define and construct $M'_{\mathcal{F}}$ as follows. We mark, along each 2-face F of $M_{\mathcal{F}}$, the locus γ_F of all points of F which are either singular or have a vertical tangency (in the z-direction). The arcs γ_F, for all 2-faces F, lie along $O(n^2)$ curves in $\mathbb{R}^3$, each being the xyz-projection of (i) the locus of all singular points or points with z-vertical tangency along some 2-manifold $f_i = f_j$, for a pair of indices $i \neq j$, or (ii) of points along the boundary of some f_i. We consider below only the former case; the latter case can be handled in almost the same (and, actually, simpler) manner. Let δ be one of these curves. We consider the 2-dimensional surface V_δ, within the xyz-space, obtained as the union of all lines passing through points of δ and parallel to the z-axis; let V_δ^+, V_δ^- denote the portions of V_δ that lie, respectively, above and below δ. (Since δ is not necessarily an xy-monotone arc, V_δ may have self-intersections along some vertical lines, along which V_δ^+ and V_δ^- are not well defined; we omit here details of the (rather easy) handling of such cases.)

Let δ_0 be the portion of δ over which the functions f_i and f_j attain the envelope $E_{\mathcal{F}}$. Clearly, δ_0 is the union of all arcs γ_F that are contained in δ, and the number of connected components in δ_0, summed over all intersection curves δ, is $O(n^{3+\varepsilon})$. Let w be a point in δ_0. Then the cell c of $M_{\mathcal{F}}$ lying immediately above w in the z-direction is such that within c the envelope $E_{\mathcal{F}}$ is attained by either f_i or f_j. Thus the upward-directed z-vertical ray emanating from w leaves c (if at all) at a point above w that lies on the xyz-projection of a 2-manifold of the form $f_i = f_k$ or $f_j = f_k$. (In addition, it is also possible that the ray will encounter a point on the projection of the boundary of the domain of f_i or f_j—this can be handled by similar, but easier, means.) For fixed i and j, there are only $O(n)$ possible 2-manifolds of this kind (i.e., xyz-projections

of surfaces of the form $f_i = f_k$ or $f_j = f_k$). We compute the lower envelope $E^{(\delta)}$ of these $O(n)$ 2-manifolds, restricted to V_δ^+. It is easily seen that the complexity of $E^{(\delta)}$ is $O(\lambda_q(n))$, for some constant q depending on the maximum degree of these 2-manifolds; $\lambda_q(n)$ is the maximum length of Davenport–Schinzel sequences of order q that are composed of n symbols and is close to linear in n for any fixed q [ASS, SA]. We next take the portions of the graph of $E^{(\delta)}$ that lie over δ_0 and "etch" them along the corresponding 2-faces of $M_\mathcal{F}$. We apply a fully symmetric procedure within V_δ^- and repeat these steps for all curves δ. The overall combinatorial complexity of all the added curves is thus $O(n^2\lambda_q(n) + n^{3+\varepsilon}) = O(n^{3+\varepsilon})$, for any $\varepsilon > 0$, and they can be computed in $O(n^{3+\varepsilon})$ time, as is easily verified.

Let $M'_\mathcal{F}$ denote the refined cell decomposition of $\mathbb{R}^3$, obtained by adding to $M_\mathcal{F}$, for each of the curves δ, the arcs γ_F, the etched arcs of the upper and lower envelopes in the vertical manifolds V_δ, and the z-vertical walls (i.e., union of all z-vertical segments) contained in V_δ and connecting the arcs γ_F to the etched arcs. If an edge of $M'_\mathcal{F}$ is not monotone in the y-direction, we split it into $O(1)$ edges by adding a vertex at every local y-extremal point on this edge. As just argued, the combinatorial complexity of $M'_\mathcal{F}$ is still $O(n^{3+\varepsilon})$, and $M'_\mathcal{F}$ has the following crucial property.

LEMMA 3.1. *For each 3-cell c of $M'_\mathcal{F}$, every connected component of a cross section of c by a plane parallel to the xz-plane is x-monotone.*

Proof. Suppose the contrary, and let π be a plane parallel to the xz-plane, for which there exists a connected component c' of $c \cap \pi$ that is not x-monotone. Then there is a point $w \in \partial c'$ so that the z-vertical line passing through w meets c' both slightly above and slightly below w. But then w is either a singular point or a point with z-vertical tangency lying on one of the curves γ_F. By construction, $M'_\mathcal{F}$ must contain the vertical segment passing through w and contained in c, as part of some vertical wall, a contradiction. This completes the proof of the lemma. □

Let $M'_\mathcal{F}(y_0)$ denote the cross section of $M'_\mathcal{F}$ by the plane $y = y_0$. Lemma 3.1 implies that $M'_\mathcal{F}(y_0)$ is an x-monotone planar subdivision, for each y_0. Hence, if we orient the edges of $M'_\mathcal{F}$ in the positive x-direction, add a pair of nominal points s, t at $x = -\infty$ and $x = +\infty$, respectively, and connect them to the appropriate unbounded edges of $M'_\mathcal{F}(y_0)$, this map becomes a planar st-graph, in the notation of Preparata and Tamassia [PT]. (Note that the vertical decomposition that generates $M'_\mathcal{F}$ from $M_\mathcal{F}$ may create z-vertical edges; if these edges are oriented consistently, say in the upward z-direction, then $M'_\mathcal{F}$ remains an st-graph.) We denote this st-graph also by $M'_\mathcal{F}(y_0)$.

LEMMA 3.2. *Let I be an open interval of the y-axis that does not contain the y-coordinate of any vertex of $M'_\mathcal{F}$. Then, for each $y_1, y_2 \in I$, $M'_\mathcal{F}(y_1)$, $M'_\mathcal{F}(y_2)$ are isomorphic, as labeled embedded planar st-graphs.*

Proof. We call a y-coordinate *critical* if it is the y-coordinate of a vertex of $M'_\mathcal{F}$. Since each vertex of $M'_\mathcal{F}(\cdot)$ is an intersection of the sweep plane with an edge of $M'_\mathcal{F}$ and since each edge of $M'_\mathcal{F}$ is y-monotone, the set of vertices of $M'_\mathcal{F}(y)$ can change only at critical values of y. In other words, the set of vertices of the cross section $M'_\mathcal{F}(\cdot)$ does not change combinatorially between any two consecutive critical values. Let (u_1, v_1) be a nonvertical directed edge of $M'_\mathcal{F}(y_1)$, and let u_2, v_2 be the vertices of $M'_\mathcal{F}(y_2)$ corresponding, respectively, to u_1 and v_1 (i.e., the intersections of the sweep plane with the same edges of $M'_\mathcal{F}$). Suppose to the contrary that (u_2, v_2) is not an edge of $M'_\mathcal{F}(y_2)$. There are several ways in which the edge (u_1, v_1) can disappear during the sweep:

(i) it might cease to be x-monotone (with a vertical inflection point, or another

singular point, appearing in its middle),
(ii) some vertical edge might appear and split it in two,
(iii) its two vertices might approach each other and merge into a common vertex, or
(iv) either of the vertices might split into two vertices.

However, as is easily verified, the y-coordinate of any of these events must be one of the critical values of y, contrary to assumption that the interval I does not contain the y-coordinate of any vertex of $M'_{\mathcal{F}}$. A similar argument handles the case where (u_1, v_1) is a vertical edge of $M'_{\mathcal{F}}(y_1)$: such an edge is erected if one of these vertices, say u_1, has a z-vertical tangency, or is otherwise singular along one of the surfaces of $M_{\mathcal{F}}$. As we sweep from y_1 to y_2, this vertical edge might disappear if any of the following events occurs:

(i) the point corresponding to u_1 changes its singular status (for simplicity of exposition, the term "singular" refers here both to singular points and to points with z-vertical tangency);
(ii) the point corresponding to u_1 splits into two vertices or merges with another vertex of the cross section;
(iii) the point corresponding to v_1 splits into two vertices or merges with another vertex;
(iv) the point corresponding to v_1 becomes singular on its surface; or
(v) another singular point appears on the vertical segment u_1v_1.

Again, for any of these events, its y-coordinate must be critical, contrary to assumption. This completes the proof of the lemma. □

We are now in a position to apply the point-location technique of Preparata and Tamassia [PT]. They show that a 3-dimensional subdivision of combinatorial complexity N can be preprocessed in time $O(N \log^2 N)$ into a data structure of size $O(N \log^2 N)$, which supports $O(\log^2 N)$-time point-location queries in the given subdivision. However, for this to hold, the subdivision must have the following two properties:

(a) the 1-skeleton of each cross section of the subdivision by a plane parallel to the xz-plane is a planar st-graph, whose faces are all monotone in the x-direction and whose edges are all oriented in the positive x-direction (or, for vertical edges, in the positive z-direction).
(b) At each point where the cross section, as an st-graph, changes, the graph can be updated by a constant number of operations from the following collection:
  (b.1) insertion of a vertex in the middle of an edge, or the complementary operation of deleting a vertex of in-degree and out-degree 1 and replacing the two incident edges by one, while maintaining x-monotonicity of the incident faces;
  (b.2) insertion of an edge partitioning an x-monotone face into two x-monotone subfaces, or, conversely, deletion of an edge and merging its two adjacent faces, provided their union is x-monotone;
  (b.3) merging two adjacent vertices into one vertex (collapsing the edge connecting them) or splitting a vertex into two vertices (forming a new edge between these vertices), again maintaining x-monotonicity.

It is easily verified that, indeed, each change occurring in the structure of the cross section $M'_{\mathcal{F}}(\cdot)$, at any of the critical y values in Lemma 3.2, can be expressed as a constant number of operations of the types mentioned above. In summary, we thus obtain the following main result of the paper.

THEOREM 3.3. *Let $\mathcal{F}$ be a given collection of n trivariate, possibly partially defined, functions, all algebraic of constant maximum degree, and whose domains of definition (if they are partially defined) are each defined by a constant number of algebraic equalities and inequalities of constant maximum degree. Then, for any $\varepsilon > 0$, the lower envelope $E_{\mathcal{F}}$ of $\mathcal{F}$ can be computed in randomized expected time $O(n^{3+\varepsilon})$, and stored in a data structure of size $O(n^{3+\varepsilon})$, so that, given any query point $w \in \mathbb{R}^3$, we can compute $E_{\mathcal{F}}(w)$, as well as the function(s) attaining $E_{\mathcal{F}}$ at w, in $O(\log^2 n)$ time.*

4. Applications: Query problems. In this section we apply Theorem 3.3 to two problems involving preprocessing and querying certain collections of objects in 3-space.

4.1. Ray shooting amidst spheres. Given a collection $\mathcal{S}$ of n spheres in 3-space, we wish to preprocess $\mathcal{S}$ into a data structure that supports *ray-shooting* queries, each seeking the first sphere, if any, met by a query ray. This problem has recently been studied in [AMb, AGPS, MS]. The first two papers present a data structure that requires $O(n^{4+\varepsilon})$ storage and preprocessing, for any $\varepsilon > 0$, and answers a query in time $O(\log^2 n)$. The third paper [MS] gives a rather elaborate and improved solution that requires $O(n^{3+\varepsilon})$ preprocessing and storage, for any $\varepsilon > 0$, and answers a query in $O(n^{\varepsilon})$ time.

These algorithms use the technique described in [AMa] and reduce the problem to that of determining whether a query segment e intersects any sphere of $\mathcal{S}$. Then, using a multilevel data structure, they reduce the problem to that of detecting whether the line λ containing the segment e intersects any sphere of $\mathcal{S}$. As observed in [AGPS, MS], this problem can be further reduced to point location in the upper envelope of a set of certain trivariate functions, as follows. Let π be the plane passing through λ and orthogonal to the vertical plane V passing through λ. Let π^+ (resp., π^-) be the half-space lying above (resp., below) π. Let S be a sphere whose center lies in π^+ and that intersects V in a disc D. Then the center of D lies above λ, so either λ intersects S or passes below S, in the sense that λ and S are disjoint and there is a point on λ that lies vertically below a point in S (see Figure 1). A similar property holds if S intersects V and its center lies in π^-. We preprocess the centers of spheres of $\mathcal{S}$ into a half-space range-searching data structure of size $O(n^{3+\varepsilon})$. Then, for a query λ, we can decompose $\mathcal{S}$ into $O(1)$ canonical subsets, so that, within each subset, either the centers of all spheres lie in π^+ or they all lie in π^-. Let us consider the case when the centers of all spheres lie in π^+.

Hence, we need to solve the following subproblem: Given a set $\mathcal{S}$ of n spheres in 3-space, and given a query line λ with the property that for each sphere $S \in \mathcal{S}$, either λ intersects S or there is no point of S lying vertically below λ, determine whether λ intersects any sphere of $\mathcal{S}$. We reduce this problem to point location in the lower envelope of certain trivariate functions as follows. We can parametrize a line λ in 3-space by four parameters $(\xi_1, \xi_2, \xi_3, \xi_4)$, so that the equations defining λ are $y = x\xi_1 + \xi_2$, $z = x\xi_3 + \xi_4$. (We assume here that λ is not parallel to the yz-plane; such lines can be handled in a different, and much simpler manner.) For each sphere $S \in \mathcal{S}$, define a function $\xi_4 = F_S(\xi_1, \xi_2, \xi_3)$, so that the line $\lambda(\xi_1, \xi_2, \xi_3, \xi_4)$ is tangent to S from below (F_S is only partially defined, and we put $F_S = +\infty$ when it is undefined; it is easily checked that F_S is algebraic of bounded degree and that its boundary is also algebraic of bounded degree). Let Φ be the lower envelope of the functions F_S, for $S \in \mathcal{S}$. Then a query line $\lambda(\xi_1, \xi_2, \xi_3, \xi_4)$ having the above properties misses all spheres of $\mathcal{S}$ if and only if $\xi_4 < \Phi(\xi_1, \xi_2, \xi_3)$. Thus, by Theorem 3.3,

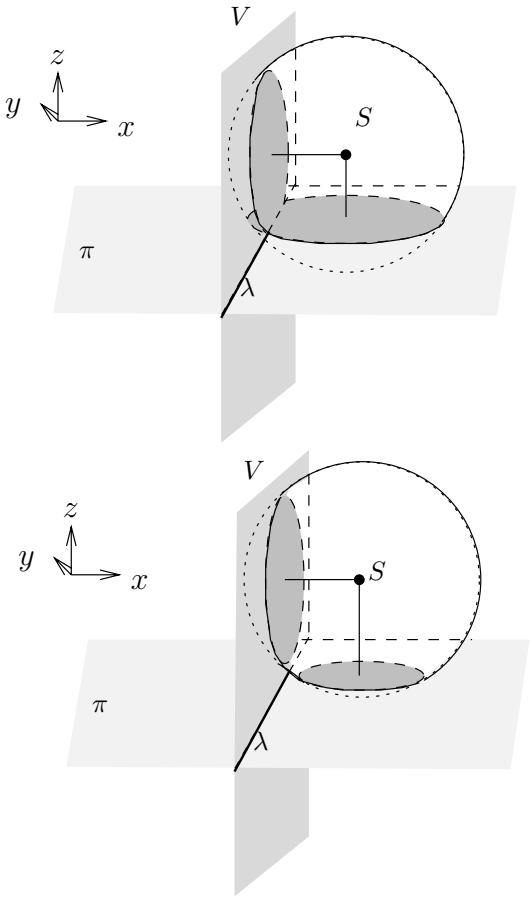


FIG. 1. *In the reduced subproblem, λ either intersects S or passes below it.*

one can answer such queries in time $O(\log^2 n)$, using $O(n^{3+\varepsilon})$ preprocessing time and storage, for any $\varepsilon > 0$. Plugging the half-space range-searching data structure and the point-location data structure into the multilevel data structure described in [AGPS, MS], we obtain a data structure for the segment-emptiness problem. A closer analysis of this data structure shows that the preprocessing time and storage of the overall data structure is still $O(n^{3+\varepsilon})$, for any $\varepsilon > 0$, and that the query time is $O(\log^2 n)$. Finally, plugging this data structure for segment-emptiness queries into the general ray-shooting technique of Agarwal and Matoušek [AMa], we obtain a final data structure, still requiring near-cubic storage and preprocessing, using which one can answer a ray-shooting query in time $O(\log^4 n)$. That is, we have shown the following.

THEOREM 4.1. *A set $\mathcal{S}$ of n spheres in $\mathbb{R}^3$ can be preprocessed in randomized expected time $O(n^{3+\varepsilon})$ into a data structure of size $O(n^{3+\varepsilon})$, for any $\varepsilon > 0$, so that a ray-shooting query can be answered in time $O(\log^4 n)$.*

4.2. Nearest-neighbor queries. Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a collection of n objects ("sites") in 3-space, each having *constant description complexity*, namely, each defined by a constant number of algebraic equalities and inequalities of constant max-

imum degree. We wish to compute the *Voronoi diagram* $Vor(\mathcal{S})$ of $\mathcal{S}$ and preprocess it for efficient point-location queries. That is, each query specifies a point w in 3-space and seeks the site of $\mathcal{S}$ nearest to w (say, under the Euclidean distance). This is a generalization to 3-space of the classical *post-office problem*.

As observed in [ES], the problem is equivalent to the computation of the following lower envelope in 4-space. For each $s \in \mathcal{S}$, define $F_s(x, y, z)$ to be the Euclidean distance from (x, y, z) to s, and let Φ be the lower envelope of these functions. Then, given a query point (x, y, z), the site(s) $s \in \mathcal{S}$ nearest to w are those for which F_s attains Φ at (x, y, z). Thus, by Theorem 3.3, S can be preprocessed in $O(n^{3+\varepsilon})$ time, for any $\varepsilon > 0$, into a data structure of size $O(n^{3+\varepsilon})$, so that each query can be answered in $O(\log^2 n)$ time. (Note that Theorem 3.3 is indeed applicable here, because the functions F_s are all (piecewise) algebraic of constant maximum degree, as is easy to verify from the conditions assumed above.) Hence, we can conclude the following.

THEOREM 4.2. *A set $\mathcal{S}$ of n objects in $\mathbb{R}^3$, as described above, can be preprocessed in randomized expected time $O(n^{3+\varepsilon})$ into a data structure of size $O(n^{3+\varepsilon})$, for any $\varepsilon > 0$, so that a nearest-neighbor query can be answered in time $O(\log^2 n)$.*

This is a fairly general framework and admits numerous generalizations, e.g., we may replace the Euclidean distance by other distances, perform queries with objects other than points (as long as the location of a query object can be specified by only three real parameters; this is the case, e.g., if the query objects are translates of some rigid convex object), etc. Of course, any such generalization requires that the metric or the query objects also have constant description complexity, in the above sense. An interesting generalization is to *dynamic* nearest-neighbor queries in the plane, where each object of $\mathcal{S}$ moves along some given trajectory and each query (x, y, t) asks for the object of $\mathcal{S}$ nearest to the point (x, y) at time t. Using the same approach as above, this can be done, under appropriate assumptions on the shape and motion of the objects of $\mathcal{S}$, with $O(n^{3+\varepsilon})$ preprocessing time and storage, for any $\varepsilon > 0$ and $O(\log^2 n)$ query time.

Remarks. (i) In Theorem 4.2 we do not assume that the objects are pairwise disjoint. In this case one cannot hope to do much better, because the complexity of the Voronoi diagram of a set of intersecting objects in $\mathbb{R}^3$ is easily seen to be cubic in the worst case. For example, consider the Voronoi diagram of a set of planes in $\mathbb{R}^3$.

(ii) There is a prevailing conjecture that the complexity of generalized Voronoi diagrams of a set of *pairwise disjoint convex* objects in $\mathbb{R}^3$, and of dynamic Voronoi diagrams in the plane, is only near-quadratic, under reasonable assumptions concerning the distance function and the shape of the sites (and of their motions in the dynamic sense). This was indeed proved recently in [CKSW] for the case where the sites are lines in $\mathbb{R}^3$ and the distance function is induced by a convex polyhedron, and in [BSTY] for point sets under the L_1 or L_∞ metric. If this conjecture is established, then the above algorithm, of near-cubic cost, is far from being optimal for pairwise disjoint objects and will need to be improved considerably.

5. Applications: Batched problems. We next consider *batched* applications. These applications solve a variety of rather unrelated problems, but they all involve an almost identical subproblem, in which lower (or upper) envelopes in 4-space play a role. To avoid repetition, we describe in detail only one application and then state the improved bounds for the other applications without proof, referring the reader to the relevant literature.

The following applications are based on the *parametric-search* technique of Megiddo

[Me], which, in our case, requires a parallel algorithm for computing the lower envelope of a set of surfaces. However, the algorithm presented above is highly unparallelizable, because it uses an incremental insertion procedure of regions into 2-dimensional arrangements and later uses a plane-sweep in 3-space. To finesse this difficulty, we apply the parametric-search technique with an additional twist that avoids the need for a parallel algorithm.

5.1. Width in 3-space. Let S be a set of n points in 3-space. The *width* of S is the smallest distance between a pair of parallel planes so that the closed slab between the planes contains S. In two dimensions, the problem can be easily solved in $O(n \log n)$ time. An $O(n^2)$-time algorithm for three dimensions was presented in [HT]. Recently, Chazelle et al. [CEGSb] presented an $O(n^{8/5+\varepsilon})$-time algorithm for any $\varepsilon > 0$. In this subsection, we present an improved randomized algorithm, whose expected running time is $O(n^{17/11+\varepsilon}) = O(n^{1.546})$.

Clearly, it suffices to compute the width of the convex hull of S, so assume that the points of S are in convex position and that the convex hull $\mathcal{P}$ of S is given. It is known that any two planes defining the width of S are such that either one touches a face of $\mathcal{P}$ and one touches a vertex of $\mathcal{P}$, or each of them touches an edge of $\mathcal{P}$; see [CEGSb, HT]. The first case can be handled in $O(n \log n)$ time [CEGSb, HT]. The difficult case is to compute the smallest distance between a pair of parallel planes supporting $\mathcal{P}$ at two "antipodal" edges because, in the worst case, there can be $\Theta(n^2)$ such pairs of edges. Chazelle et al. [CEGSb] presented an $O(n^{8/5+\varepsilon})$-time algorithm to find the smallest distance, using the parametric-search technique. We will present a randomized algorithm, whose expected running time is $O(n^{17/11+\varepsilon})$, for any $\varepsilon > 0$, using a somewhat different approach.

Let $\mathcal{M}$ denote the *Gaussian diagram* (or *normal diagram*) of $\mathcal{P}$. $\mathcal{M}$ is a spherical map on the unit sphere $\mathbb{S}^2 \subset \mathbb{R}^3$. The vertices of $\mathcal{M}$ are points on $\mathbb{S}^2$, each being the outward normal of a face of $\mathcal{P}$, the edges of $\mathcal{M}$ are great circular arcs, each being the locus of the outward normal directions of all planes supporting $\mathcal{P}$ at some fixed edge, and the faces of $\mathcal{M}$ are regions, each being the locus of the outward normal directions of all planes supporting $\mathcal{P}$ at a vertex. $\mathcal{M}$ can be computed in linear time from $\mathcal{P}$. Let $\mathcal{M}'$ denote the spherical map $\mathcal{M}$ reflected through the origin, and consider the superposition of $\mathcal{M}$ and $\mathcal{M}'$. It suffices to consider the top parts of $\mathcal{M}$ and $\mathcal{M}'$, i.e., their portions within the hemisphere $z \geq 0$. Each intersection point between an edge of $\mathcal{M}$ and an edge of $\mathcal{M}'$ gives us a direction $\mathbf{u}$ for which there exist two parallel planes orthogonal to $\mathbf{u}$ and supporting $\mathcal{P}$ at two so-called "antipodal" edges. Thus the problem reduces to that of finding an intersection point for which the distance between the corresponding parallel supporting planes is minimized.

We centrally project the edges of (the top portions of) $\mathcal{M}$ and $\mathcal{M}'$ onto the plane $z = 1$. Each edge projects to a line segment or a ray. Let $\mathcal{E}$ and $\mathcal{E}'$ be the resulting sets of segments and rays in the plane. Using the algorithm described in [CEGSa], we can decompose $\mathcal{E} \times \mathcal{E}'$, in time $O(n \log^2 n)$, into a family of "canonical subsets"

$$\mathcal{F} = \{(\mathcal{E}_1, \mathcal{E}'_1), (\mathcal{E}_2, \mathcal{E}'_2), \ldots, (\mathcal{E}_t, \mathcal{E}'_t)\}\,, \tag{1}$$

such that

(i) $\mathcal{E}_i \subseteq \mathcal{E}$ and $\mathcal{E}'_i \subseteq \mathcal{E}'$;
(ii) $\sum_{i=1}^{t} (|\mathcal{E}_i| + |\mathcal{E}'_i|) = O(n \log^2 n)$;
(iii) each segment in $\mathcal{E}_i$ intersects every segment of $\mathcal{E}'_i$; and
(iv) for every pair (e, e') of intersecting segments in $\mathcal{E} \times \mathcal{E}'$, there is an i such that $e \in \mathcal{E}_i$ and $e' \in \mathcal{E}'_i$.

By property (iv), it suffices to consider each pair $(\mathcal{E}_i, \mathcal{E}'_i)$ separately. Let $\mathcal{L}_i$ (resp., $\mathcal{L}'_i$) denote the set of lines containing the edges of $\mathcal{P}$ corresponding to the edges of $\mathcal{E}_i$ (resp., $\mathcal{E}'_i$). By construction, all lines of $\mathcal{L}_i$ lie above all lines of $\mathcal{L}'_i$. Property (iii) implies that every pair of parallel planes π, π', where π contains a line λ of $\mathcal{L}_i$ and π' contains a line λ' of $\mathcal{L}'_i$, are supporting planes of $\mathcal{P}$. Hence, we want to compute the smallest distance between two such planes. Since the distance between π and π', as above, is equal to the distance between λ and λ', it follows that this problem is equivalent to that of computing the closest pair of lines in $\mathcal{L}_i \times \mathcal{L}'_i$. Hence, we need to solve the following problem: Given a set $\mathcal{L}$ of m "red" lines and another set $\mathcal{L}'$ of n "blue" lines in $\mathbb{R}^3$, such that all red lines lie above all blue lines, compute the closest pair of lines in $\mathcal{L} \times \mathcal{L}'$. For any pair of lines $\ell, \ell' \in \mathbb{R}^3$, let $d(\ell, \ell')$ be the Euclidean distance between them, and let

$$d(\mathcal{L}, \mathcal{L}') = \min_{\ell \in \mathcal{L},\, \ell' \in \mathcal{L}'} d(\ell, \ell')\,.$$

We first describe a simple randomized divide-and-conquer algorithm for computing $\delta^* = d(\mathcal{L}, \mathcal{L}')$, whose expected running time is $O(n^{3+\varepsilon} + m \log^2 n)$. If $m = O(1)$, we compute $d(\ell, \ell')$ for all pairs $\ell \in \mathcal{L}$, $\ell' \in \mathcal{L}'$ and choose the minimum distance. Otherwise, the algorithm performs the following steps:

(i) Choose a random subset $R_1 \subseteq \mathcal{L}$ of $m/2$ red lines; each subset of size $m/2$ is chosen with equal probability.
(ii) Solve the problem recursively for $(R_1, \mathcal{L}')$. Let $\delta_1 = d(R_1, \mathcal{L}')$.
(iii) Compute the set $R_2 = \{\ell \in \mathcal{L} \setminus R_1 \mid d(\ell, \mathcal{L}') < \delta_1\}$.
(iv) Compute $d(\ell, \ell')$ for all pairs $\ell \in R_2$, $\ell' \in \mathcal{L}'$, and output the minimum distance (or output δ_1 if R_2 is empty).

For a line $\ell' \in \mathcal{L}'$, let $R^{(\ell')} = \{\ell \in \mathcal{L} \mid d(\ell, \ell') < \delta_1\}$, so that $R_2 = \bigcup_{\ell' \in \mathcal{L}'} R^{(\ell')}$. Using a standard probabilistic argument, we can show that the expected size of $R^{(\ell')}$ is $O(1)$, for each $\ell' \in \mathcal{L}'$. Therefore the expected size of R_2 is $O(n)$. Consequently, the expected running time of step (iv) is $O(n^2)$. The only nontrivial step in the above algorithm is step (iii). We compute R_2 as follows. We map each line $\ell \in \mathcal{L}$ to a point $\psi(\ell) = (a_1, a_2, a_3, a_4)$ in $\mathbb{R}^4$, where $y = a_1 x + a_2$ is the equation of the xy-projection of ℓ, and $z = a_3 u + a_4$ is the equation of ℓ in the vertical plane $y = a_1 x + a_2$ (here u denotes the axis orthogonal to the z-axis). We can also map a line ℓ' to a surface $\gamma = \gamma(\ell')$ in this parameter space, which is the locus of all points $\psi(\ell)$ such that $d(\ell, \ell') = \delta_1$ and ℓ lies above ℓ'. Our choice of parameters ensures that $\gamma(\ell')$ is $x_1x_2x_3$-monotone; i.e., any line parallel to the x_4-axis intersects $\gamma(\ell')$ in at most one point. For any point lying below $\gamma(\ell')$ the corresponding line ℓ either lies below ℓ' or lies above ℓ' and $d(\ell, \ell') < \delta_1$. For a point lying above $\gamma(\ell')$, the corresponding line ℓ lies above ℓ' and $d(\ell, \ell') > \delta_1$. In view of the above discussion, $\gamma(\ell')$ is the graph of a partial function $x_4 = f_{\ell'}(x_1, x_2, x_3)$. (The function $f_{\ell'}$ is undefined only at points (x_1, x_2, x_3) that represent lines whose xy-projection is parallel to that of ℓ'; the locus of these points is a plane.) Let

$$F(x_1, x_2, x_3) = \max_{\ell' \in \mathcal{L}'} f_{\ell'}(x_1, x_2, x_3)$$

be the upper envelope of the set $\{f_{\ell'} \mid \ell' \in \mathcal{L}'\}$. For a line $\ell \in \mathcal{L}$ with $\psi(\ell) = (a_1, a_2, a_3, a_4)$, we have $a_4 \geq F(a_1, a_2, a_3)$ if and only if $d(\{\ell\}, \mathcal{L}') \geq \delta_1$. The problem of computing R_2 thus reduces to that of computing the set of points $\psi(\ell)$, for $\ell \in \mathcal{L} \setminus R_1$, that lie below the upper envelope F. By Theorem 3.3, this can be accomplished in

time $O(n^{3+\varepsilon} + m\log^2 n)$, for any $\varepsilon > 0$. The total time spent in steps (iii) and (iv) is thus $O(n^{3+\varepsilon} + m\log^2 n)$. Let $T(m,n)$ denote the maximum expected running time of the algorithm. Then

$$T(m,n) = \begin{cases} O(n) & \text{if } m \le m_0\,, \\ T\left(\frac{m}{2}, n\right) + O(n^{3+\varepsilon} + m\log^2 n) & \text{if } m > m_0\,, \end{cases}$$

where m_0 is a constant. The solution to this recurrence is easily seen to be

$$T(m,n) = O(n^{3+\varepsilon}\log m + m\log^2 n).$$

Hence, we can conclude the following.

LEMMA 5.1. *Given a set $\mathcal{L}$ of m lines and another set $\mathcal{L}'$ of n lines in $\mathbb{R}^3$, such that all lines in $\mathcal{L}$ lie above all lines of $\mathcal{L}'$, the closest pair of lines in $\mathcal{L}\times\mathcal{L}'$ can be computed in time $O(n^{3+\varepsilon}\log m + m\log^2 n)$ for any $\varepsilon > 0$.*

Next, we describe another algorithm for computing $\delta^* = d(\mathcal{L},\mathcal{L}')$, which uses the above procedure as a subroutine. We first describe an algorithm for the "fixed-size" problem that, given a parameter δ, determines whether $\delta^* < \delta$, $\delta^* = \delta$, or $\delta^* > \delta$. Next, we apply the parametric-search technique to this algorithm, with an additional twist, to compute $d(\mathcal{L},\mathcal{L}')$.

Consider the fixed-size problem. If $m > n^{3+\varepsilon}$, the problem can be solved in $O(m\log^2 n)$ time by computing δ^* using Lemma 5.1 and then comparing δ^* with δ. So assume that $m \le n^{3+\varepsilon}$.

We now map each line $\ell' \in \mathcal{L}'$ to the point $\psi(\ell')$, as defined above, and each line $\ell \in \mathcal{L}$ to a surface $\phi(\ell)$, which is the locus of all points $\psi(\ell')$ such that $d(\ell,\ell') = \delta$ and ℓ' lies below ℓ. (This is a dual representation of the problem, where the roles of $\mathcal{L}$ and of $\mathcal{L}'$ are interchanged.) The open region ϕ^- lying below $\phi(\ell)$ consists of points corresponding to lines ℓ' that lie below ℓ and satisfy $d(\ell,\ell') > \delta$. Again, each surface $\phi(\ell)$ is $x_1x_2x_3$-monotone; i.e., $\phi(\ell)$ is the graph of a partial function $x_4 = g_\ell(x_1,x_2,x_3)$. Let

$$G(x_1,x_2,x_3) = \min_{\ell\in\mathcal{L}} g_\ell(x_1,x_2,x_3)$$

be the lower envelope of these functions. By the same argument as above, $d(\mathcal{L},\mathcal{L}') < \delta$ if and only if there is a point $\psi(\ell')$, for some $\ell' \in \mathcal{L}'$, that lies above the lower envelope G, and $d(\mathcal{L},\mathcal{L}') = \delta$ if and only if no point corresponding to the lines of $\mathcal{L}'$ lies above G and at least one such point lies on G. To detect these two conditions, we proceed as follows.

(i) Fix a sufficiently large constant r. Choose a random subset $R \subset \mathcal{L}$ of size $t = cr\log r$, where c is some appropriate constant independent of r.

(ii) Let $\Gamma_R = \{g_\ell \mid \ell \in R\}$ be the set of t trivariate functions corresponding to the lines of R, and let G_R be the lower envelope of Γ_R. Decompose the region of $\mathbb{R}^4$ lying below G_R into a collection $\Xi = \{\tau_1, \ldots, \tau_k\}$ of $k = O(t^4\beta(t)) = O(r^4\beta(r)\log^4 r)$ semialgebraic cells of constant description complexity each; here $\beta(r)$ is a slowly growing function whose exact form is determined by the algebraic degree of the surfaces in Γ. Such a decomposition can be obtained using the algorithms described in [AST, CEGSb]. It is based on a decomposition of $\mathbb{R}^3$ into a family Ξ' of $O(r^4\beta(r)\log^4 r)$ cells of constant description complexity, so that, within each cell, the same set of

function graphs appear on the lower envelope G_R. The decomposition Ξ is then defined as the following collection of semiunbounded "prisms":

$$\Xi = \Big\{ \{(\mathbf{x}, z) \mid \mathbf{x} \in \Delta, z \le G_R(\mathbf{x})\} \mid \Delta \in \Xi' \Big\}.$$

(iii) For each prism $\tau \in \Xi$, let $\mathcal{L}_\tau = \{\ell \in \mathcal{L} \mid \phi(\ell) \cap \tau \neq \emptyset\}$ and $\mathcal{L}'_\tau = \{\ell' \in \mathcal{L}' \mid \psi(\ell') \in \tau\}$.

(iv) If $\bigcup_{\tau \in \Xi} \mathcal{L}'_\tau \neq \mathcal{L}'$ (i.e., there is a line $\ell' \in \mathcal{L}'$ such that the point $\psi(\ell')$ lies above G_R), then return $\delta > \delta^*$. If there is a $\tau \in \Xi$ such that $|\mathcal{L}_\tau| > m/r + 1$, then go back to Step (i).

(v) For each prism $\tau \in \Xi$, if both $\mathcal{L}_\tau, \mathcal{L}'_\tau$ are nonempty, then solve the problem recursively for $\mathcal{L}_\tau$ and $\mathcal{L}'_\tau$. If there is a subproblem for which $\delta > d(\mathcal{L}_\tau, \mathcal{L}'_\tau)$, then return $\delta > \delta^*$. If there is no such subproblem, but there is one subproblem for which $\delta = \delta^*$, then return $\delta = \delta^*$. Finally, if none of these two cases occur, then return $\delta < \delta^*$.

The correctness of the algorithm follows from the above observations. As for the running time, the well-known results on random sampling [ClS, HW] imply that if c is chosen sufficiently large then, with high probability, $|\mathcal{L}_\tau| \le m/r + 1$ for every $\tau \in \Xi$. Hence, the expected number of times we have to go back to step (i) from step (iv) is only a constant. Let $T(m, n)$ denote the expected running time of the algorithm. Then

$$T(m,n) = \begin{cases} O(m \log^2 n) & \text{if } m \ge n^{3+\varepsilon}, \\ \displaystyle\sum_{i=1}^{k} T\Big(\frac{m}{r} + 1, n_i\Big) + O(m+n) & \text{if } m < n^{3+\varepsilon}, \end{cases}$$

where $\sum_i n_i = n$ and $k = O(r^4 \beta(r) \log^4 r)$. The solution of the above recurrence is easily seen to be

$$T(m,n) = O(m^{8/11+\varepsilon'} n^{9/11+\varepsilon'} + m^{1+\varepsilon'} + n^{1+\varepsilon'})$$

for any $\varepsilon' > \varepsilon$.

Next, we describe how to apply the parametric-search technique to this algorithm. As mentioned earlier, we cannot use this algorithm directly in the parametric-search paradigm, because we do not know how to parallelize the first algorithm. We therefore simulate the above sequential algorithm at $\delta = \delta^*$ with the additional twist that, instead of just simulating the solution of the fixed-size problem at δ^*, the algorithm will attempt to compute δ^* explicitly. Since r is chosen to be constant, the set Ξ can be computed by making only $r^{O(1)}$ implicit "sign tests" involving δ^*, thus only a constant number, $r^{O(1)}$, of solutions of the fixed-size problem are needed; see [AST, CEGSb] for details. It can be checked that the problem of determining whether a surface $\phi(\ell)$ intersects a prism $\tau \in \Xi$ or whether a point $\psi(\ell')$ lies in τ can be reduced to computing the signs of a constant number of univariate polynomials, each of constant degree, at δ^* (this follows since τ has constant description complexity). Let $\Pi = \{p_1(\delta), \ldots, p_s(\delta)\}$, where $s = O(m+n)$, be the set of these polynomials, over all lines of $\mathcal{L}, \mathcal{L}'$ and over all prisms of Ξ. Let $\delta_1 < \cdots < \delta_u$ be the real roots of these polynomials, where u is also $O(m+n)$. By a binary search over these roots we compute the largest root δ_α such that $\delta_\alpha \le d(\mathcal{L}, \mathcal{L}')$. Each step of the binary search involves comparing δ^* with some δ_i, which in turn involves solving an instance of the fixed-size problem for some δ_i. If $\delta_\alpha = \delta^*$, we have found the value of δ^* so we stop right away. Thus,

assume that $\delta_\alpha < \delta^*$. We can now easily resolve the signs of all polynomials of Π by evaluating them at $(\delta_\alpha + \delta_{\alpha+1})/2$. Once we have computed $\mathcal{L}_\tau, \mathcal{L}'_\tau$ for all $\tau \in \Xi$, we compute $d(\mathcal{L}_\tau, \mathcal{L}'_\tau)$ recursively and return $\min_{\tau\in\Xi} d(\mathcal{L}_\tau, \mathcal{L}'_\tau)$ as $d(\mathcal{L}, \mathcal{L}')$. (Note that $\bigcup_{\tau\in\Xi} \mathcal{L}'_\tau = \mathcal{L}'$, since we are simulating the algorithm at δ^*.) The correctness of the algorithm is established by the following lemma.

LEMMA 5.2. *If none of the calls to the fixed-size problem in steps* (i)–(iv) *returns the value of* $d(\mathcal{L}, \mathcal{L}')$, *then* $d(\mathcal{L}, \mathcal{L}') = \min_{\tau\in\Xi} d(\mathcal{L}_\tau, \mathcal{L}'_\tau)$.

Proof. Let $\delta^* = d(\mathcal{L}, \mathcal{L}')$, and let $\ell \in \mathcal{L}$, $\ell' \in \mathcal{L}'$ be a pair of lines such that $d(\ell, \ell') = \delta^*$. Since we are simulating the fixed-size problem at $\delta = \delta^*$, no point corresponding to the lines of $\mathcal{L}'$ lies above the lower envelope of the surfaces defined by $\mathcal{L}$ (for $\delta = \delta^*$), that is, there exists a prism $\tau \in \Xi$ such that $\ell' \in \mathcal{L}'_\tau$ (i.e., $\psi(\ell')$ lies in τ). If ℓ does not belong to $\mathcal{L}_\tau$, then $\psi(\ell')$ lies below $\phi(\ell)$, which implies that $d(\ell, \ell') > \delta^*$, a contradiction. Hence the pair (ℓ, ℓ') appears in the subproblem involving $\mathcal{L}_\tau$ and $\mathcal{L}'_\tau$. □

Let $T'(m, n)$ denote the maximum expected running time of the algorithm. Then we obtain the following recurrence

$$T'(m,n) = \begin{cases} O(m \log^2 n) & \text{if } m \geq n^{3+\varepsilon}, \\ \displaystyle\sum_{i=1}^{O(r^4\beta(r)\log^4 r)} T'\Big(\frac{m}{r} + 1, n_i\Big) + r^{O(1)} O(m^{8/11+\varepsilon} n^{9/11+\varepsilon} + m^{1+\varepsilon} + n^{1+\varepsilon}) & \\ & \text{if } m < n^{3+\varepsilon}. \end{cases}$$

The solution of the above recurrence is also

$$T'(m,n) = O(m^{8/11+\varepsilon'} n^{9/11+\varepsilon'} + m^{1+\varepsilon'} + n^{1+\varepsilon'})$$

for a different but still arbitrarily small constant $\varepsilon' > \varepsilon$.

Hence, we obtain the following result.

LEMMA 5.3. *Given a set* $\mathcal{L}$ *of* m *lines and another set* $\mathcal{L}'$ *of* n *lines such that all lines in* $\mathcal{L}$ *lie above all lines of* $\mathcal{L}'$, *the closest pair of lines in* $\mathcal{L} \times \mathcal{L}'$ *can be computed in randomized expected time* $O(m^{8/11+\varepsilon} n^{9/11+\varepsilon} + m^{1+\varepsilon} + n^{1+\varepsilon})$ *for any* $\varepsilon > 0$.

Finally, we apply Lemma 5.3 to all subsets $(\mathcal{E}_i, \mathcal{E}'_i)$ of $\mathcal{F}$ (see (1)) and output the minimum of the distances obtained for each subproblem. This is equal to the minimum distance between any pair of parallel planes, each supporting an edge of $\mathcal{P}$. The total expected running time is

$$\sum_{i=1}^{t} O\left(|\mathcal{E}_i|^{8/11+\varepsilon} |\mathcal{E}'_i|^{9/11+\varepsilon} + |\mathcal{E}_i|^{1+\varepsilon} + |\mathcal{E}'_i|^{1+\varepsilon}\right) = O(n^{17/11+\varepsilon'}),$$

where ε' is yet another but still arbitrarily small positive constant.

Putting everything together, we can conclude the following.

THEOREM 5.4. *The width of a set of* n *points in* $\mathbb{R}^3$ *can be computed in randomized expected time* $O(n^{17/11+\varepsilon})$, *for any* $\varepsilon > 0$.

Remark. Informally speaking, the "ugly" exponent 17/11 is the result of an interaction between the exponent 3, appearing in the bound of Theorem 3.3, and the exponent 4, appearing in the bound for the size of the vertical decomposition Ξ used above. After the original submission of this paper, Agarwal and Sharir [ASb] observed that, for the width problem and for the two problems studied below, one can show that the size of this vertical decomposition is only near cubic in the number of

surfaces. This in turn leads to an improved analysis of the above algorithm and implies that its running time is only $O(n^{3/2+\varepsilon})$ for any $\varepsilon > 0$ (and a similar improvement applies to the two subsequent algorithms given below). Agarwal and Sharir [ASb] gave a different randomized algorithm for these problems, without using parametric searching.

5.2. Biggest stick in a simple polygon. Let P be a simple polygon in the plane having n edges. We wish to find the longest line segment e that is contained in (the closed set) P. Chazelle and Sharir presented in [CS] an $O(n^{1.99})$-time algorithm for this problem, which was later improved by Agarwal, Sharir, and Toledo [AST] to $O(n^{8/5+\varepsilon})$ for any $\varepsilon > 0$. The latter paper reduces this problem, just as in the computation of the width of a point set in 3-space, to that of finding the extreme value of a certain function of two parameters, optimized over all intersection points between the edges of two straight-edge planar maps. This problem, in turn, can be reduced to the problem of optimizing the function over all intersection points of pairs of lines in $\mathcal{L}_1 \times \mathcal{L}_2$, where $\mathcal{L}_1, \mathcal{L}_2$ are two appropriate families of lines in the plane. By regarding the lines of $\mathcal{L}_1$ as data points, and each line of $\mathcal{L}_2$ as inducing a certain function over the lines of $\mathcal{L}_1$, and by using an appropriate parametrization of these points and functions, the problem can be reduced to that of testing whether any point in a certain set of points in 4-space lies above the lower envelope of a certain collection of trivariate functions. Combining the approach of [AST] with the one described in the previous subsection, we can compute the longest segment e in expected time $O(n^{17/11+\varepsilon})$ for any $\varepsilon > 0$. Omitting all the details, which can be found in [AST], we conclude the following.

THEOREM 5.5. *One can compute the biggest stick that fits inside a simple polygon with n edges, in randomized expected time $O(n^{17/11+\varepsilon})$, for any $\varepsilon > 0$.*

5.3. Minimum-width annulus. Let S be a set of n points in the plane. We wish to find the (closed) annulus of smallest width that contains S, i.e., we want to compute two concentric disks D_1 and D_2 of radii r_1 and r_2, such that all points of S lie in the closure of $D_2 \setminus D_1$ and $r_2 - r_1$ is minimized. This problem has been studied in [AST]. It is known [AST, EFNN] that the center of such an annulus is a vertex of the closest-point Voronoi diagram, $\mathrm{Vor}_c(S)$, of S, a vertex of the farthest-point Voronoi diagram, $\mathrm{Vor}_f(S)$, of S, or the intersection point of an edge of $\mathrm{Vor}_c(S)$ and an edge of $\mathrm{Vor}_f(S)$. The difficult part is testing the intersection points of edges of the two diagrams, because there can be $\Theta(n^2)$ such points in the worst case. Following the same idea as in [AST], which reduces the problem to that of batched searching of points relative to an envelope of functions in four dimensions, but using the technique described above for computing the width in 3-space, we obtain the following result (see [AST] for details).

THEOREM 5.6. *The smallest-width annulus containing a set of n points in the plane can be computed in randomized expected time $O(n^{17/11+\varepsilon})$, for any $\varepsilon > 0$.*

REFERENCES

[AMa] P. AGARWAL AND J. MATOUŠEK, *Ray shooting and parametric search,* SIAM J. Comput., 22 (1993), pp. 794–806.

[AMb] P. AGARWAL AND J. MATOUŠEK, *Range searching with semialgebraic sets,* Discrete Comput. Geom., 11 (1994), pp. 393–418.

[ASS] P. AGARWAL, O. SCHWARZKOPF, AND M. SHARIR, *The overlay of lower envelopes and its applications,* Discrete Comput. Geom., 15 (1996), pp. 1–13.

[ASa] P. Agarwal and M. Sharir, *On the number of views of polyhedral terrains,* Discrete Comput. Geom., 12 (1994), pp. 177–182.

[ASb] P. Agarwal and M. Sharir, *Efficient randomized algorithms for some geometric optimization problems,* Discrete Comput. Geom., 16 (1996), pp. 317–337.

[AST] P. Agarwal, M. Sharir, and S. Toledo, *New applications of parametric searching in computational geometry,* J. Algorithms, 17 (1994), pp. 292–318.

[AGPS] P. Agarwal, L. Guibas, M. Pellegrini, and M. Sharir, *Ray Shooting among Spheres,* manuscript, 1992.

[ArS] B. Aronov and M. Sharir, *Triangles in space, or: Building (and analyzing) castles in the air,* Combinatorica, 10 (1990), pp. 137–173.

[BDS+] J. D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec, *Applications of random sampling to on-line algorithms in computational geometry,* Discrete Comput. Geom., 8 (1992), pp. 51–71.

[BD] J. D. Boissonnat and K. Dobrindt, *On-line randomized construction of the upper envelope of triangles and surface patches in* $\mathbb{R}^3$*,* Comput. Geom. Theory Appl., 5 (1996), pp. 303–320.

[BSTY] J. D. Boissonnat, M. Sharir, B. Tagansky, and M. Yvinec, *Voronoi diagrams in higher dimensions under certain polyhedral convex distance functions,* in Proc. 11th ACM Symp. Comput. Geom., Vancouver, 1995, ACM, New York, pp. 79–88.

[CEGSa] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *Algorithms for bichromatic line segment problems and polyhedral terrains*, Algorithmica, 11 (1994), pp. 116–132.

[CEGSb] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *Diameter, width, closest line pair, and parametric searching,* Discrete Comput. Geom., 10 (1993), pp. 183–196.

[CEGSS] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Snoeyink, *Computing a single face in an arrangement of line segments and related problems,* SIAM J. Computing, 22 (1993), pp. 1286–1302.

[CS] B. Chazelle and M. Sharir, *An algorithm for generalized point location and its applications,* J. Symbolic Comput., 10 (1990), pp. 281–309.

[CKSW] L. P. Chew, K. Kedem, M. Sharir, B. Tagansky, and E. Welzl, *Voronoi diagrams of lines in three dimensions under a polyhedral convex distance function,* in Proc. 6th ACM-SIAM Symp. on Discrete Algorithms, San Francisco, 1995, SIAM, Philadelphia, pp. 197–204.

[ClS] K. Clarkson and P. Shor, *Applications of random sampling in computational geometry* II, Discrete Comput. Geom., 4 (1989), pp. 387–421.

[dBDS] M. de Berg, K. Dobrindt, and O. Schwarzkopf, *On lazy randomized incremental construction,* Discrete Comput. Geom., 14 (1995), pp. 261–286.

[EFNN] H. Ebara, N. Fukuyama, H. Nakano, and Y. Nakanishi, *Roundness algorithms using the Voronoi diagrams,* in First Canadian Conf. Comput. Geom., abstract, 1989.

[ES] H. Edelsbrunner and R. Seidel, *Voronoi diagrams and arrangements,* Discrete Comput. Geom., 1 (1986), pp. 25–44.

[HS] D. Halperin and M. Sharir, *New bounds for lower envelopes in* 3 *dimensions, with applications to visibility in terrains,* Discrete Comput. Geom., 12 (1994), pp. 313–326.

[HW] D. Haussler and E. Welzl, ϵ*-nets and simplex range queries*, Discrete Comput. Geom., 2 (1987), pp. 127–151.

[HRR] J. Heintz, T. Recio, and M.-F. Roy, *Algorithms in real algebraic geometry and applications to computational geometry,* in Discrete and Computational Geometry: Papers from the DIMACS Special Year, J.E. Goodman, R. Pollack, and W. Steiger, eds., AMS Press, Providence, RI, 1991, pp. 137–163.

[HT] M. Houle and G. Toussaint, *Computing the width of a set,* IEEE Trans. Pattern Matching and Machine Intelligence, 5 (1988), pp. 761–765.

[Me] N. Megiddo, *Applying parallel computation algorithms in the design of serial algorithms,* J. Assoc. Comput. Mach., 30 (1983), pp. 852–865.

[MS] S. Mohaban and M. Sharir, *Ray shooting amidst spheres in three dimensions,* SIAM J. Comput., 26 (1997), pp. 654–674.

[Mua] K. Mulmuley, *A fast planar partition algorithm*, I, J. Symbolic Comput., 10 (1990), pp. 253–280.

[Mub] K. Mulmuley, *A fast planar partition algorithm*, II, J. Assoc. Comput. Mach., 38 (1991), pp. 74–103.

[PS] J. Pach and M. Sharir, *The upper envelope of piecewise linear functions and the bound-*

ary of a region enclosed by convex plates: Combinatorial analysis, Discrete Comput. Geom., 4 (1989), pp. 291–309.

[PT] F. P. Preparata and R. Tamassia, *Efficient point location in a convex spatial cell-complex,* SIAM J. Comput., 21 (1992), pp. 267–280.

[Sha] M. Sharir, *Almost tight upper bounds for lower envelopes in higher dimensions,* Discrete Comput. Geom., 12 (1994), pp. 327–345.

[SA] M. Sharir and P. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, New York, 1995.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1733–1748, December 1997

009

A SPECTRAL TECHNIQUE FOR COLORING RANDOM 3-COLORABLE GRAPHS*

NOGA ALON† AND NABIL KAHALE‡

Abstract. Let $G_{3n,p,3}$ be a random 3-colorable graph on a set of $3n$ vertices generated as follows. First, split the vertices arbitrarily into three equal color classes, and then choose every pair of vertices of distinct color classes, randomly and independently, to be edges with probability p. We describe a polynomial-time algorithm that finds a proper 3-coloring of $G_{3n,p,3}$ with high probability, whenever $p \geq c/n$, where c is a sufficiently large absolute constant. This settles a problem of Blum and Spencer, who asked if an algorithm can be designed that works almost surely for $p \geq \text{polylog}(n)/n$ [*J. Algorithms*, 19 (1995), pp. 204–234]. The algorithm can be extended to produce optimal k-colorings of random k-colorable graphs in a similar model as well as in various related models. Implementation results show that the algorithm performs very well in practice even for moderate values of c.

Key words. graph eigenvalues, graph coloring, algorithms, random graphs

AMS subject classifications. 05C15, 05C80, 05C85, 68R10

PII. S0097539794270248

1. Introduction.

A vertex coloring of a graph G is *proper* if no adjacent vertices receive the same color. The *chromatic number* $\chi(G)$ of G is the minimum number of colors in a proper vertex coloring of G. The problem of determining or estimating this parameter has received a considerable amount of attention in combinatorics and in theoretical computer science, as several scheduling problems are naturally formulated as graph coloring problems. It is well known (see [13, 12]) that the problem of properly coloring a graph of chromatic number k with k colors is NP-hard, even for any fixed $k \geq 3$, and it is therefore unlikely that there are efficient algorithms for optimally coloring an arbitrary 3-chromatic input graph.

On the other hand, various researchers noticed that *random* k-colorable graphs are usually easy to color optimally. Polynomial-time algorithms that optimally color random k-colorable graphs, for every fixed k with high probability, have been developed by Kucera [15], Turner [18], and Dyer and Frieze [8]; the latter paper provides an algorithm whose average running time over all k-colorable graphs on n vertices is polynomial. Note, however, that most k-colorable graphs are quite dense and, hence, easy to color. In fact, in a typical k-colorable graph, the number of common neighbors of any pair of vertices with the same color exceeds considerably that of any pair of vertices of distinct colors, and hence a simple coloring algorithm based on this fact already works with high probability. It is more difficult to color sparser random k-colorable graphs. A precise model for generating sparse random k-colorable graphs is described in the next subsection, where the sparsity is governed by a parameter p that specifies the edge probability. Petford and Welsh [16] suggested a randomized

*Received by the editors June 24, 1994; accepted for publication (in revised form) December 6, 1995. A preliminary version of this paper appeared in the Proceedings of the 26th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1994, pp. 346–355.
http://www.siam.org/journals/sicomp/26-6/27024.html

†Institute for Advanced Study, Princeton, NJ 08540 and Department of Mathematics, Tel-Aviv University, Tel Aviv, Israel (noga@math.tau.ac.il). This research was supported in part by Sloan Foundation grant 93-6-6 and by a USA–Israel BSF grant.

‡AT&T Bell Laboratories, Murray Hill, NJ 07974 (kahale@research.att.com). This work was done while the author was at DIMACS.

heuristic for 3-coloring random 3-colorable graphs and supplied experimental evidence that it works for most edge probabilities. Blum and Spencer [6] (also see [3] for some related results) designed a polynomial algorithm and proved that it optimally colors, with high probability, random 3-colorable graphs on n vertices with edge probability p provided $p \geq n^{\epsilon}/n$ for some arbitrarily small but fixed $\epsilon > 0$. Their algorithm is based on a path-counting technique and can be viewed as a natural generalization of the simple algorithm based on counting common neighbors (that counts paths of length 2) mentioned above.

Our main result here is a polynomial-time algorithm that works for sparser random 3-colorable graphs. If the edge probability p satisfies $p \geq c/n$, where c is a sufficiently large absolute constant, the algorithm optimally colors the corresponding random 3-colorable graph with high probability. This settles a problem of Blum and Spencer [6] who asked if an algorithm can be designed that works almost surely for $p \geq \text{polylog}(n)/n$. (Here and in what follows, *almost surely* always means "with probability that approaches 1 as n tends to infinity.") The algorithm uses the spectral properties of the graph and is based on the fact that, almost surely, a rather accurate approximation of the color classes can be read from the eigenvectors corresponding to the smallest two eigenvalues of the adjacency matrix of a large subgraph. This approximation can then be improved to yield a proper coloring.

The algorithm can be easily extended to the case of k-colorable graphs, for any fixed k, and to various models of random *regular* 3-colorable graphs.

We implemented our algorithm and tested it for hundreds of graphs drawn at random from the distribution of $G_{3n,p,3}$. Experiments show that our algorithm performs very well in practice. The running time is a few minutes on graphs with up to 100,000 nodes, and the range of edge probabilities on which the algorithm is successful is in fact even larger than what our analysis predicts.

1.1. The model. There are several possible models for random k-colorable graphs. See [8] for some of these models and for the relation between them. Our results hold for most of these models, but it is convenient to focus on one, which will simplify the presentation. Let V be a fixed set of kn labeled vertices. For a real $p = p(n)$, let $G_{kn,p,k}$ be the random graph on the set of vertices V obtained as follows: first, split the vertices of V arbitrarily into k color classes $W_1, \ldots, W_k$, each of cardinality n. Next, for each u and v that lie in distinct color classes, choose uv to be an edge, randomly and independently, with probability p. The input to our algorithm is a graph $G_{kn,p,k}$ obtained as above, and the algorithm succeeds to color it if it finds a proper k coloring. Here we are interested in fixed $k \geq 3$ and large n. We say that an algorithm colors $G_{kn,p,k}$ *almost surely* if the probability that a randomly chosen graph, as above, is properly colored by the algorithm tends to one as n tends to infinity. Note that we consider here deterministic algorithms, and the above statement means that the algorithm succeeds in coloring almost all random graphs generated.

A closely related model to the one given above is the model in which we do not insist that the color classes have equal sizes. In this model one first splits the set of vertices into k disjoint color classes by letting each vertex choose its color randomly, independently, and uniformly, among the k possibilities. Next, one chooses every pair of vertices of distinct color classes to be edges with probability p. All of our results hold for both models, and we focus on the first one as it is more convenient. To simplify the presentation, we restrict our attention to the case $k = 3$ of 3-colorable graphs, since the results for this case easily extend to every *fixed* k. In addition, we

make no attempt to optimize the constants and assume, whenever this is needed, that c is a sufficiently large constant, and the number of vertices $3n$ is sufficiently large.

1.2. The algorithm. Here is a description of the algorithm, which consists of three phases. Given a graph $G = G_{3n,p,3} = (V, E)$, define $d = pn$. Let $G' = (V, E')$ be the graph obtained from G by deleting all edges incident to a vertex of degree greater than $5d$. Denote by A the adjacency matrix of G', i.e., the $3n$ by $3n$ matrix $(a_{uv})_{u,v \in V}$ defined by $a_{uv} = 1$ if $uv \in E'$ and $a_{uv} = 0$ otherwise. It is well known that, since A is symmetric, it has real eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{3n}$ and an orthonormal basis of eigenvectors $e_1, e_2, \ldots, e_{3n}$, where $Ae_i = \lambda_i e_i$. The crucial point is that, almost surely, one can deduce a good approximation of the coloring of G from e_{3n-1} and e_{3n}. Note that there are several efficient algorithms to compute the eigenvalues and the eigenvectors of symmetric matrices (cf. [17]) and hence e_{3n-1} and e_{3n} can certainly be calculated in polynomial time. For the remainder of the algorithm, we will deal with G rather than G'.

Let t be a nonzero linear combination of e_{3n-1} and e_{3n} whose median is zero; that is, the number of positive components of t as well as the number of its negative components are both at most $3n/2$. (It is easy to see that such a combination always exists and can be found efficiently.) Suppose also that t is normalized so that its l_2-norm is $\sqrt{2n}$. Define $V_1^0 = \{u \in V : t_u > 1/2\}$, $V_1^0 = \{u \in V : t_u < -1/2\}$, and $V_3^0 = \{u \in V : |t_u| \leq 1/2\}$. This is an approximation for the coloring, which will be improved in the second phase by iterations, and then in the third phase to obtain a proper 3-coloring.

In iteration i of the second phase, $0 < i \leq q = \lceil \log n \rceil$, construct the color classes V_1^i, V_2^i, and V_3^i as follows. For every vertex v of G, let $N(v)$ denote the set of all its neighbors in G. In the ith iteration, color v by the least popular color of its neighbors in the previous iteration. That is, put v in V_j^i if $|N(v) \cap V_j^{i-1}|$ is the minimum among the three quantities $|N(v) \cap V_l^{i-1}|$, $l = 1, 2, 3$, where equalities are broken arbitrarily. We will show that the three sets V_i^q correctly color all but $n2^{-\Omega(d)}$ vertices.

The third phase consists of two stages. First, repeatedly uncolor every vertex colored j that has less than $d/2$ neighbors (in G) colored l for some $l \in \{1, 2, 3\} - \{j\}$. Then, if the graph induced on the set of uncolored vertices has a connected component of size larger than $\log_3 n$, the algorithm fails. Otherwise, find a coloring of every component consistent with the rest of the graph using brute-force exhaustive search. If the algorithm cannot find such a coloring, it fails.

Our main result is the following.

THEOREM 1.1. *If $p > c/n$, where c is a sufficiently large constant, the algorithm produces a proper 3-coloring of G with probability $1 - o(1)$.*

The intuition behind the algorithm is as follows. Suppose that every vertex in G had exactly d neighbors in every color class other than its own. Then $G' = G$. Let F be the two-dimensional subspace of all vectors $x = (x_v : v \in V)$ which are constant on every color class, and whose sum is 0. A simple calculation (as observed in [1]) shows that any nonzero element of F is an eigenvector of A with eigenvalue $-d$. Moreover, if E is the union of random matchings, one can show that $-d$ is almost surely the smallest eigenvalue of A and that F is precisely the eigenspace corresponding to $-d$. Thus, any linear combination t of e_{3n-1} and e_{3n} is constant on every color class. If the median of t is 0 and its l_2-norm is $\sqrt{2n}$, then t takes the values 0, 1, or -1 depending on the color class, and the coloring obtained after phase 1 of the algorithm is a proper coloring. In the model specified in subsection 1.1 these regularity assumptions do not hold, but every vertex has the same *expected* number of neighbors in every color class

other than its own. This is why phase 1 gives only an approximation of the coloring and phases 2 and 3 are needed to obtain a proper coloring.

We prove Theorem 1.1 in the next two sections. We use the fact that, almost surely, the largest eigenvalue of G' is at least $(1-2^{-\Omega(d)})2d$, and that its two smallest eigenvalues are at most $-(1-2^{-\Omega(d)})d$ and all other eigenvalues are in absolute value $O(\sqrt{d})$. The proof of this result is based on a proper modification of techniques developed by Friedman, Kahn, and Szemerédi in [11] and is deferred to section 3. We show in section 2 that our modification implies that each of the two eigenvectors corresponding to the two smallest eigenvalues is close to a vector, which is a constant on every color class, where the sum of these three constants is zero. This suffices to show that the sets V_j^0 form a reasonably good approximation to the coloring of G with high probability.

Theorem 1.1 can then be proved by applying the expansion properties of the graph G (that hold almost surely) to show that the iteration process above converges quickly to a proper coloring of a predefined large subgraph H of G. The uncoloring procedure will uncolor all vertices which are wrongly colored, but will not affect the subgraph H. We then conclude by showing that the largest connected component of the induced subgraph of G on $V-H$ is of logarithmic size almost surely, thereby showing that the brute-force search on the set of uncolored vertices terminates in polynomial time. We present our implementation results in section 4. Section 5 contains some concluding remarks together with possible extensions and results for related models of random graphs.

2. The proof of the main result. Let $G = G_{3n,p,3} = (V, E)$ be a random 3-colorable graph generated according to the model described above. Denote by W_1, W_2, and W_3 the three color classes of vertices of G. Let G' be the graph obtained from G by deleting all edges adjacent to vertices of degree greater than $5d$, and let A be the adjacency matrix of G'. Denote by $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{3n}$ the eigenvalues of A and by $e_1, e_2, \ldots, e_{3n}$ the corresponding eigenvectors, chosen so that they form an orthonormal basis of R^{3n}.

In this section we first show that the approximate coloring produced by the algorithm using the eigenvectors e_{3n-1} and e_{3n} is rather accurate almost surely. Then we exhibit a large subgraph H and show that, almost surely, the iterative procedure for improving the coloring colors H correctly. We then show that the third phase finds a proper coloring of G in polynomial time, almost surely. We use the following statement, whose proof is relegated to section 3.

PROPOSITION 2.1. *In the above notation, almost surely,*

(i) $\lambda_1 \geq (1-2^{-\Omega(d)})2d$,

(ii) $\lambda_{3n} \leq \lambda_{3n-1} \leq -(1-2^{-\Omega(d)})d$, *and*

(iii) $|\lambda_1| \leq O(\sqrt{d})$ *for all* $2 \leq i \leq 3n-2$.

Remark. One can show that, when $p = o(\log n/n)$, Proposition 2.1 would not hold if we were dealing with the spectrum of G rather than that of G', since the graph G is likely to contain many vertices of degree $\gg d$, and in this case the assertion of (iii) does not hold for the eigenvalues of G.

2.1. The properties of the last two eigenvectors. In this subsection we show that the eigenvectors e_{3n-1} and e_{3n} are almost constant on every color class. For this, we exhibit two orthogonal vectors constant on every color class which, roughly speaking, are close to being eigenvectors corresponding to $-d$. Let $x = (x_v : v \in V)$ be the vector defined by $x_v = 2$ for $v \in W_1$, and $x_v = -1$ otherwise. Let $y = (y_v : v \in V)$

be the vector defined by $y_v = 0$ if $v \in W_1$, $y_v = 1$ if $v \in W_2$, and $y_v = -1$ if $v \in W_3$. We denote by $\|f\|$ the l_2-norm of a vector f.

LEMMA 2.2. *Almost surely there are two vectors $\epsilon = (\epsilon_v : v \in V)$ and $\delta = (\delta_v : v \in V)$, satisfying $\|\epsilon\|^2 = O(n/d)$ and $\|\delta\|^2 = O(n/d)$ so that $x - \epsilon$ and $y - \delta$ are both linear combinations of e_{3n-1} and e_{3n}.*

Proof. We use the following lemma, whose proof is given below.

LEMMA 2.3. *Almost surely, $\|(A + dI)y\|^2 = O(nd)$ and $\|(A + dI)x\|^2 = O(nd)$.*

We prove the existence of δ as above. The proof of the existence of ϵ is analogous. Let $y = \sum_{i=1}^{3n} c_i e_i$. We show that the coefficients $c_1, c_2, \ldots, c_{3n-2}$ are small compared to $\|y\|$. Indeed, $(A + dI)y = \sum_{i=1}^{3n} c_i(\lambda_i + d)e_i$, and so

$$\|(A + dI)y\|^2 = \sum_{i=1}^{3n} c_i^2(\lambda_i + d)^2 \geq \Omega(d^2) \sum_{i=1}^{3n-2} c_i^2, \tag{2.1}$$

where the last inequality follows from parts (i) and (iii) of Proposition 2.1. Define $\delta = \sum_{i=1}^{3n-2} c_i e_i$. By equation (1) and Lemma 2.3, it follows that $\|\delta\|^2 = \sum_{i=1}^{3n-2} c_i^2 = O(n/d)$. On the other hand, $y - \delta$ is a linear combination of e_{3n-1} and e_{3n}. □

Note that it was crucial to the proof of Lemma 2.2 that, almost surely, $\|(A+dI)y\|^2$ is $O(nd)$ rather than $\Omega(nd)^2$ as is the case for some vectors in $\{-1, 0, 1\}^{3n}$.

Proof of Lemma 2.3. To prove the first bound, observe that it suffices to show that the sum of squares of the coordinates of $(A+dI)y$ on W_1 is $O(nd)$ almost surely, as the sums on W_2 and W_3 can be bounded similarly. The expectation of the vector $(A+dI)y$ is the null vector, and the expectation of the square of each coordinate of $(A + dI)y$ is $O(d)$ by a standard calculation. This is because each coordinate of $(A + dI)y$ is the sum of n independent random variables, each with mean 0 and variance $O(d/n)$. This implies that the *expected* value of the sum of squares of the coordinates of $(A + dI)y$ on W_1 is $O(nd)$. Similarly, the expectation of the fourth power of each coordinate of $(A + dI)y$ is $O(d^2)$. Hence the variance of the square of each coordinate is $O(d^2)$. However, the coordinates of $(A + dI)y$ on W_1 are independent random variables, and hence the variance of the sum of the squares of the W_1 coordinates is equal to the sum of the variances, which is $O(nd^2)$. The first bound can now be deduced from Chebyshev's inequality. The second bound can be shown in a similar manner. We omit the details. □

The vectors $x - \epsilon$ and $y - \delta$ are independent since they are nearly orthogonal. Indeed, if $\alpha(x - \epsilon) + \beta(y - \delta) = 0$, then $\alpha x + \beta y = \alpha\epsilon + \beta\delta$, and so $6n\alpha^2 + 2n\beta^2 = \|\alpha\epsilon + \beta\delta\|^2$. However,

$$\begin{aligned}\|\alpha\epsilon + \beta\delta\| &\leq |\alpha|\ \|\epsilon\| + |\beta|\ \|\delta\| \\ &= O\left((|\alpha| + |\beta|)\sqrt{n/d}\right).\end{aligned}$$

Thus, $6\alpha^2 + 2\beta^2 = O((\alpha^2 + \beta^2)/d)$ and hence $\alpha = \beta = 0$.

Therefore, by the above lemma the two vectors $\sqrt{3n}e_{3n-1}$ and $\sqrt{3n}e_{3n}$ can be written as linear combinations of $x - \epsilon$ and $y - \delta$. Moreover, the coefficients in these linear combinations are all $O(1)$ in absolute value. That is because $x - \epsilon$ and $y - \delta$ are nearly orthogonal, and the l_2-norm of each of the four vectors $x - \epsilon$, $y - \delta$, $\sqrt{3n}e_{3n-1}$, and $\sqrt{3n}e_{3n}$ is $\Theta(\sqrt{n})$. More precisely, if one of the vectors $\sqrt{3n}e_{3n-1}$,

$\sqrt{3n}e_{3n}$ is written as $\alpha(x-\epsilon)+\beta(y-\delta)$, then by the triangle inequality, $\|\alpha x+\beta y\| \leq \Theta(\sqrt{n}) + |\alpha|\,\|\epsilon\| + |\beta|\,\|\delta\|$, which, by a calculation similar to the one above, implies that $6\alpha^2 + 2\beta^2 \leq O(1) + O((\alpha^2+\beta^2)/d)$, and thus α and β are $O(1)$. On the other hand, the coefficients of the vector t defined in subsection 1.2 along the vectors e_{3n-1} and e_{3n} are at most $\|t\| = \sqrt{2n}$. It follows that the vector t defined in the algorithm is also a linear combination of the vectors $x-\epsilon$ and $y-\delta$ with coefficients whose absolute values are both $O(1)$. Since both x and y belong to the vector space F defined in the proof of Proposition 2.1, this implies that $t = f+\eta$, where $f \in F$ and $\|\eta\|^2 = O(n/d)$. Let α_i be the value of f on W_i for $1 \leq i \leq 3$. Assume without loss of generality that $\alpha_1 \geq \alpha_2 \geq \alpha_3$. Since $\|\eta\|^2 = O(n/d)$, at most $O(n/d)$ of the coordinates of η are greater than 0.01 in absolute value. This implies that $|\alpha_2| \leq 1/4$, because otherwise at least $2n - O(n/d)$ coordinates of t would have the same sign, contradicting the fact that 0 is a median of t. As $\alpha_1+\alpha_2+\alpha_3 = 0$ and $\alpha_1^2+\alpha_2^2+a_3^2 = \|f\|^2/n = 2+O(d^{-1})$, this implies that $\alpha_1 > 3/4$ and $\alpha_3 < -3/4$. Therefore, the coloring defined by the sets V_j^0 agrees with the original coloring of G on all but at most $O(n/d) < 0.001n$ coordinates.

2.2. The iterative procedure. Denote by H the subset of V obtained as follows. First, set H to be the set of vertices having at most $1.01d$ neighbors in G in each color class. Then, repeatedly delete any vertex in H having less than $0.99d$ neighbors in H in some color class (other than its own). Thus, each vertex in H has roughly d neighbors in H in each color class other than its own.

PROPOSITION 2.4. *Almost surely, by the end of the second phase of the algorithm, all vertices in H are properly colored.*

To prove Proposition 2.4, we need the following lemma.

LEMMA 2.5. *Almost surely, there are no two subsets of vertices U and W of V such that $|U| \leq 0.001n$, $|W| = |U|/2$, and every vertex v of W has at least $d/4$ neighbors in U.*

Proof. Note that if there are two such (not necessarily disjoint) subsets U and W, then the number of edges joining vertices of U and W is at least $d|W|/8$. Therefore, by a standard calculation the probability that there exist two such subsets is at most

$$\begin{aligned}
\sum_{i=1}^{0.0005n} \binom{3n}{i}\binom{3n}{2i}\binom{2i^2}{di/8}\left(\frac{d}{n}\right)^{di/8} &\leq \sum_{i=1}^{0.0005n} \left(\frac{3en}{i}\right)^{3i}\left(\frac{16ei}{n}\right)^{di/8} \\
&\leq \sum_{i=1}^{0.0005n} \left(\frac{3en}{i}\right)^{di/40}\left(\frac{16ei}{n}\right)^{di/8} \\
&= \sum_{i=1}^{0.0005n} (48e^2)^{di/40}\left(\frac{16ei}{n}\right)^{di/10} \\
&\leq \sum_{i=1}^{0.0005n} (48e^2(16e/2000)^2)^{di/40}\left(\frac{16ei}{n}\right)^{di/20} \\
&\leq \sum_{i=1}^{0.0005n} \left(\frac{16ei}{n}\right)^{di/20} \\
&= O(1/n^{\Omega(d)}). \qquad \square
\end{aligned}$$

If a vertex in H is colored incorrectly at the end of iteration i of the algorithm in phase 2 (i.e., if it is colored j and does not belong to W_j), it must have more than $d/4$ neighbors in H colored incorrectly at the end of iteration $i-1$. To see this, observe that any vertex of H has at most $2(1.01d - 0.99d) = 0.04d$ neighbors outside H, and hence if it has at most $d/4$ wrongly colored neighbors in H, it must have at least $0.99d - d/4 > d/2$ neighbors of each color other than its correct color and at most $d/4 + 0.04d$ neighbors of its correct color. By repeatedly applying the property asserted by the above lemma, with U being the set of vertices of H whose colors in the end of the iteration $i-1$ are incorrect, we deduce that the number of incorrectly colored vertices decreases by a factor of at least two in each iteration, implying that all vertices of H will be correctly colored after $\lceil \log_2 n \rceil$ iterations. This completes the proof of Proposition 2.4. We note that by being more careful one can show that $O(\log_d n)$ iterations suffice here, but since this only slightly decreases the running time, we do not prove the stronger statement here.

A standard probabilistic argument based on the Chernoff bound (see, for example [2, Appendix A]) shows that $H = V$ almost surely if $p \geq \beta \log n/n$, where β is a suitably large constant. Thus, it follows from Proposition 2.4 that the algorithm almost surely properly colors the graph by the end of phase 2 if $p \geq \beta \log n/n$.

For two sets of vertices X and Z, let $e(X, Z)$ denote the number of edges $(u, v) \in E$, with $u \in X$ and $v \in Z$.

LEMMA 2.6. *There exists a constant $\gamma > 0$ such that almost surely the following hold.*

(i) *For any two distinct color classes V_1 and V_2, and any subset X of V_1 and any subset Y of V_2, if $|X| = 2^{-\gamma d}n$ and $|Y| \leq 3|X|$, then $|e(X, V_2 - Y) - d|X|| \leq 0.001d|X|$.*

(ii) *If J is the set of vertices having more than $1.01d$ neighbors in G in some color class, then $|J| \leq 2^{-\gamma d}n$.*

Proof. For any subset X of V_1, $e(X, V_2 - Y)$ is the sum of independent Bernoulli variables. By standard Chernoff bounds, the probability that there exist two color classes V_1 and V_2, a subset X of V_1, and a subset Y of V_2 such that $|X| = \epsilon n$ and $|Y| \leq 3|X|$ and $|e(X, V_2 - Y) - d|X|| > 0.001d|X|$ is at most

$$6\binom{n}{\epsilon n}\sum_{i=0}^{3\epsilon n}\binom{n}{i}2^{-\Omega(\epsilon n d)} = 2^{O(H(\epsilon))n}2^{-\Omega(\epsilon n d)}.$$

Therefore, (i) holds almost surely if γ is a sufficiently small constant. A similar reasoning applies to (ii). Therefore, both (i) and (ii) hold if γ is a sufficiently small constant. □

LEMMA 2.7. *Almost surely, H has at least $(1 - 2^{-\Omega(d)})n$ vertices in every color class.*

Proof. It suffices to show that there are at most $7 \cdot 2^{-\gamma d}n$ vertices outside H. Assume for contradiction that this is not true. Recall that H is obtained first by deleting all the vertices in J and then by a deletion process in which vertices with less than $0.99d$ neighbors in the other color classes of H are deleted repeatedly. By Lemma 2.6 $|J| \leq 2^{-\gamma d}n$ almost surely, and so at least $6 \cdot 2^{-\gamma d}n$ vertices have been deleted because they had less than $0.99d$ neighbors in H in some color class (other than their own). Consider the first time during the deletion process where there exists a subset X of a color class V_i of cardinality $2^{-\gamma d}n$, and a $j \in \{1, 2, 3\} - \{i\}$ such that every vertex of X has been deleted because it had less than $0.99d$ neighbors in the remaining subset of V_j. Let Y be the set of vertices of V_j deleted so far. Then

$|Y| \leq |J| + 2|X| \leq 3|X|$. Note that every vertex in X has less than $0.99d$ neighbors in $V_j - Y$. We therefore obtain a contradiction by applying Lemma 2.6 to (X, Y). □

2.3. The third phase. We need the following lemma, which is an immediate consequence of Lemma 2.5.

LEMMA 2.8. *Almost surely, there exists no subset U of V of size at most $0.001n$ such that the graph induced on U has minimum degree at least $d/2$.*

LEMMA 2.9. *Almost surely, by the end of the uncoloring procedure in phase 3 of the algorithm, all vertices of H remain colored, and all colored vertices are properly colored; i.e., any vertex colored i belongs to W_i. (We assume, of course, that the numbering of the colors is chosen appropriately.)*

Proof. By Proposition 2.4, almost surely all vertices of H are properly colored by the end of phase 2. Since every vertex of H has at least $0.99d$ neighbors (in H) in each color class other than its own, all vertices of H remain colored. Moreover, if a vertex is wrongly colored at the end of the uncoloring procedure, then it has at least $d/2$ wrongly colored neighbors. Assume for contradiction that there exists a wrongly colored vertex at the end of the uncoloring procedure. Then the subgraph induced on the set of wrongly colored vertices has minimum degree at least $d/2$, and hence it must have at least $0.001n$ vertices by Lemma 2.8. However, since it does not intersect H, it has at most $2^{-\Omega(d)}n$ vertices by Lemma 2.7, leading to a contradiction. □

In order to complete the proof of correctness of the algorithm, it remains to show that almost surely every connected component of the graph induced on the set of uncolored vertices is of size at most $\log_3 n$. We prove this fact in the remainder of this section. We note that it is easy to replace the term $\log_3 n$ by $O(\frac{\log_3 n}{d})$, but for our purposes the above estimate suffices. Also note that if $p = o(\log n/n)$ some of these components are actually components of the original graph G, as for such value of p the graph G is almost surely disconnected (and has many isolated vertices).

LEMMA 2.10. *Let K be a graph, (V_1, V_2, V_3) a partition of the vertices of K into three disjoint subsets, i an integer, and L the set of vertices of K that remain after repeatedly deleting the vertices having less than i neighbors in V_1, V_2, or V_3. Then the set L does not depend on the order in which vertices are deleted.*

Proof. Let L be the set of vertices that remain after a deletion process according to a given order. Consider a deletion process according to a different order. Since every vertex in L has at least i neighbors in $L \cap V_1$, $L \cap V_2$, and $L \cap V_3$, no vertex in L will be deleted in the second deletion process (otherwise, we obtain a contradiction by considering the first vertex in L deleted). Therefore, the set of vertices that remain after the second deletion process contains L and thus equals L by symmetry. □

Lemma 2.10 implies that H does not depend on the order in which vertices are deleted.

PROPOSITION 2.11. *Almost surely the largest connected component of the graph induced on $V - H$ has at most $\log_3 n$ vertices.*

Proof. Let T be a fixed tree on $\log_3 n$ vertices of V all of whose edges have their two endpoints in distinct color classes W_i, W_j, $1 \leq i < j \leq 3$. Our objectives are to estimate the probability that G contains T as a subgraph that does not intersect H and to show that this probability is sufficiently small to ensure that, almost surely, the above will not occur for any T. This property would certainly hold if $V - H$ were a random subset of V of cardinality $2^{-\Omega(d)}n$. Indeed, if this were the case, the probability that G contains T as a subgraph that does not intersect H would be upper bounded by the probability $2^{-\Omega(d|T|)}$ that T is a subset of $V - H$ times the probability $(d/n)^{|T|-1}$ that T is a subgraph of G. This bound is sufficiently small for our needs.

Although $V-H$ is *not* a random subset of V, we will be able to show a similar bound on the probability that G contains T as a subgraph that does not intersect H. To simplify the notation, we let T denote the set of edges of the tree. Let $V(T)$ be the set of vertices of T, and let I be the subset of all vertices $v \in V(T)$ whose degree in T is at most 4. Since T contains $|V(T)|-1$ edges, $|I| \geq |V(T)|/2$. Let H' be the subset of V obtained by the following procedure, which resembles that of producing H (but depends on $V(T)-I$). First, set H' to be the set of vertices having at most $1.01d-4$ neighbors in G in each color class V_i. Then delete from H' all vertices of $V(T)-I$. Then, repeatedly delete any vertex in H' having less than $0.99d$ neighbors in H' in some color class (other than its own).

LEMMA 2.12. *Let F be a set of edges, each having endpoints in distinct color classes W_i, W_j. Let $H(F \cup T)$ be the set obtained by replacing E by $F \cup T$ in our definition of H, and $H'(F)$ be the set obtained by replacing E by F in our definition of H'. Then $H'(F) \subseteq H(F \cup T)$.*

Proof. First, we show that the initial value of $H'(F)$, i.e., the value obtained after deleting the vertices with more than $1.01d-4$ neighbors in a color class of G and after deleting the vertices in $V(T)-I$, is a subset of the initial value of $H(F \cup T)$. Indeed, let v be any vertex that does not belong to the initial value of $H(F \cup T)$, i.e., v has more than $1.01d$ neighbors in some color class of $(V, F \cup T)$. We distinguish two cases:

1. $v \in V(T)-I$. In this case, v does not belong to the initial value of $H'(F)$.
2. $v \notin V(T)-I$. Then v is incident with at most 4 edges of T, and so it has more than $1.01d-4$ neighbors in some color class in (V, F).

In both cases, v does not belong to the initial value of $H'(F)$. This implies the assertion of the lemma, since the initial value of $H'(F)$ is a subgraph of the initial value of $H(F \cup T)$ and hence, by Lemma 2.10, any vertex which will be deleted in the deletion process for constructing H will be deleted in the corresponding deletion process for producing H' as well. □

LEMMA 2.13.

$$\begin{aligned} &Pr[T \text{ is a subgraph of } G \text{ and } V(T) \cap H = \emptyset] \\ &\quad \leq Pr[T \text{ is a subgraph of } G]\ Pr[I \cap H' = \emptyset]. \end{aligned}$$

Proof. It suffices to show that

$$\Pr[I \cap H = \emptyset | T \text{ is a subgraph of } G] \leq \Pr\,[I \cap H' = \emptyset].$$

However, by Lemma 2.12

$$\begin{aligned} \Pr[I \cap H' = \emptyset] &= \sum_{F: I \cap H'(F) = \emptyset} \Pr[E(G) = F] \\ &\geq \sum_{F: I \cap H(F \cup T) = \emptyset} \Pr[E(G) = F] \\ &= \sum_{F': F' \cap T = \emptyset, I \cap H(F' \cup T) = \emptyset} \Pr[E(G) - T = F'] \\ &= \sum_{F': F' \cap T = \emptyset, I \cap H(F' \cup T) = \emptyset} \Pr[E(G) - T = F' \mid T \text{ is a subgraph of } G] \\ &= \Pr[I \cap H = \emptyset | T \text{ is a subgraph of } G], \end{aligned}$$

where F ranges over the sets of edges with endpoints in different color classes, and F' ranges over those sets that do not intersect T. The third equation follows by regrouping the edge-sets F according to $F' = F - T$ and noting (the obvious fact) that, for a given set F' that does not intersect T, the probability that $E(G) = F$ for some F such that $F - T = F'$ is equal to $\Pr[E(G) - T = F']$. The fourth equation follows from the independence of the events $E(G) - T = F'$ and T is a subgraph of G. □

Returning to the proof of Proposition 2.11, we first note that we can assume, without loss of generality, that $d \leq \beta \log n$, for some constant $\beta > 0$ (otherwise $H = V$). If this inequality holds then, by modifying the arguments in the proof of Lemma 2.7, one can show that each of the graphs H' (corresponding to the various choices of $V(T)-I$) misses at most $2^{-\Omega(d)}n$ vertices in each color class, with probability at least $1 - 2^{-n^{\Theta(1)}}$. Since the distribution of H' depends only on $V(T) - I$ (assuming the W_i's are fixed), it is not difficult to show that this implies that $\Pr[I \cap H' = \emptyset]$ is at most $2^{-\Omega(d|I|)}$. Since $|I| \geq |V(T)|/2$ and since the probability that T is a subgraph of G is precisely $(d/n)^{|V(T)|-1}$ we conclude, by Lemma 2.13, that the probability of there existing some T of size $\log_3 n$ which is a connected component of the induced subgraph of G on $V - H$ is at most $2^{-\Omega(d \log_3 n)}(d/n)^{\log_3 n - 1}$ multiplied by the number of possible trees of this size, which is

$$\binom{3n}{\log_3 n}(\log_3 n)^{\log_3 n - 2}.$$

Therefore, the required probability is bounded by

$$\binom{3n}{\log_3 n}(\log_3 n)^{\log_3 n - 2}2^{-\Omega(d \log_3 n)}\left(\frac{d}{n}\right)^{\log_3 n - 1} = O(1/n^{\Omega(d)}),$$

completing the proof. □

3. Bounding the eigenvalues. In this section, we prove Proposition 2.1. Let $G = (V, E), A, p, d, \lambda_i, e_i, W_1, W_2, W_3$ be as in section 2. We begin with the following lemma.

LEMMA 3.1. *There exists a constant $\beta > 0$ such that, almost surely, for any subset X of $2^{-\beta d}n$ vertices, $e(X, V) \leq 5d|X|$.*

Proof. As in the proof of Lemma 2.6, the probability that there exists a subset X of cardinality ϵn such that $e(X, V) > 5d|X|$ is at most

$$\binom{3n}{\epsilon n}2^{-\Omega(d\epsilon n)} \leq 2^{3H(\epsilon/3)n}2^{-\Omega(d\epsilon n)} = 2^{-\Omega(d\epsilon n)}$$

if $\log(1/\epsilon) < d/b$, where b is a sufficiently large constant. Therefore, if β is a sufficiently small constant, this probability goes to 0 as n goes to infinity. □

Proof of Proposition 2.1. Parts (i) and (ii) are simple. By the variational definition of eigenvalues (see [19, p. 99]), λ_1 is simply the maximum of $x^t Ax/(x^t x)$ where the maximum is taken over all nonzero vectors x. Therefore, by taking x to be the all-1 vector we obtain the well-known result that λ_1 is at least the average degree of G'. By the known estimates for binomial distributions, the average degree of G is $(1+o(1))2d$. On the other hand, Lemma 3.1 can be used to show that $|E - E'| \leq 2^{-\Omega(d)}n$, as it easily implies that the number of vertices of degree greater than $5d$ in each color class of G is almost surely less than $2^{-\beta d}n$. Hence the average degree of G' is at least $(1 - 2^{-\Omega(d)})2d$. This proves (i).

The proof of (ii) is similar. It is known [19, p. 101] that

$$\lambda_{3n-1} = \min_{F} \max_{x \in F, x \neq 0} \frac{x^t A x}{x^t x},$$

where the minimum is taken over all two-dimensional subspaces F of R^{3n}. Let F denote the two-dimensional subspace of all vectors $x = (x_v : v \in V)$ satisfying $x_v = \alpha_i$ for all $v \in W_i$, where $\alpha_1 + \alpha_2 + \alpha_3 = 0$. For x as above,

$$x^t A x = 2\alpha_1\alpha_2 e'(W_1, W_2) + 2\alpha_2\alpha_3 e'(W_2, W_3) + 2\alpha_1\alpha_3 e'(W_1, W_3),$$

where $e'(W_i, W_j)$ denotes the number of edges of G' between W_i and W_j. Almost surely $e'(W_i, W_j) \geq (1 - 2^{-\Omega(d)})nd$ for all $1 \leq i < j \leq 3$, and since $x^t x = n(\alpha_1^2 + \alpha_2^2 + \alpha_3^2) = -2n(\alpha_1\alpha_2 + \alpha_2\alpha_3 + \alpha_1\alpha_3)$ it follows that $x^t Ax/(x^t x) \leq -(1 - 2^{-\Omega(d)})d$ almost surely for all $x \in F$, implying that $\lambda_{3n} \leq \lambda_{3n-1} \leq -(1 - 2^{-\Omega(d)})d$ and establishing (ii).

The proof of (iii) is more complicated. Its assertion for somewhat bigger p (for example, for $p \geq \log^6 n/n$) can be deduced from the arguments of [10]. To prove it for the graph G' and $p \geq c/n$ we use the basic approach of Friedman, Kahn, and Szemerédi in [11], where they show that the second largest eigenvalue in absolute value of a random d-regular graph is almost surely $O(\sqrt{d})$. (Also see [9] for a different proof.) Since in our case the graph is not regular, a few modifications are needed. Our starting point is again the variational definition of the eigenvalues, from which we will deduce that it suffices to show that, almost surely, the following holds.

LEMMA 3.2. *Let S be the set of all unit vectors $x = (x_v : v \in V)$ for which $\sum_{v \in W_j} x_v = 0$ for $j = 1, 2, 3$, then $|x^t A x| \leq O(\sqrt{d})$ for all $x \in S$.*

The matrix A consists of nine blocks arising from the partition of its rows and columns according to the classes W_j. It is clearly sufficient to show that the contribution of each block to the sum $x^t Ax$ is bounded, in absolute value, by $O(\sqrt{d})$. This, together with a simple argument based on ϵ-nets (see [11, Proposition 2.1]) can be used to show that Lemma 3.2 follows from the following statement.

Fix $\epsilon > 0$, say $\epsilon = 1/2$, and let T denote the set of all vectors x of length n, every coordinate of which is an integral multiple of $\epsilon/\sqrt{n}$, where the sum of coordinates is zero and the l_2-norm is at most 1. Let B be a random n by n matrix with 0, 1 entries, where each entry of B, randomly and independently, is 1 with probability d/n.

LEMMA 3.3. *If d exceeds a sufficiently large absolute constant, then almost surely, $|x^t B y| \leq O(\sqrt{d})$ for every x, $y \in T$ for which $x_u = 0$ if the corresponding row of B has more than $5d$ nonzero entries and for which $y_v = 0$ if the corresponding column of B has more than $5d$ nonzero entries.*

The last lemma is proved, as in [11], by separately bounding the contribution of terms $x_u y_v$ with small absolute values and the contribution of similar terms with large absolute values. Here is a description of the details that differ from those that appear in [11]. Let C denote the set of all pairs (u, v) with $|x_u y_v| \leq \sqrt{d}/n$ and let $X = \sum_{(u,v) \in C} x_u B(u, v) y_v$. As in [11], one can show that the absolute value of the expectation of X is at most $\sqrt{d}$. Next, one has to show that with high probability X does not deviate from its expectation by more than $c\sqrt{d}$. This is different from (and in fact, somewhat easier than) the corresponding result in [11], since here we are dealing with independent random choices. It is convenient to use the following variant of the Chernoff bound.

LEMMA 3.4. *Let $a_1, \ldots, a_m$ be (not necessarily positive) reals, and let Z be the random variable $Z = \sum_{i=1}^{m} \epsilon_i a_i$, where each ϵ_i is chosen, randomly and independently,*

to be 1 *with probability* p *and* 0 *with probability* $1-p$. *Suppose* $\sum_{i=1}^{m} a_i^2 \le D$ *and suppose* $|Sa_i| \le ce^c pD$ *for some positive constants* c, S. *Then* $Pr[|Z - E(Z)| > S] \le 2e^{-S^2/(2pe^c D)}$.

For the proof, one first proves the following.

LEMMA 3.5. *Let* c *be a positive real. Then, for every* $x \le c$,

$$e^x \le 1 + x + \frac{e^c}{2}x^2.$$

Proof. Define $f(x) = 1 + x + \frac{e^c}{2}x^2 - e^x$. Then $f(0) = 0$, $f'(x) = 1 + e^c x - e^x$, and $f''(x) = e^c - e^x$. Therefore, $f''(x) \ge 0$ for all $x \le c$, and as $f'(0) = 0$, this shows that $f'(x) \le 0$ for $x < 0$ and $f'(x) \ge 0$ for $c \ge x > 0$, implying that $f(x)$ is nonincreasing for $x \le 0$ and nondecreasing for $c \ge x \ge 0$. Thus $f(x) \ge 0$ for all $x \le c$, as needed. □

Proof of Lemma 3.4. Define $\lambda = \frac{S}{e^c pD}$; then, by assumption, $\lambda a_i \le c$ for all i. Therefore, by the above lemma,

$$\begin{aligned}
E(e^{\lambda Z}) &= \prod_{i=1}^{m} [pe^{\lambda a_i} + (1-p)] \\
&= \prod_{i=1}^{m} [1 + p(e^{\lambda a_i} - 1)] \\
&\le \prod_{i=1}^{m} \left[1 + p\left(\lambda a_i + \frac{e^c}{2}\lambda^2 a_i^2\right)\right] \\
&\le \exp\left(p\lambda \sum_{i=1}^{m} a_i + p\frac{e^c}{2}\lambda^2 \sum_{i=1}^{m} a_i^2\right) \\
&\le e^{\lambda E(Z) + \frac{pe^c}{2}\lambda^2 D}.
\end{aligned}$$

Therefore,

$$\begin{aligned}
\Pr[(Z - E(Z)) > S] &= \Pr[e^{\lambda(Z - E(Z))} > e^{\lambda S}] \\
&\le e^{-\lambda S} E(e^{\lambda(Z - E(Z))}) \\
&\le e^{-\lambda S + \frac{pe^c}{2}\lambda^2 D} \\
&= e^{-\frac{S^2}{2pe^c D}}.
\end{aligned}$$

Applying the same argument to the random variable defined with respect to the reals $-a_i$, the assertion of the lemma follows. □

Using Lemma 3.4, it is not difficult to deduce that almost surely the contribution of the pairs in C to $|x^t By|$ is $O(\sqrt{d})$. This is because we can simply apply the lemma with $m = |C|$, with the a_i's being all the terms $x_u y_v$ where $(u, v) \in C$, with $p = d/n$, $D = 1$, and $S = ce^c\sqrt{d}$ for some $c > 0$. Since here $|Sa_i| \le ce^c\sqrt{d}\sqrt{d}/n = ce^c pD$, we conclude that for all fixed vectors x and y in T, the probability that X deviates from its expectation (which is $O(\sqrt{d})$) by more than $ce^c\sqrt{d}$ is smaller than $2e^{-c^2 e^c n/2}$, and since the cardinality of T is only b^n for some absolute constant $b = b(\epsilon)$, one can

choose c so that X would almost surely not deviate from its expectation by more than $ce^c\sqrt{d}$.

The contribution of the terms $x_u y_v$, whose absolute values exceed $\sqrt{d}/n$, can be bounded by following the arguments of [11] with a minor modification arising from the fact that the maximum number of ones in a row (or column) of B can exceed d (but can never exceed $5d$ in a row or a column in which the corresponding coordinates x_u or y_v are nonzero). We sketch the argument below, beginning with the following lemma.

LEMMA 3.6. *There exists a constant C such that, with high probability, for any distinct color classes V_1, V_2, and any subset U of V_1 and any subset W of V_2 such that $|U| \le |W|$, at least one of the following two conditions holds:*

1. $e'(U,W) \le 10\mu(U,W)$,
2. $e'(U,W)\log(e'(U,W)/\mu(U,W)) \le C|W|\log(n/|W|)$,

where $e'(U,W)$ is the number of edges in G' between U and W, and $\mu(U,W) = |U|\,|W|d/n$ is the expected number of edges in G between U and W.

Proof. Condition 1 is clearly satisfied if $|W| \ge n/2$, since the maximum degree in G' is at most $5d$. So we can assume without loss of generality that $0 < |W| \le n/2$. Given two subsets U and W satisfying the requirements of the lemma, define $\beta = \beta(|U|,|W|)$ to be the unique positive real number such that $\beta\mu(U,W)\log\beta = C|W|\log(n/|W|)$ (the constant C will be determined later). Condition 2 is equivalent to $e'(U,W) \le \beta\mu(U,W)$. Thus U, W violate condition 1 as well as condition 2 only if $e'(U,W) > \beta'\mu(U,W)$, where $\beta' = \max(10,\beta)$. Hence, by standard Chernoff bounds, the probability of this event is at most $e^{-\gamma\beta'\mu(U,W)\log\beta'} \le (|W|/n)^{\gamma C|W|}$ for some absolute constant $\gamma > 0$. Denoting $|W|/n$ by b, the probability that there exist two subsets U and W that do not satisfy either condition is at most

$$6 \sum_{b:\ bn \text{ integer } \le n/2} \binom{n}{bn}^2 b^{\gamma Cbn} = 6 \sum_{b:\ bn \text{ integer } \le n/2} 2^{O(b\log(1/b)n)} b^{\gamma Cbn}$$

$$= n^{-\Omega(1)}$$

if C is a sufficiently large constant. □

Friedman, Kahn, and Szemerédi [11] show that for any d-regular graph satisfying the conditions of Lemma 3.6 (without restriction on the ranges of U and W), the contribution of the terms $x_u y_v$ whose absolute values exceed $\sqrt{d}/n$ is $O(\sqrt{d})$. Up to replacing some occurrences of d by $5d$, the same proof shows that, for any 3-colorable graph of maximum degree $5d$ satisfying the conditions of Lemma 3.6, the contribution of the terms $x_u y_v$ whose absolute values exceed $\sqrt{d}/n$ is $O(\sqrt{d})$. This implies the assertion of Lemma 3.3, which implies Lemma 3.2.

To deduce part (iii) of Proposition 2.1, we need the following lemma.

LEMMA 3.7. *Let F denote, as before, the two-dimensional subspace of all vectors $x = (x_v : v \in V)$ satisfying $x_v = \alpha_i$ for all $v \in W_i$, where $\alpha_1 + \alpha_2 + \alpha_3 = 0$. Then, almost surely, for all $f \in F$ we have $\|(A + dI)f\|^2 = O(d\|f\|^2)$.*

Proof. Let x, y be as in the proof of Lemma 2.3. Note that $x^t y = 0$ and that both $\|x\|^2$ and $\|y\|^2$ are $\Theta(n)$. Thus, every vector $f \in F$ can be expressed as the sum of two orthogonal vectors x' and y' proportional to x and y, respectively. Lemma 2.3 shows that $\|(A + dI)y'\|^2 = O(d\|y'\|^2)$, which implies that $\|(A + dI)y'\|^2 = O(d\|f\|^2)$, since $\|f\|^2 = \|x'\|^2 + \|y'\|^2$. Similarly, it can be shown that $\|(A + dI)x'\|^2 = O(d\|f\|^2)$. We conclude the proof of the lemma using the triangle inequality $\|(A + dI)f\| \le \|(A + dI)x'\| + \|(A + dI)y'\|$. □

TABLE 1
Implementation results.

Number of vertices	d
1000	12
10000	10
100000	8

We now show that $\lambda_2 \leq O(\sqrt{d})$ by using the formula $\lambda_2 = \min_H \max_{x \in H, x \neq 0} x^t Ax/(x^t x)$, where H ranges over the linear subspaces of R^{3n} of codimension 1. Indeed, let H be the set of vectors whose sum of coordinates is 0. Any $x \in H$ is of the form $f + s$, where $f \in F$ and s is a multiple of a vector in S, and so

$$\begin{aligned} x^t Ax &= f^t Af + 2s^t Af + s^t As \\ &= f^t Af + 2s^t (A + dI)f + s^t As \\ &\leq -(1 - 2^{-\Omega(d)})d\|f\|^2 + 2\|s\| \, \|(A + dI)f\| + O(\sqrt{d})\|s\|^2 \\ &\leq O(\sqrt{d}\|s\| \, \|f\|) + O(\sqrt{d})\|s\|^2 \\ &\leq O(\sqrt{d}(\|s\|^2 + \|f\|^2)) \\ &= O(\sqrt{d}\|x\|^2). \end{aligned}$$

This implies the desired upper bound on λ_2.

The bound $|\lambda_{3n-2}| \leq O(\sqrt{d})$ can be deduced from similar arguments, namely by showing that $x^t Ax/(x^t x) \geq -\Theta(\sqrt{d})$ for any $x \in F^{\perp}$. This completes the proof of Proposition 2.1. □

4. Implementation and experimental results. We have implemented the following tuned version of our algorithm. The first two phases are as described in section 1. In the third phase, we find the minimum i such that, after repeatedly uncoloring every vertex colored j that has less than i neighbors colored l, for some $l \in \{1, 2, 3\} - \{j\}$, the algorithm can find a proper coloring using brute-force exhaustive search on every component of uncolored vertices. If the brute-force search takes more steps than the first phase (up to a multiplicative constant), the algorithm fails. Otherwise, it outputs a legal coloring. The eigenvectors e_{3n} and e_{3n-1} are calculated approximately using an iterative procedure. The coordinates of the initial vectors are independent random variables uniformly chosen in $[0, 1]$.

The range of values of p where the algorithm succeeded was in fact considerably larger than what our analysis predicts. Table 1 shows some values of the parameters for which we tested our algorithm. For each of these parameters, the algorithm was run on more than one hundred graphs drawn from the corresponding distribution and successfully found a proper coloring for all these tests. The running time was less than six minutes on a Sun SPARCstation 2 for the largest graphs. The algorithm failed for some graphs drawn from distributions with smaller integral values of d than the ones in the corresponding row. Note that the number of vertices is not a multiple of 3; the size of one color class exceeds the others by 1.

5. Concluding remarks.

1. There are many heuristic graph algorithms based on spectral techniques but very few rigorous proofs of correctness for any of those in a reasonable model of random graphs. Our main result here provides such an example. Another example is

the algorithm of Boppana [7], who designed an algorithm for graph bisection based on eigenvalues, and showed that it finds the best bisection almost surely in an appropriately defined model of random graphs with a relatively small bisection width. Aspvall and Gilbert [1] gave a heuristic for graph coloring based on eigenvectors of the adjacency matrix, and showed that their heuristic optimally colors complete 3-partite graphs as well as certain other classes of graphs with regular structure.

2. By modifying some of the arguments of section 2, we can show that if p is somewhat bigger ($p \geq \log^3 n/n$ suffices) then almost surely the initial coloring V_i^0 that is computed from the eigenvectors e_{3n-1} and e_{3n} in the first phase of our algorithm is completely correct. In this case the last two phases of the algorithm are not needed. By refining the argument in subsection 2.2, it can also be shown that if $p > 10 \log n/n$, the third phase of the algorithm is not needed, and the coloring obtained by the end of the second phase will almost surely be correct.

3. We can show that a variant of our algorithm finds, almost surely, a proper coloring in the model of random *regular* 3-colorable graphs in which one randomly chooses d perfect matchings between each pair of distinct color classes, when d is a sufficiently large absolute constant. Here, in fact, the proof is simpler, as the smallest two eigenvalues (and their corresponding eigenspaces) are known precisely, as noted in subsection 1.2.

4. The results easily extend to the model in which each vertex first picks a color randomly, independently, and uniformly from among the three possibilities and, next, every pair of vertices of distinct colors becomes an edge with probability $p(> c/n)$.

5. If $G = G_{3n,p,3}$ and $p \leq c/n$ for some small positive constant c, it is not difficult to show that almost surely G does not have any subgraph with minimum degree at least 3, and hence it is easy to 3-color it by a greedy-type (linear-time) algorithm. For values of p that are bigger than this c/n but satisfy $p = o(\log n/n)$, the graph G is almost surely disconnected, and has a unique component of $\Omega(n)$ vertices, which is called the *giant component* in the study of random graphs (see, e.g., [2], [4]). All other components are almost surely sparse, i.e., they contain no subgraph with minimum degree at least 3 and can thus be easily colored in total linear time. Our approach here suffices to find, almost surely, a proper 3-coloring of the giant component (and hence of the whole graph) for all $p \geq c/n$, where c is a sufficiently large absolute constant, and there are possible modifications of it that may even work for all values of p. At the moment, however, we are unable to obtain an algorithm that provably works for all values of p almost surely. Note that, for any constant c, if $p < c/n$ then the greedy algorithm will almost surely color $G_{3n,p,3}$ with a constant number of colors. Thus, our result implies that $G_{3n,p,3}$ can be almost surely colored in polynomial time with a constant number of colors for *all* values of p.

6. Our basic approach easily extends to k-colorable graphs, for every fixed k, as follows. Phases 2 and 3 of the algorithm are essentially the same as in the case $k = 3$. Phase 1 needs to be modified to extract an approximation of the coloring. Let e_i, $i \geq 1$, be an eigenvector of G' corresponding to its ith largest eigenvalue (replace $5d$ by $5kd$ in the definition of G'). Find vectors $x_1, x_2, \ldots, x_{k+1}$ of norm $\sqrt{kn}$ in $\mathrm{Span}(e_1, e_n, e_{n-1}, \ldots, e_{n-k+2})$ such that $x_i = \sqrt{kn} e_i$ and $(x_i, x_j) = -n$ for $1 \leq i < j \leq k+1$. For $2 \leq i \leq n$, and any z, let W_ϵ be the set of vertices whose coordinates in x_i are in $(z - \epsilon, z + \epsilon)$. If, for some i and z, both $|W_{\epsilon_k}|$ and $|W_{\epsilon_k/2}|$ deviate from n by at most $\beta_k n/d$, where ϵ_k and β_k are constants depending on k, color the elements in W_{ϵ_k} with a new color and delete them from the graph. Repeat this

process until the number of vertices left is $O(n/d)$, and color the remaining vertices arbitrarily.

7. The existence of an approximation algorithm based on the spectral method for coloring *arbitrary* graphs is a question that deserves further investigation (and which we do *not* address here). Recently, improved approximation algorithms for graph coloring have been obtained using semidefinite programming [14], [5].

Acknowledgment. We thank two anonymous referees for several suggestions which improved the presentation of the paper.

REFERENCES

[1] B. Aspvall and J. R. Gilbert, *Graph coloring using eigenvalue decomposition*, SIAM J. Alg. Discrete Meth., 5 (1984), pp. 526–538.
[2] N. Alon and J. H. Spencer, *The Probabilistic Method*, John Wiley, New York, 1991.
[3] A. Blum, *Some tools for approximate* 3*-coloring*, in Proc. 31st IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 554–562.
[4] B. Bollobás, *Random Graphs*, Academic Press, New York, 1985.
[5] A. Blum and D. Karger, *An* $\tilde{O}(n^{3/14})$*-coloring algorithm for* 3*-colorable graphs*, Inform. Process. Lett., 61 (1997), pp. 49–53.
[6] A. Blum and J. H. Spencer, *Coloring random and semi-random k-colorable graphs*, J. Algorithms, 19 (1995), pp. 204–234.
[7] R. Boppana, *Eigenvalues and graph bisection: An average case analysis*, in Proc. 28th IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 280–285.
[8] M. E. Dyer and A. M. Frieze, *The solution of some random NP-hard problems in polynomial expected time*, J. Algorithms, 10 (1989), pp. 451–489.
[9] J. Friedman, *On the second eigenvalue and random walks in random d-regular graphs*, Combinatorica, 11 (1991), pp. 331–362.
[10] Z. Füredi and J. Komlós, *The eigenvalues of random symmetric matrices*, Combinatorica, 1 (1981), pp. 233–241.
[11] J. Friedman, J. Kahn, and E. Szemerédi, *On the second eigenvalue in random regular graphs*, in Proc. 21st ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 587–598.
[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, 1979.
[13] R. Karp, *Reducibility Among Combinatorial Problems*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972.
[14] D. Karger, R. Motwani, and M. Sudan, *Approximate graph coloring by semidefinite programming*, in Proc. 35th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 2–13.
[15] L. Kucera, *Expected behavior of graph colouring algorithms*, in Lecture Notes Comput. Sci. 56, Springer-Verlag, New York, 1977, pp. 447–451.
[16] A. D. Petford and D. J. A. Welsh, *A randomised* 3*-colouring algorithm*, Discrete Math., 74 (1989), pp. 253–261.
[17] A. Ralston, *A First Course in Numerical Analysis*, McGraw-Hill, New York, 1985, Section 10.4.
[18] J. S. Turner, *Almost all k-colorable graphs are easy to color*, J. Algorithms, 9 (1988), pp. 63–82.
[19] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1749–1763, December 1997

010

A FAST ALGORITHM FOR THE COMPUTATION AND ENUMERATION OF PERFECT PHYLOGENIES*

SAMPATH KANNAN† AND TANDY WARNOW†

Abstract. The perfect phylogeny problem is a classical problem in computational evolutionary biology, in which a set of species/taxa is described by a set of qualitative characters. In recent years, the problem has been shown to be *NP-complete* in general, while the different fixed parameter versions can each be solved in polynomial time. In particular, Agarwala and Fernández-Baca have developed an $O(2^{3r}(nk^3+k^4))$ algorithm for the perfect phylogeny problem for n species defined by k r-state characters [*SIAM J. Comput.*, 23 (1994), pp. 1216–1224]. Since, commonly, the character data are drawn from alignments of molecular sequences, k is the length of the sequences and can thus be very large (in the hundreds or thousands). Thus, it is imperative to develop algorithms which run efficiently for large values of k. In this paper we make additional observations about the structure of the problem and produce an algorithm for the problem that runs in time $O(2^{2r}k^2n)$. We also show how it is possible to efficiently build a structure that implicitly represents the set of all perfect phylogenies and to randomly sample from that set.

Key words. evolutionary trees, perfect phylogeny, combinatorial enumeration, dynamic programming, polynomial delay algorithms

AMS subject classifications. 05C05, 05C30, 05C85, 68Q20

PII. S0097539794279067

1. Introduction.

A fundamental problem in biology is that of inferring the evolutionary history of a set S of species, each of which is specified by the set of *traits* or *characters* that it exhibits. Information about evolutionary history can be conveniently represented by an evolutionary or *phylogenetic* tree, often referred to simply as a *phylogeny*. In one of the standard models, the problem can be expressed mathematically as follows. Let $\mathcal{C}$ denote the characters defining the species set $\mathcal{S}$, and $\mathcal{A}_c$ the set of states of character c. Let $|\mathcal{C}| = k$ and $\max_c|\mathcal{A}_c| = r$. For ease of discussion we will assume that $\mathcal{A}_c \in \{1, 2, \ldots, r\}$, so that each species is identified with a vector in Z^k (where Z denotes the integers) and $c(\boldsymbol{s})$ is referred to as the *state of character c for $\boldsymbol{s}$* or the *state of $\boldsymbol{s}$ on character c*.

The input to the perfect phylogeny problem is therefore an $n \times k$ matrix M, where $M_{i,j}$ is the state of the ith species on the jth character. The *perfect phylogeny problem* is to determine whether a given set of n distinct species $\mathcal{S}$ has a tree T with the following properties:

(C1) $\mathcal{S} \subseteq V(T) \subseteq Z^k$;
(C2) every leaf in T is in $\mathcal{S}$; and
(C3) for every $c \in \mathcal{C}$ and every $j \in \mathcal{A}_c$, the set of all $\boldsymbol{u} \in V(T)$ such that $c(u) = j$ induces a subtree of T.

The tree T, if it exists, is called a *perfect phylogeny* for $\mathcal{S}$, and the set of characters $\mathcal{C}$ is said to be *compatible*. In the biology literature, the perfect phylogeny problem is more commonly known as the *character compatibility problem* [8, 9, 14, 15], where it was introduced in the 1950s.

* Received by the editors December 27, 1994; accepted for publication (in revised form) December 6, 1995.
http://www.siam.org/journals/sicomp/26-6/27906.html
† Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 (kannan@central.cis.upenn.edu, tandy@central.cis.upenn.edu). The first author was supported in part by NSF grant CCR-9108969. The second author was supported in part by ARO grant DAAL03-89-0031PRI.

The perfect phylogeny problem was shown to be NP-complete by Bodlaender, Fellows, and Warnow [3] and, independently, by Steel [18]. This fact suggests at least two lines of attack: one is to restrict k, the number of characters; the other is to restrict r.

The perfect phylogeny problem was shown to reduce to a graph-theoretic problem in [5] called the *triangulating colored graphs problem*. The number of characters in an instance I equals the number of colors in the corresponding graph G_I. Algorithms for the perfect phylogeny problem based upon the graph-theoretic formulation were given in [4, 12, 16]. These algorithms are thus useful when the number of characters in the input can be bounded.

In this paper, we pursue the second approach, in which the number of states is bounded (as is the case for molecular data). Gusfield [11] gave an $O(nk)$ algorithm for $r = 2$, the *binary* character case. Dress and Steel [7] devised an $O(nk^2)$ algorithm for $r \leq 3$. Kannan and Warnow [13] gave an $O(n^2k)$ algorithm for $r \leq 4$ (*quaternary* characters). Agarwala and Fernández-Baca [1] give an $O(2^{3r}(nk^3 + k^4))$ algorithm for the perfect phylogeny problem when the characters are restricted to having at most r states.

We present an $O(2^{2r}nk^2)$ time algorithm for this problem. The algorithm we present uses some of the same structure as that of [1] but uses additional lemmas about the problem structure and different data structures to achieve the faster running time.

Biomolecular data (such as DNA, RNA, or amino-acid sequences) define characters for which the maximum number of states is bounded by a small constant ($r = 4$ for DNA or RNA, and $r = 20$ for amino-acid sequences). As a consequence, efficient algorithms for the perfect phylogeny problem for fixed numbers of states are of particular importance in molecular phylogenies.

In most applications of computation in biology, it is not sufficient to know a single optimal solution. Instead it is important to know the entire space of optimal solutions (such information could be used, for example, to settle the *African Eve hypothesis*). In the second half of this paper, we present algorithms that allow us to construct a representation of this space. This representation can be used to count the number of optimal solutions, enumerate all minimal optimal solutions, generate random optimal solutions, and gather aggregate statistics about the set of optimal solutions.

2. Preliminaries. We state some definitions and briefly summarize the algorithm of Agarwala and Fernández-Baca since our algorithm is a modification of theirs. Several of these definitions appear in [1], and equivalent definitions to some of these definitions can also be found in [13].

In the remainder of this section, we will let α denote an arbitrary character in C, and we will refer to the states of α by α-states. We will use $*$ to denote a dummy state for a character.

DEFINITION 2.1. *Suppose T is a perfect phylogeny and let p be some vertex in T. We shall say that the α-state of p is* forced *if p lies on the path between leaves u and v of T with $\alpha(u) = \alpha(v)$.*

DEFINITION 2.2. *A subset $S' \subset S$ is called a* cluster *if for every character α, if $\exists\{x, y\} \subseteq S', \{x', y'\} \subseteq S - S'$ such that $\alpha(x) = \alpha(x')$ and $\alpha(y) = \alpha(y')$, then $\alpha(x) = \alpha(x')$ (i.e., at most one α-state is "shared" by S' and $S - S'$).*

DEFINITION 2.3. *A cluster S' is called a* proper cluster *if there is some character α for which S' does not share any α-state with $S - S'$.*

In the subsequent definitions, we will be interested in defining the character states

of internal nodes according to a canonical labeling; for this purpose, we will use $*$ to indicate the dummy state of a character.

DEFINITION 2.4. *Let G be a cluster. We will denote by $Sv(G)$ the unique vector $v \in (Z \cup \{*\})^k$ such that $\forall \alpha \in C$, if $\exists a \in G, b \in S - G, \alpha(a) = \alpha(b)$, then $\alpha(v) = \alpha(a)$, and otherwise $\alpha(v) = *$. $Sv(G)$ is called the* splitting vector of G.

DEFINITION 2.5. *If $G_1 \subset G$ and both G and G_1 are clusters, then G_1 is said to be* compatible *with G if for all characters for which both $Sv(G)$ and $Sv(G_1)$ are set to nondummy states, they are set to the same state. We will also say that G_1 is compatible with the vector $Sv(G)$ in this case.*

The definitions of splitting vectors and compatibility can be extended as follows.

DEFINITION 2.6. *If G, G_1 are proper clusters and $G_1 \subset G$ is compatible with G, then the splitting vector $Sv(G, G_1)$ is defined as follows: for any character α having a shared state i between G and $S - G$ or between G_1 and $S - G_1$, $\alpha(Sv(G, G_1)) = i$.*

Since G_1 is compatible with G, there will be no conflict in setting the states of $Sv(G, G_1)$.

$Sv(G, G_1)$ is obtained very simply from $Sv(G)$ and $Sv(G_1)$. For any character on which either of the latter two vectors has a nondummy state, $Sv(G, G_1)$ is also set to that state. Otherwise $Sv(G, G_1)$ has the dummy state. We denote this by saying that $Sv(G, G_1) = Sv(G) \oplus Sv(G_1)$.

This extension is motivated by the following scenario. If v is an internal node of degree at least three in a perfect phylogeny T, then the removal of v splits T into three or more components. The definition of $Sv(G, G_1)$ gives a labeling for a node v in a perfect phylogeny (if such a node exists) whose removal will split T into components, some of which union to G_1 and others of which union to $G - G_1$. Note that, in fact, in an *actual* partition into components some of the dummy states of v might be forced. To see this, consider the case where $G - G_1$ partitions into $G_{11}, G_{12}, \ldots, G_{1t}$, where G_{1i} and G_{1j} share an α-state i although no α-state is shared between any two of $S - G, G - G_1$, and G_1; in this situation, the α-state for the splitting vector $Sv(G, G_1)$ is the dummy state, but the actual setting for the node v rooting the subtree defined by G is i for this particular partition.

DEFINITION 2.7. *A perfect phylogeny P for S is* minimal *if for all edges $e \in E(P)$, the topology which results by contracting the edge e in P cannot be vertex-labeled to make a perfect phylogeny.*

Clearly, a set S has a perfect phylogeny if and only if it has a minimal perfect phylogeny.

The relevance of proper clusters to the perfect phylogeny problem and to the algorithm of [1] is shown in the following lemma.

LEMMA 2.8 (see [1, Lemma 1]). *Let T be a minimal perfect phylogeny for S and let T' be an arbitrary subtree of T obtained as a component by the removal of an edge $e = (u, v)$ of T. Let S' be the subset of species from S labeling nodes of T'. Then S' is a proper cluster.*

Proof. S' must be a cluster, since at most one state of each character can be shared across the edge e. Furthermore, if S' is not a proper cluster, then both u and v must be identically labeled (since on each character their state is forced to be the shared state between S' and $S - S'$). Thus an equivalent "reduced" perfect phylogeny could be obtained by collapsing edge e and identifying nodes u and v. Since T is presumed to be a minimal perfect phylogeny, it follows that for any subtree T' as above, the subset S' of S labeling internal nodes of T' forms a proper cluster. □

We need one more definition.

DEFINITION 2.9. *A proper cluster G is* good *if the set $G \cup \{Sv(G)\}$ has a perfect phylogeny. A pair of proper clusters (G, G_1) for which $G_1 \subset G$ and G_1 is compatible with G is* good *if there exists a perfect phylogeny for G with an internal node v labeled by $Sv(G, G_1)$ such that the removal of v partitions G into subsets (defined by the species from S labeling nodes of each component tree) some of which union to G_1.*

Using this idea, the algorithm of [1] works, in a bottom-up dynamic programming fashion, to determine which proper clusters are good.

In [1] it is shown that if G_1 is a good proper cluster and G is another proper cluster containing G_1 and compatible with G_1, then we can determine (in polynomial time) if (G, G_1) is good.

3. The algorithm of Agarwala and Fernández-Baca. Before applying any of the dynamic programming techniques they develop, Agarwala and Fernández-Baca first determine if any of the species in S can occupy an internal node in the perfect phylogeny they construct. This can be determined in polynomial time (in fact, in $O(nk)$ time, as we will show later). If any node can be an internal node, they reduce to subproblems. The more difficult case is where no species in S can be an internal node in any perfect phylogeny for S.

The algorithm in [1] determines, for all proper clusters G' of smaller cardinality than G, whether G' is good. Now, in considering the pair of proper clusters (G, G_1), there are two cases. If $G_2 = G - G_1$ is also a proper cluster, then (G, G_1) is good if and only if G_2 is also good, and this can be determined by a simple table look-up. If on the other hand, G_2 is not a proper cluster, then $Sv(G, G_1)$ is forced for all characters. The algorithm then loops over all possible good proper clusters $G' \subset G_2$ that are compatible with $Sv(G, G_1)$. Then (G, G_1) is good if and only if G_2 is the disjoint union of a set of good proper clusters $G' \subseteq G_2$.

The determination of whether a proper cluster G is good requires finding a proper cluster G_1 which is contained in and compatible with G, such that (G, G_1) is good. In [1], this determination is accomplished by checking all potential proper clusters G_1. For each pair of proper clusters (G, G_1), the algorithm may spend $2^r k$ time going over all possible proper clusters G_2 and for each of these clusters spends $O(n + k)$ time determining whether G_2 is compatible with $Sv(G, G_1)$. Thus the overall running time of the algorithm in [1] is $O(2^{3r} k^3 (k + n))$.

4. Our improvements to the algorithm. While most of the innovation in our algorithm occurs by exploiting structural properties of the solution so as to reduce the number of loops as well as to speed up the inner loop, in order to achieve an overall cost of $O(2^{2r} nk^2)$ time, we will also do some preprocessing of the data.

4.1. Preprocessing. Given a vector $s \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_m$, we can define an equivalence relation E_s on S as the transitive closure of the following relation R: for $a, b \in S$, $(a, b) \in R$ if $\exists c \in C$ s.t. $c(a) = c(b) \neq c(s) \neq *$. It is clear from this definition that two species in S which are in the same equivalence class must be in the same component of $T - \{s\}$ for any perfect phylogeny on $S \cup \{s\}$. We denote the operation of computing the equivalence classes by S/s.

We assume that the species are given in some fixed order $s_1, s_2, \ldots, s_n$. As in the Agarwala and Fernández-Baca algorithm, we first determine whether any $s \in S$ can be internal nodes, and if so reduce to subproblems. The additional preprocessing of the data we perform involves the following steps:

1. Compute all proper clusters.
2. Sort the proper clusters lexicographically and store them in a table.

3. Viewing each proper cluster as a lexicographically ordered sequence of species, build a *trie* for the proper clusters which is pruned so that each internal node has at least one proper cluster that lies below it. Also, any node of the trie that represents a proper cluster will have a pointer to that proper cluster in the table.
4. For every proper cluster G compute $Sv(G)$.
5. For each vector $Sv(G)$ compute $S/Sv(G)$.

Running time.

Step 1. Each proper cluster partitions the species set S by partitioning the states of some character. Let L be the number of proper clusters. There are $O(2^r)$ proper clusters for each character, and hence only $O(2^r k)$ proper clusters in all; thus L is $O(2^r k)$. Generating the set of clusters involves generating all possible subsets of character states and determining which ones are clusters. Generating the subsets takes $O(n)$ time per subset. To check whether a subset A is a cluster we list each state of each character which appears in A. This takes $O(nk)$ time. We then compare the list for A and $S - A$ to check that for each character, at most one state is shared between A and $S - A$. This takes $O(kr)$ time, which is $O(nk)$ since $r \leq n$. Hence the total cost for generating all the clusters is $O(nk)$ per cluster, and so this step uses $O(2^r k^2 n)$ time.

Step 2. Sorting the proper clusters lexicographically (using radix sort) costs $O(2^r kn)$.

Step 3. Note that since each leaf of the trie is a proper cluster and since the depth of the trie is at most n, there are at most $O(2^r kn)$ nodes in the trie. Also, a simple traversal of the sorted list of proper clusters builds the trie in $O(2^r kn)$ steps.

Step 4. The computation of $Sv(G)$ can be done in $O(nk)$ steps. To do this, for each character α build a table of size r, and go through the species in S noting (in the table) which α-states are present in species in G and which α-states are present in the species in $S - G$. (For instance, table entries could be initialized to 0, set to 1 if a species in G has the corresponding state, set to 2 if a species in $S - G$ has that state, and set to 3 if there are species in both sets having that state.) Now, we simply check if there is any table entry which is a "3" and simply set $Sv(G)$ to that state. Otherwise $Sv(G)$ is set to "$*$." Since we do $O(nk)$ work per proper cluster, this costs us $O(2^r nk^2)$ time for Step 4.

Step 5. We need to compute $S/Sv(G)$ in this step. We will show that for any vector x we can compute S/x in $O(nk)$ time.

LEMMA 4.1. *For any vector x, S/x can be computed in $O(nk)$ time.*

Proof. A straightforward implementation using UNION-FIND appears to take $O(nk\alpha(nk, n))$ time. We give a constructive proof of the $O(nk)$ bound. For each character, α such that $\alpha(x)$ is not the dummy state, we maintain information about the equivalence class to which each state of α other than $\alpha(x)$ "belongs." (A state of α belongs to a class if there is some species in that class having the specified α-state.) Note that, since two species which share an α-state other than $\alpha(x)$ must be in the same equivalence class, this is well defined.

Now we process each species in turn taking $O(k)$ time per species. When we look at species s, for each character α we look to see if $\alpha(s)$ belongs to some equivalence class. If so, we include s in that class and, for every character β on which $\beta(x)$ is not the dummy state, if $\beta(s) \neq \beta(x)$, we include $\beta(s)$ as a β-state of that class. If none of the states of s belong to existing classes, we create a new class with s as the only species in that class and we initialize the states of this class appropriately. Also, if s belongs to more than one class, we union these classes and associate with the new

class the union of the set of states associated with each of the old classes. Since there can be at most n unions and each union takes only $O(k)$ time, the total cost for all unions is at most $O(nk)$. The other costs are at most $O(k)$ per species and these also result in a total cost of $O(nk)$. □

The computation of Step 5 costs us a total of $O(2^r nk^2)$. We will represent the equivalence classes of $S/Sv(G)$ by a spanning forest $T_{Sv(G)}$ on S, so that the components of $T_{Sv(G)}$ are the distinct equivalence classes of $S/Sv(G)$.

Overall cost. Thus the total cost of the preprocessing is $O(2^r nk^2)$.

4.2. Reducing to two nested loops. The algorithm of [1] has three nested loops, each of which steps through all possible proper clusters in the worst case. Thus the number of executions of the innermost loop becomes $O(2^{3r}k^3)$. The three loops arise because for all possible proper clusters G, and for all possible proper clusters G_1 that are subsets of G, and for all possible proper clusters which are subsets of $G_2 = G - G_1$ we execute the innermost loop. Since the cost of the innermost loop is bounded by $O(n+k)$, the overall cost of the [1] algorithm is therefore $O(2^{3r}(nk^3+k^4))$.

In this section we will show that we can reduce to only two nested loops, and bring the cost of the innermost loop to $O(n)$ through extensive preprocessing, thus bringing the entire cost of the algorithm to $O(2^{2r}nk^2)$.

We begin with some lemmas.

LEMMA 4.2. *Let s be a vector such that none of its states are equal to the dummy state. If s occurs as the label of an internal node in a minimal perfect phylogeny for S and Q is an equivalence class of S/s, then Q is a proper cluster.*

Proof. Let T be a minimal perfect phylogeny for S where s occurs as the label of an internal node v. By a result in [2], Q is entirely contained in one of the subtrees of $T - \{v\}$. Let T_Q be this subtree. Without loss of generality, we may assume that the leaf-set of T_Q is exactly Q since if T_Q contains leaves outside Q, these leaves can be moved to a different component of $T - \{v\}$. Now, v is adjacent to a single node y in T_Q. Since T is minimal, we cannot contract the edge (v, y), so that for some character α, $\alpha(v)$ and $\alpha(y)$ are both forced and different. However, then Q is a proper cluster, since no state of α crosses the boundary of Q. □

THEOREM 4.3. *Suppose G is a proper cluster, $G_1 \subset G$ a good proper cluster, and $G - G_1 = G_2$ not a proper cluster. Let $x = Sv(G, G_1)$. Then (G, G_1) is good if and only if each $Q \in G_2/x$ is a good proper cluster.*

Proof. Note that for each $Q \in G_2/x$, the splitting vector, $Sv(Q)$ is compatible with x. Thus if each $Q \in G_2/x$ is good, then the perfect phylogenies for each of these components, as well as the perfect phylogeny for G_1, may be made to hang off the node labeled x.

Conversely, suppose that (G, G_1) is good, and let T be a minimal perfect phylogeny for S in which the species in G define an edge-removal-induced subtree T_G and G_1 is the union of the leaves in a collection of subtrees off the root of T_G (such a tree exists by the fact that (G, G_1) is good). The canonical labeling for the root of T_G is x. Since $G - G_1$ is not a proper cluster, x is fully defined (by a lemma proved in [1]) i.e., none of its states are dummy states. By Lemma 4.2, each equivalence class Q of S/x is a proper cluster. It is easy to verify that each $Q \in S/x$ is also good and that $G_2/x \subset S/x$. □

As a consequence of these lemmas, we can reduce the number of loops from three to two.

Reducing the number of loops: Suppose G is a proper cluster, $G_1 \subset G$ is a good proper cluster, and $G_2 = G - G_1$ is not a proper cluster.

Let $x = Sv(G, G_1)$, and compute the equivalence classes Q of G_2/x.

Report *YES* for G, G_1 if and only if each of these equivalence classes is a good proper cluster.

Else return *NO*.

4.3. Speeding up the inner loop. In the process of going from three nested loops to two, we have increased the time of execution of the inner loop from $O(n+k)$ in the algorithm of [1] to $O(nk)$. We can avoid this blow-up using the information computed during preprocessing. In fact, we will reduce the time for the inner loop to $O(n)$. Since k can be $O(n^{r-1})$, this is significant. To see this, let T be a tree with n leaves and $2n-3$ edges; each choice of $r-1$ edges separates the tree into r components, and thus defines a unique r-state character. Thus, we can define $\Omega(n^{r-1})$ distinct characters that are consistent with a perfect phylogeny. Furthermore, most data sets these days use characters derived from biomolecular sequences in which the number of characters equals the length of the sequences in a multiple alignment, and this number can be quite large (and significantly larger than the number of species). Reducing the dependence on k here is important.

Consider how we execute the inner loop. We are given a pair of proper clusters G, G_1, and $G_1 \subset G$. We need to see if (G, G_1) is good (that is, if G has a perfect phylogeny rooted at $Sv(G, G_1)$ in which G_1 is the union of some of the subtrees).

Condition 1. A necessary condition for (G, G_1) to be good is that $Sv(G_1)$ and $Sv(G)$ are compatible. If that condition holds, we then compute $G_2 = G - G_1$. Note that this ensures that G_2 is a cluster.

Condition 2. If G_2 is a proper cluster, then (G, G_1) is good if and only if G_2 is good.

Condition 3. If G_2 is not a proper cluster, then (G, G_1) is good if and only if we can write G_2 as a union of proper clusters $G_{11}, G_{12}, \ldots, G_{1q}$, such that for each G_{1i}, $Sv(G_{1i})$ is compatible with $Sv(G)$ and G_{1i} is good.

LEMMA 4.4. *Let G and $G_1 \subset G$ be clusters. Let $x_1 = Sv(G_1)$ and $x = Sv(G)$. Then x_1 and x are compatible iff there is no edge in T_x between a node in G_1 and a node not in G_1.*

Proof. Suppose that x and x_1 are compatible but that in T_x there is an edge from vertex $u \in G_1$ to vertex $v \notin G_1$. Then there exist $u' \in G_1$ and $v' \notin G_1$ and a character c such that $c(u') = c(v') \neq c(x)$. Since x_1 is defined as the splitting vector of G_1 it follows that $c(x_1) = c(u') \neq c(x)$, contradicting the assumption that x_1 and x are compatible.

Conversely, suppose that x and x_1 are not compatible. Then there is a character c for which $c(x)$ and $c(x_1)$ are set to different states. Since $c(x_1)$ is set, there must be a $u \in G_1$ and a $v \notin G_1$ such that $c(u) = c(x_1) = c(v)$. Since $c(x) \neq c(x_1)$, u and v must be in the same component of T_x. □

Thus the check of Condition 1 takes $O(n)$ time. If Condition 1 holds, we can determine in $O(n)$ time whether G_2 is a proper cluster. If so, we perform a table look-up to see if G_2 is good. Thus, Condition 2 takes only $O(n)$ time to check. Condition 3, surprisingly, also takes only $O(n)$ time (instead of the $O(2^r k(n+k))$ time used in [1]).

LEMMA 4.5. *We can determine whether Condition* 3 *holds in $O(n)$ time.*

Proof. If $G - G_1 = G_2$ is not a proper cluster, then by Theorem 4.3, (G, G_1) is good if and only if each of the equivalence classes of G_2/y is a good proper cluster, where $y = Sv(G, G_1)$.

Let $Sv(G) = x$ and $Sv(G_1) = x_1$. Recall that $y = x \oplus x_1$. Since G_2/x and G_2/x_1 are already known, G_2/y can be computed as follows.

We start with the graphs T_x and T_{x_1}, and we add an edge between each pair of corresponding vertices. We then need to compute the connected components. This takes $O(n)$ time, since the number of edges in each graph is linear. These connected components are the equivalence classes of S/y. Each component is a proper cluster by Lemma 4.2.

To see if each of the components is a good proper cluster, all we need to do is a set of table look-ups. However, there is a subtle point here. We have only $O(n)$ time to perform perhaps as many as $O(n)$ look-ups. To accomplish this we will exploit the fact that there is a trie representing the proper clusters.

When we determine the partition of G_2 into equivalence classes, we will number these classes and determine for each $s \in G_2$ the number of the class to which it belongs.

After this, we go through the species contained in G_2 in lexicographic order. If species s is contained in a class l, we can use this information to narrow down the range of possible indices at which class l may occur in the table of all proper clusters. Thus, only a constant amount of work is performed per species in G_2. At the end it is determined whether each of the classes is a proper cluster, in which case an index is found in the table to each of these proper clusters. It is then easy to perform table look-up to determine which of these proper clusters are good. We can therefore determine whether G has a phylogeny with a subtree T_1 for G_1 hanging off the root x of G. Clearly all of this is accomplished in $O(n)$ time. □

4.4. Analysis of running time. The preprocessing takes $O(2^r nk^2)$. Then we process each proper cluster in turn. For each of the possible pairs (G, G_1) of proper clusters where $G_1 \subset G$ and G_1 is good, we use only $O(n)$ time, for a total cost of $O(2^{2r} nk^2)$ time. The algorithm therefore runs in $O(2^{2r} nk^2)$.

5. Enumerating all minimal perfect phylogenies. In this section we consider the problem of enumerating all minimal perfect phylogenies. We are interested only in minimal perfect phylogenies for two reasons. The first rather banal reason is that there can be many more perfect phylogenies than minimal ones (each minimal perfect phylogeny which has nodes of degree greater than three can yield many different perfect phylogenies by refinement) and since enumeration is costly, it's better if we can just enumerate the minimal ones. The second reason is more significant. Since we wish to determine the evolutionary relationships that are supported by the data, it is in fact the minimal trees which are significant. Since evolutionary hypotheses are usually expressed by saying that speciation events occurred in a specific order, if we resolve a node of degree greater than three, we are (falsely) indicating support for a hypothesis that the minimal tree does not provide. Thus, it behooves us to consider only the minimal perfect phylogenies.

The key issues for efficient enumeration are to avoid following "dead ends" in the dynamic programming algorithm, to ensure that we never enumerate anything twice and that the enumeration is exhaustive.

To simplify our presentation, we will describe only the enumeration of minimal perfect phylogenies where all the species in S are forced to occur as leaves. Slight modifications of the method allow it to be extended to enumerating other minimal perfect phylogenies as well.

We will say that a proper cluster H has a *subphylogeny* if there is a perfect phylogeny P for S with an edge e such that the removal of e partitions the leaves of

P into H and $S - H$. The *root* of the subphylogeny for H is that endpoint of e on whose side the leaf set is H. Note that H being good does not ensure that H has a subphylogeny because there is also a constraint on $S - H$. We will call a proper cluster H *very good* if it has a subphylogeny. This definition easily extends to pairs. A pair of proper clusters (H, G) is very good if and only if (H, G) is good and H has a subphylogeny. It is clear that, after running the algorithm described in the previous section, one more pass through the data structures built up by the algorithm suffices to determine all very good pairs.

Throughout this section let H, G, and G_1 be proper clusters with $G_1 \subset G \subset H$ and such that (H, G) and (G, G_1) are each *very* good pairs.

Since a subphylogeny is, in particular, a perfect phylogeny for a subset of S, we can talk about minimal subphylogenies. We are interested in identifying very good pairs (H, G) such that H has a *minimal* subphylogeny P rooted at r_H and one of the children of r_H is in turn the root, r_G, of a (minimal) subphylogeny for G. We will refer to such minimal subphylogenies by $\mathrm{Min}(H, G)$, while the set of subphylogenies (i.e., not necessarily minimal) of H with this form is indicated by $\mathrm{Sub}(H, G)$.

We now introduce two definitions which are minor variants of previously defined terms. Given a topology T for a perfect phylogeny, we define a canonical labeling of the internal nodes as follows.

DEFINITION 5.1. *Let v be an internal node. Without loss of generality, the degree of v is greater than or equal to 3. Let $T_1(v), T_2(v), \ldots, T_k(v)$ be the subtrees of T obtained by the removal of v. For any character α we define $\alpha(v)$ to be the unique α-state shared by two or more of these subtrees, if such a state exists. Otherwise $\alpha(v)$ is set to equal a dummy state "$*$." The vector so defined is called the* canonical labeling of *v, and we denote this by $cl(v)$.*

We also need to define the notion of *compatibility* in more general terms.

DEFINITION 5.2. *Two vectors x and y in Z^k are* compatible *if for every character α, if $\alpha(x)$ and $\alpha(y)$ are both fixed to nondummy states, then $\alpha(x) = \alpha(y)$.*

With the above definitions, a minimal topology can be thought of as follows.

A topology T (which is perfect) is minimal if the canonical labeling of T does not produce two adjacent nodes whose labels are compatible.

We make the following additional definitions.

DEFINITION 5.3. *The set $Ext(H, G)$ consists of all vectors x such that*

- *$\exists P \in Sub(H, G)$ such that $x = cl(r_H)$;*
- *the leaf set of each subtree of the root r_H is a proper cluster.*

Note that the vector x must have the same state as $Sv(H, G)$ on all characters on which $Sv(H, G)$ has a nondummy state (i.e., x is an extension of $Sv(H, G)$).

LEMMA 5.4. *For H and G as defined, $|Ext(H, G)| \leq 2^{r^2} k^r$.*

Proof. If the root of H has r or more children and each of these children roots a subtree whose leaf set is a proper cluster, then for every character α, a state of α must be shared either between two children of the root or between H and $S - H$. In either case the α-state of the root would be determined. Thus, any vector in the extension must arise from the choice of a set of no more than $r - 1$ proper clusters, and the bound follows. □

DEFINITION 5.5. *Let H be a proper cluster. The set $Conn(H)$ consists of those vectors x such that*

- *$\exists G$ such that (H, G) is very good, and*
- *$\exists P \in \mathrm{Min}(H, G)$ such that $x = cl(r_H)$.*

Note that $Conn(H) \subseteq \cup_{G:(H,G)\mathrm{verygood}} Ext(H, G)$. The reason that $Conn(H)$

may not be identically equal to this union is that $Conn(H)$ consists of canonical labelings of roots in *minimal* perfect phylogenies for H, while $\cup_G Ext(H,G)$ is the set of canonical labelings for roots of perfect phylogenies which may or may not be minimal.

Since we are concerned with enumerating minimal phylogenies we want to avoid listing phylogenies containing edges whose endpoints are labeled compatibly. Consequently, given a very good pair (H,G), we need to ensure that the edge $e = (r_H, r_G)$ cannot be contracted. This will be true if $cl(r_H)$ and $cl(r_G)$ are incompatible.

DEFINITION 5.6. *$Vp(G)$ consists of those vectors x such that*

- *$\exists H(H,G)$ is a very good pair,*
- *$\exists P \in \text{Min}(H,G)$, $x = cl(r_H)$,*
- *$cl(r_H)$ and $cl(r_G)$ are incompatible.*

The set $Vp(G)$ is called the set of valid parents *of G.*

5.1. Algorithms for counting, enumerating, and sampling. The algorithm has three phases. In the first phase, we compute the sets $Ext(H,G)$, $Conn(G)$, and $Vp(G)$ using dynamic programming. In the second phase, a directed acyclic graph (DAG) T is constructed. This DAG is a compact representation of the set of minimal perfect phylogenies and can be used to enumerate, count, or obtain statistical information about the set of minimal perfect phylogenies. The third phase, therefore, can be used for any of these purposes.

5.1.1. Phase I. Computing the sets of vectors. Computing $Ext(H,G)$. Given a very good pair (H,G), we consider all partitions of $H - G$ into at most $r-1$ proper clusters $G_1, G_2, \ldots, G_t$ and consider the (possible) perfect phylogeny for H which has root x with subtrees perfect phylogenies for $G, G_1, G_2, \ldots, G_t$. The canonical labeling for x is then an element of $Ext(H,G)$.

Computing $Conn(H)$. Recall that $Conn(H)$ is the set of all canonical labelings of the roots of *minimal perfect subphylogenies* for H.

The candidates for membership in $Conn(H)$ are the elements of $Ext(H,G)$ where (H,G) is a very good pair. Thus we have only a "small" number of x's to run through. To determine whether a particular x is in $Conn(H)$ we will need the following two lemmas.

LEMMA 5.7. *Suppose x is a vector without dummy states. Then $x \in Conn(H)$ if and only if $x \in Vp(G)$ for every $G \in H/x$.*

Proof. Suppose $x \in Conn(H)$. There is a minimal perfect phylogeny P for H in which the canonical labeling for the root is x. P is also a perfect subphylogeny on G_i for each $G_i \in H/x$ and thus induces a minimal subphylogeny for G_i. In the minimal induced subphylogeny, if the root of G_i has a canonical labeling which is compatible with x, it can be identified with x. This is a contradiction to the definition of H/x. Therefore $x \in Vp(G_i)$ for all $G_i \in H/x$. In fact, this shows that *any* perfect phylogeny for G_i must have a root labeling that is incompatible with x.

Conversely, if $x \in Vp(G)$ for all $G \in H/x$, then we can define a perfect subphylogeny for H by making x the parent of the roots of the perfect phylogenies for each P_G, where P_G is a minimal perfect subphylogeny for G. Thus, $x \in Conn(H)$. ∎

LEMMA 5.8. *Suppose x is a vector with at least one dummy state. Then $x \in Conn(H)$ if and only if there exists a G such that*

- *(H,G) is very good,*
- *$x \in Vp(G)$, and*
- *either $x \in Vp(H-G)$ or $x = Sv(H,G) \oplus y$ for some $y \in Conn(H-G)$.*

Proof. Suppose $x \in Conn(H)$. Then there is a minimal perfect subphylogeny for H for which x is the canonical labeling of the root r_H. If r_H has two children, then H is formed by taking a minimal perfect phylogeny for two proper clusters G and $H-G$, and $x \in Vp(G) \cap Vp(H-G)$. If r_H has more than two children, then first of all, x is an extension of $Sv(H,G)$. In addition, since x is part of a canonical labeling, the reason that any dummy state in $Sv(H,G)$ is set to a nondummy state in x is because the vector $y \in Conn(H-G)$ forces this change. Thus $x = Sv(H,G) \oplus y$.

Conversely, if $x \in Vp(G) \cap Vp(H-G)$ for some very good (H,G), then we can get a minimal perfect phylogeny for H by combining the minimal perfect phylogenies for G and $H-G$ whose roots are incompatible with x. Similarly, if $x \in Vp(G)$ and $x = Sv(H,G) \oplus y$ for some $y \in Conn(H-G)$, then let T be the minimal perfect subphylogeny for $H-G$ whose root has a canonical labeling y, and let T' be the minimal perfect subphylogeny for G whose canonical labeling is $Sv(H,G)$. Then the combination of these two subphylogenies is a subphylogeny for H which is minimal, since the edges from its root (given this canonical labeling x) cannot be contracted. □

Computing $Vp(G)$. To compute $Vp(G)$, we simply look at each very good pair (H,G), examine each $x \in Ext(H,G)$, and include those x such that for some $y \in Conn(G)$, x and y are incompatible.

Analysis of Phase I. To compute these sets, we first compute all very good pairs (H,G). We then compute $Ext(H,G)$ for each such very good pair. Afterwards, we order the proper clusters G which are members of very good pairs, by cardinality, and start with the smallest.

For each such proper cluster G, we compute $Conn(G)$ and $Vp(G)$ by the above method. The most expensive operation is computing $Ext(H,G)$ for all very good pairs (H,G). For $O(2^{2r}k^2)$ pairs (H,G) we need to look at $O(2^{r^2}k^r)$ possible vectors, each of which takes $O(k)$ to examine. Thus the total time is $O(2^{2r+r^2}k^{r+3})$.

5.1.2. Phase II. Constructing the DAG. At this stage, for every proper cluster H that occurs as the set of leaves of a subtree of a minimal phylogeny induced by the removal of an edge, we have determined the sets $Conn(H)$ and $Vp(H)$. We now proceed as follows.

Let H be a proper cluster and let y be a vector in $Vp(H)$. The subroutine we describe determines what minimal perfect phylogenies of H are possible that have y as the parent of the root of H.

Let $x \in Conn(H)$ such that x is not compatible with y. There are two cases. Suppose x is fully set. We create a graph $\Gamma(H,x)$ in which the nodes are the proper clusters G such that (H,G) is very good and $x \in Vp(G)$. There is an edge between two nodes G and G' if they represent two clusters having a nonempty intersection. Then a consequence of Lemma 5.7 is that *every maximal independent set* in this graph represents a partition of H into proper clusters which must be realized in some perfect phylogeny.

Now suppose that x is not fully set. For each G such that (H,G) is very good and $x \in Vp(G)$, we check to see if $x \in Vp(H-G)$ or if there is a vector $z \in Conn(H-G)$ such that $x = z \oplus Sv(H,G)$. Each of these possibilities gives rise to possible structural decompositions of the perfect phylogeny for H, and we recurse on components.

This phase of the algorithm produces the following output.

It produces a DAG where there are two kinds of nodes—sum nodes and product nodes. A sum node is labeled by a pair (H,y), where H is a proper cluster and $y \in Vp(H)$. The children of this sum node are product nodes. Each product node

represents a partition of H into proper clusters $G_1, G_2, \ldots, G_l$ and a choice of a root label x from $Conn(H)$ such that x is not compatible with y and $x \in Vp(G_i)$ for $i = 1, \ldots, l$. These product node children of (H, y) are obtained in the manner described in the above paragraph, i.e., by considering all vectors $x \in Conn(H)$ which are incompatible with y. If x is fully set, the product nodes are obtained from the enumeration of the maximal independent sets of $\Gamma(H, x)$. If x is not fully set, then a product node, $(G, H - G; x)$, is created for each pair $G, H - G$ such that $x \in Vp(G) \cup Vp(H - G)$. Also, for each $z \in Conn(H - G)$ such that $x = Sv(H, G) \oplus z$, we enumerate all maximal independent sets in $\Gamma(H - G, z)$ and create a product node for each such set, where the product node contains the proper clusters in the independent set as well as G and is also labeled by the vector x. Note that the enumeration of maximal independent sets in graphs of the form $\Gamma(H, x)$ may be repeated many times in the above algorithm. We could modify the algorithm to enumerate these sets exactly once for each such $\Gamma(H, x)$ and then memorize the result. While this may improve the practical performance, it does not change the asymptotic complexity since each re-enumeration of these sets produces new minimal perfect phylogenies.

The product node $(G_1, G_2, \ldots, G_l; x)$ in turn points to l different sum nodes which are labeled by pairs (G_i, x) for $i = 1, \ldots, l$. Note that the DAG has at most one sum node for each possible label pair.

To finish the description of the DAG we need to describe its "boundaries." First of all, note that if all species in S are distinct and a perfect phylogeny exists for S, then there must be some state of some character which is present in exactly one species, say x. (If not, look at any perfect phylogeny P for S and note that it contains a pair of sibling leaves a, b. Since both a and b share states with other species on all characters, a and b are forced to be identical.) Thus the top of the DAG will be a sum node labeled $(S - x, x)$. (Note that $S - x$ is a proper cluster.) The bottom of the DAG consists of pairs (G, y) where G is a proper cluster with cardinality 1. Note also that the assumption that all species in S are forced to be leaves guarantees that each singleton subset of S is a proper cluster. (To remove this assumption, we would have to modify the algorithm as follows. When considering proper cluster H and a particular vector $x \in Conn(H)$ if x is compatible with a species $s \in H$, we also allow the possibility that r_H is labeled with s, thus making s an internal node.)

Starting from the top sum node, if we always follow one of the children from each sum node and all of the children from each product node, we will have implicitly defined a minimal perfect phylogeny. (The topology of this perfect phylogeny can be obtained from the above traversal by homeomorphically eliminating the product nodes.)

The following lemma proves some properties of this DAG representation.

LEMMA 5.9.

1. *Every node in the DAG is useful in the implicit representation of some minimal perfect phylogeny.*
2. *Every minimal perfect phylogeny is implicitly represented by the DAG (in the above sense) exactly once.*

Proof. By construction, for any sum node labeled (H, x) there is a minimal perfect phylogeny for S which contains a subphylogeny for H whose root r_H is labeled incompatibly with x. If $(G_1, \ldots, G_l; y)$ is a product node child of H, then y is incompatible with x and can be used to label r_H without destroying minimality. Also, since y is in $Vp(G_i)$ for all i, there are minimal subphylogenies for the G_i's whose roots can be labeled incompatibly with y. Thus, all sum nodes and product nodes are useful in some minimal perfect phylogeny.

Since each sum node has a distinct label, each minimal perfect phylogeny is represented at most once in the DAG. Lemmas 5.7 and 5.8 exhaustively enumerate all possibilities for the top-level decomposition of a subphylogeny for a very good proper cluster H with a valid parent y. Since the DAG contains all these possibilities, it implicitly represents all minimal perfect phylogenies. □

5.1.3. Phase III. Enumerating, counting, or sampling. Once this DAG is generated, we can count the number of minimal perfect phylogenies by working bottom-up. If the number at all descendants of a node has been determined, the number at the node can be determined by an addition if it is a sum node or by a multiplication if it is a product node. Once such a count can be made it is a very simple matter to use these counts to introduce an implicit numbering of the minimal perfect phylogenies and to enumerate or generate the ith minimal perfect phylogeny given the ordering. Clearly, it is also possible to obtain other statistics efficiently from this structure. The technique used to enumerate is also as easy as the counting.

5.2. Analysis. In section 5.1.1 we showed that phase I requires $O(2^{2r+r^2}k^{r+3})$ time. Phases II and III have running times that depend on the number of nodes and edges in the DAG. The generation of sum nodes can be easily accomplished once their parent product nodes have been found. Thus the complexity of phase II is essentially the complexity of finding all product nodes. The critical step in this process is the enumeration of maximal independent sets in graphs of the form $\Gamma(H, x)$. In general, for a graph G containing v nodes and e edges, the enumeration of all maximal independent sets can be done by a simple backtracking algorithm that takes $O(v+e)$ time to produce the next maximal independent set at each point. Such an algorithm is a polynomial delay algorithm [10] since it takes only polynomial time to produce the next member of the list being enumerated. In our case, v is at most $2^r k$ and hence e is at most $2^{2r}k^2$. Noting that the number of product nodes in the DAG is at most n times the number of minimal perfect phylogenies L, we get a bound on the running time of phase II to be $O(2^{2r}k^2Ln)$.

All of the computations of phase III involve simple traversals of the DAG constructed in phase II. The work done at each product node is upper bounded by the number of its children and the work done at each sum node is upper bounded by the number of top-level decompositions of the subphylogeny for the proper cluster represented at that node. By charging this work to individual minimal perfect phylogenies in a natural manner, it can be seen that no minimal perfect phylogeny is charged more than $O(n)$ cost, and hence phase III can be performed in $O(Ln)$ time.

5.3. Polynomial delay. In this section we show that the above approach can be modified to design an algorithm with polynomial delay for the enumeration of all minimal perfect phylogenies. Note that the algorithm, as described above, constructs the DAG in a breadth-first fashion and hence is not polynomial delay. We simply modify the algorithm to work in a depth-first fashion, fully exploring one child of a sum node before constructing others by simple backtracking. Since each product node can be constructed from its parent sum node with $O(2^{2r}k^2)$ delay, and since there are $O(n)$ product nodes in any minimal perfect phylogeny, this depth-first search version of the algorithm is an $O(2^{2r}k^2n)$-delay algorithm for enumerating minimal perfect phylogenies.

6. Comments. The algorithm we have presented here is a significant theoretical improvement on the algorithm of Agarwala and Fernández-Baca [1]. The running time of this algorithm ($O(f(r)nk^2)$) is close to the running time of the algorithms

for $r = 3, 4$, for which $O(n^2k)$ algorithms were found [13], and matches the $O(nk^2)$ running time of the algorithm for $r = 3$ given by Dress and Steel [7]. Although the enumeration algorithms have bad worst-case running times, the actual running time when the number of minimal perfect phylogenies is small is likely to be much more reasonable.

Although perfect phylogenies rarely arise in molecular phylogenies, the recent application of perfect phylogeny to the reconstruction of evolutionary trees for families of natural languages has shown, surprisingly, that properly encoded linguistic information does yield data sets with perfect phylogenies (or in which very few characters must be removed in order to obtain perfect phylogenies [17]). Furthermore, even though perfect phylogenies are rare in molecular phylogenies, biologists still seek maximal sets of compatible characters. The problem of finding a maximum cardinality compatible subset of characters is, unfortunately, *NP-hard*, even for the case of binary characters [6]. Finding maximal sets, on the other hand, can be done in a greedy fashion, given an algorithm to determine compatibility of characters.

Since, until recently, the only algorithms for perfect phylogeny were for binary characters and two characters at a time, biologists have instead looked for maximal sets of pairwise compatible characters. This practice (of seeking sets of characters which are pairwise compatible) does not ensure set-wise compatibility. The algorithm that we have presented provides a tool which could be used by biologists to efficiently construct maximal subsets of compatible characters in a greedy manner. This greedy heuristic has already been shown to produce extremely reasonable phylogenetic trees [19] for biological data and for trees on natural languages [17]. The greedy heuristic for constructing maximal subsets of compatible characters is thus useful for any domain where it is possible to construct perfect phylogenies using almost all of the characters in the data set (i.e., using $k - O(1)$ characters). Perhaps we will find such data in the biological arena as well.

REFERENCES

[1] R. Agarwala and D. Fernández-Baca, *A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed*, SIAM J. Comput., 23 (1994), pp. 1216–1224.

[2] R. Agarwala and D. Fernández-Baca, *Fast and Simple Algorithms for Perfect Phylogeny and Triangulating Colored Graphs*, DIMACS Tech. Report 94-51, Rutgers University, New Brunswick, NJ, 1994.

[3] H. Bodlaender, M. Fellows, and T. Warnow, *Two strikes against perfect phylogeny*, in Proceedings of the 19th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, Springer-Verlag, New York, 1992, pp. 273–283.

[4] H. Bodlaender and T. Kloks, *A simple linear time algorithm for triangulating three-colored graphs*, J. Algorithms, 15 (1993), pp. 160–172.

[5] P. Buneman, *A characterization of rigid circuit graphs*, Discrete Math., 9 (1974), pp. 205–212.

[6] W. H. E. Day and D. Sankoff, *Computational complexity of inferring phylogenies by compatibility*, Syst. Zool., 35 (1986), pp. 224–229.

[7] A. Dress and M. Steel, *Convex tree realizations of partitions*, Appl. Math. Lett., 5 (1992), pp. 3–6.

[8] G. F. Estabrook, *Cladistic methodology: a discussion of the theoretical basis for the induction of evolutionary history*, Ann. Rev. Ecol. Syst., 3 (1972), pp. 427–456.

[9] G. F. Estabrook, C. S. Johnson Jr., and F. R. McMorris, *An idealized concept of the true cladistic character*, Math. Biosciences, 23 (1975), pp. 263–272.

[10] L. A. Goldberg, *Efficient Algorithms for Listing Combinatorial Structures*, Distinguished Dissertation Series, Cambridge University Press, Cambridge, 1993.

[11] D. Gusfield, *Efficient algorithms for inferring evolutionary trees*, Networks, 21 (1991), pp. 19–28.

[12] S. Kannan and T. Warnow, *Triangulating three-colored graphs*, SIAM J. Discrete Math., 5 (1992), pp. 249–258.
[13] S. Kannan and T. Warnow, *Inferring evolutionary history from DNA sequences*, SIAM J. Comput., 23 (1994), pp. 713–737.
[14] W. J. Le Quesne, *A method of selection of characters in numerical taxonomy*, Syst. Zool., 18 (1969), pp. 201–205.
[15] W. J. Le Quesne, *Further studies based on the uniquely derived character concept*, Syst. Zool., 21 (1972), pp. 281–288.
[16] F. R. McMorris, T. Warnow, and T. Wimer, *Triangulating vertex colored graphs*, SIAM J. Discrete Math., 7 (1994), pp. 296–306.
[17] D. Ringe, T. Warnow, and A. Taylor, *Character-Based Construction of Evolutionary Trees for Natural Languages*, IRCS Tech. Report, University of Pennsylvania, Philadelphia, PA, 1995.
[18] M. A. Steel, *The complexity of reconstructing trees from qualitative characters and subtrees*, J. Classification, 9 (1992), pp. 91–116.
[19] T. Warnow, *Constructing phylogenetic trees efficiently using compatibility criteria,* New Zealand J. Botany, 31 (1993), pp. 239–248.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1764–1784, December 1997

011

FAULT-TOLERANT MESHES WITH SMALL DEGREE*

JEHOSHUA BRUCK†, ROBERT CYPHER‡, AND CHING-TIEN HO§

Abstract. This paper presents constructions for fault-tolerant, two-dimensional mesh architectures. The constructions are designed to tolerate k faults while maintaining a healthy n by n mesh as a subgraph. They utilize several novel techniques for obtaining trade-offs between the number of spare nodes and the degree of the fault-tolerant network.

We consider both worst-case and random fault distributions. In terms of worst-case faults, we give a construction that has constant degree and $O(k^3)$ spare nodes. This is the first construction known in which the degree is constant and the number of spare nodes is independent of n. In terms of random faults, we present several new degree-6 and degree-8 constructions and show (both analytically and through simulations) that these constructions can tolerate large numbers of randomly placed faults.

Key words. fault-tolerance, mesh, array, interconnection networks, parallel computing, graph theory

AMS subject classifications. 68M10, 68M15, 68R10

PII. S0097539794274994

1. Introduction.

As the number of processors in parallel machines increases, physical limitations and cost considerations will tend to favor interconnection networks with constant degree and short wires, such as mesh networks [6]. In fact, the two-dimensional mesh is already one of the most important interconnection networks for parallel computers. Examples of existing two-dimensional mesh computers include the MPP (from Goodyear Aerospace), VICTOR (from IBM), and DELTA and Paragon (from Intel).

Another significant issue in the design of massively parallel computers is fault-tolerance. In order to create parallel computers that have very large numbers of complex processors, it will become necessary to utilize these machines even when several components have failed. In particular, the ability to tolerate even a small number of faults may allow a machine to continue operation between the occurrence of the first fault and the repair of the faults.

A large amount of research has been devoted to creating fault-tolerant parallel architectures. The techniques used in this research can be divided into two main classes. The first class consists of techniques which *do not* add redundancy to the desired architecture. Instead, these techniques attempt to mask the effects of faults by using the healthy part of the architecture to simulate the entire machine [2, 11, 17, 19, 23]. These techniques do not pay any costs for adding fault-tolerance, but they can experience a significant degradation in performance. The second class consists of

* Received by the editors September 30, 1994; accepted for publication (in revised form) December 8, 1995. A preliminary version of this paper appeared in *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany, ACM, New York, 1993, pp. 1–10.

http://www.siam.org/journals/sicomp/26-6/27499.html

† California Institute of Technology, Mail Code 116-81, Pasadena, CA 91125 (bruck@systems.caltech.edu). This research was performed while the author was at the IBM Almaden Research Center, San Jose, CA.

‡ Sun Microsystems Computer Company, 2550 Garcia Avenue, MSUMPK12-302, Mountain View, CA, 94043-1100 (cypher@eng.sun.com). This research was performed while the author was at the IBM Almaden Research Center, San Jose, CA.

§ IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 (ho@almaden.ibm.com).

techniques which *do* add redundancy to the desired architecture. These techniques attempt to isolate the faults, usually by disabling certain links or disallowing certain switch settings, while maintaining the complete desired architecture [1, 3, 4, 7, 8, 9, 10, 13, 14, 15, 18, 20, 22, 24, 25, 26, 27, 29]. The goal of these techniques is to maintain the full performance of the desired architecture while minimizing the cost of the redundant components.

One of the most powerful techniques for adding redundancy is based on a graph-theoretic model of fault-tolerance [18]. In this model, the desired architecture is viewed as a graph (called the *target graph*), and a fault-tolerant graph is created such that after the removal of k faulty nodes, the target graph is still present as a subgraph. This technique yields fault-tolerant networks that can tolerate both node faults and edge faults (by viewing a node incident with the faulty edge as being faulty) and can implement algorithms designed for the target network without any slowdown (due to the simulation of multiple nodes by a single node or the routing of messages through switches or intermediate nodes). Unfortunately, the degree of the fault-tolerant network created with this model can be prohibitively large. In particular, all previously published techniques for creating fault-tolerant meshes with exactly k spares have a degree that is linear in the number of faults being tolerated.

In this paper, we create fault-tolerant n by n meshes with small degree by trading-off the number of spare nodes with the degree of the fault-tolerant network. We consider both worst-case and random fault distributions. In terms of worst-case faults, we give a construction that tolerates k faults and has constant degree and $O(k^3)$ spares. This is the first construction known in which the degree is constant and the number of spares is independent of n. The only other known constant degree construction for this problem requires $\Theta(n^2)$ spares [27]. In terms of random faults, we present several new degree-6 and degree-8 constructions and show (both analytically and through simulations) that they can tolerate large numbers of randomly placed faults. Our constructions require at most $O(n)$ spares and appear to be of practical interest. The only other known construction that is proven to tolerate large numbers of random faults was created by Tamaki [27]. That construction can tolerate nodes and edges which fail with constant probability, but requires $\Theta(n^2)$ spares and has degree $O(\log \log n)$.

In addition, our construction for worst-case faults is shown to require only wires of length $O(k^3)$ in Thompson's VLSI model [28], while our constructions for random faults are shown to require only constant-length wires. Thus our fault-tolerant constructions maintain much of the scalability of the mesh network. We remark that we use Thompson's VLSI model only because it provides a well-established means for quantifying the locality of an interconnection network; the use of this model does not imply that the constructions presented here are designed for the wafer-scale implementation of a parallel machine. In fact, most existing parallel machines have one, or at most a few, processors per chip. This fact motivates our concern about the degree of the fault-tolerant network (because of the limited number of pins available to connect one chip to another [12]).

The remainder of this paper is organized as follows. Definitions and several previously known results are given in section 2. The results for worst-case fault distributions and random fault distributions are presented in sections 3 and 4, respectively.

2. Preliminaries. definitions. Let k be a nonnegative integer and let $T = (V, E)$ be a graph. The graph $F = (V', E')$ is a *k-fault-tolerant graph with respect to T*, denoted a k-FT T, if the subgraph of F induced by any set of $|V'| - k$ nodes

contains T as a subgraph. The graph T will be called the *target graph*. The graph F will be said to contain $|V'| - |V|$ *spare nodes* (or *spares*).

DEFINITION. *The cycle with n nodes will be denoted C_n.*

DEFINITION. *The two-dimensional mesh with $r \geq 2$ rows and $c \geq 2$ columns will be denoted $M_{r,c}$. Each node in $M^{r,c}$ has a unique label of the form (i, j) where $0 \leq ir$ and $0 \leq j < c$. Each node (i, j) is connected to all nodes of the form $(i \pm 1, j)$ and $(i, j \pm 1)$, provided they exist. The node (i, j) will be said to be in* row i *and* column j.

DEFINITIONS. *Let n be a positive integer and let S be a set of integers in the range 1 through $n - 1$. The graph $C(n, S)$, called the n-*node circulant graph with connection set S [16, 14, 10]*, consists of n nodes numbered $0, 1, \ldots, n - 1$. Each node i is connected to all nodes of the form $(i \pm s) \bmod n$, where $s \in S$. The graph $D(n, S)$, called the n-*node diagonal graph with connection set S [10]*, consists of n nodes numbered $0, 1, \ldots, n-1$. Each node i is connected to all nodes of the form $i \pm s$ where $s \in S$, provided they exist. (The terms "circulant" and "diagonal" refer to the structure of the adjacency matrix.) The values in a connection set S will be referred to as "jumps" or "offsets" and an edge defined through an offset s will be referred to as an s-*offset edge.

DEFINITION. *Let S be a set of integers and let k be a nonnegative integer. The* expansion of S *by k, denoted* expand(S, k)*, is the set T where*

$$T = \bigcup_{s \in S} \{s, s + 1, \ldots, s + k\}.$$

The following theorems give constructions for creating fault-tolerant circulant and diagonal graphs. The basic idea is to add offsets so that faulty nodes can be "jumped over." The construction for diagonal target graphs has lower degree because a cluster of faults can be avoided by placing the cluster in the position where the missing wraparound edges would jump over them.

THEOREM 2.1 (see [14]). *Let n be a positive integer, let S be a set of integers in the range 1 through $n - 1$, let k be a nonnegative integer, and let $T = expand(S, k)$. The circulant graph $C(n + k, T)$ is a k-FT $C(n, S)$.*

THEOREM 2.2 (see [10]). *Let n be a positive integer, let $y = \lceil n/3 \rceil$, let S be a set of integers in the range 1 through y, let k be a positive integer, and let $T = expand(S, \lfloor k/2 \rfloor)$. The circulant graph $C(n + k, T)$ is a k-FT $D(n, S)$.*

The following theorems relate meshes, circulant graphs, and diagonal graphs. Combining these theorems with the two previous theorems yields constructions for fault-tolerant meshes. The first theorem follows immediately from the row-major labeling of the nodes in a mesh. The second theorem follows from a *diagonal-major order* of the nodes in a mesh; see Figure 1 for an example.

THEOREM 2.3. *The mesh $M_{r,c}$ is a subgraph of $C(rc, \{1, c\})$ and of $D(rc, \{1, c\})$.*

THEOREM 2.4. *The mesh $M_{r,c}$ is a subgraph of $C(rc, \{c - 1, c\})$.*

Proof. Let $\phi(i, j) = ((i - j) \bmod r)c + j$. It is straightforward to verify that ϕ defines an embedding of $M_{r,c}$ into $C(rc, \{c - 1, c\})$. □

3. Worst-case faults. In this section we present a graph $\tilde{M}$ that is a k-FT $M_{n,n}$ and has constant degree and $O(k^3)$ spares. Our construction is hierarchical. We first construct a graph M' that is a k-FT $M_{r,c}$ (for some suitably chosen parameters r and c) and has degree which is dependent on k. We then replace each node in M' with a supernode (a graph with certain properties) to obtain a graph $\tilde{M}$ with constant degree.

0	33	26	19	12	5	38	31
8	1	34	27	20	13	6	39
16	9	2	35	28	21	14	7
24	17	10	3	36	29	22	15
32	25	18	11	4	37	30	23

FIG. 1. *An example of a diagonal-major ordering of a mesh.*

3.1. The basic construction. We first present a construction for a k-FT cycle with degree 4 and k^2 spare nodes. We will then use this construction to create the graph M' which is a k-FT $M_{r,c}$.

THEOREM 3.1. *Let k and N be positive integers where $N \geq k^2 + k + 1$, and let the graph $C' = C(N + k^2, \{1, k + 1\})$. The graph C' is a k-FT C_N.*

Proof. First consider the case where $(N + k^2) \bmod (k + 1) = 0$. For each i, $0 \leq i \leq k$, let X_i be the set consisting of all nodes $\{j | j \bmod (k + 1) = i\}$. Because there are only k faults and there are $k+1$ disjoint sets X_i, at least one of them must be fault-free. Let X be such a fault-free X_i. Note that the nodes in X form a fault-free cycle C'' of length $(N + k^2)/(k + 1)$ using the $(k + 1)$-offset edges. Next, we augment C'' to obtain a healthy cycle of length at least N. For any two adjacent nodes a and b in C'', if all k of the nodes in C' between a and b are healthy, we traverse all k of these nodes by using the 1-offset edges. On the other hand, if there is a fault between a and b, we skip over all k of the nodes between them by traversing the $(k + 1)$-offset edge connecting a and b. It is clear that we will traverse $(k + 1)$-offset edges at most k times, so the resulting augmented cycle will have at least N nodes. If it has more than N nodes, we can choose to traverse additional $(k + 1)$-offset edges, rather than 1-offset edges, until the cycle has length exactly N. An example of a 2-FT cycle is shown in Figure 2.

Now consider the case where $(N + k^2) \bmod (k + 1) = x \neq 0$. Let R be a region of $k + 1 + x$ consecutive healthy nodes in C'. Note that such a region must exist because $N + k^2 \geq 2k^2 + k + 1$, so there must be a region of $2k + 1$ or more consecutive healthy nodes between two faults. Without loss of generality, we will assume that R consists of the $k + 1 + x$ highest numbered nodes in C'. For each i, $0 \leq i \leq k$, create the cycle C_i'' as follows. First, start at node i and traverse the $(k + 1)$-offset edges until a node in R is reached. Then, traverse the 1-offset edges x times. Finally, traverse one additional $(k + 1)$-offset edge to return to i. Note that these $k + 1$ cycles share only nodes within R. Because all of the nodes in R are healthy, there must exist an i such that C_i'' is healthy. We can augment C_i'' as before to obtain a cycle of length N. □

THEOREM 3.2. *Let k, r, and c be positive integers where $r, c \geq 2$ and $rc \geq k^2 + k + 1$, let $N = rc$, and let $M' = C(N + k^2, \{1, k + 1\} \cup \{c + ik | 0 \leq i \leq k\})$. The graph M' is a k-FT $M_{r,c}$.*

Proof. Let $T = C(N, \{1, c\})$. We will prove that M' is a k-FT T. Applying Theorem 2.3 will complete the proof. First, it follows from Theorem 3.1 that in the presence of k faults, M' contains a cycle of N healthy nodes. Let C'' denote a cycle of healthy nodes constructed according to the proof of Theorem 3.1, and number the

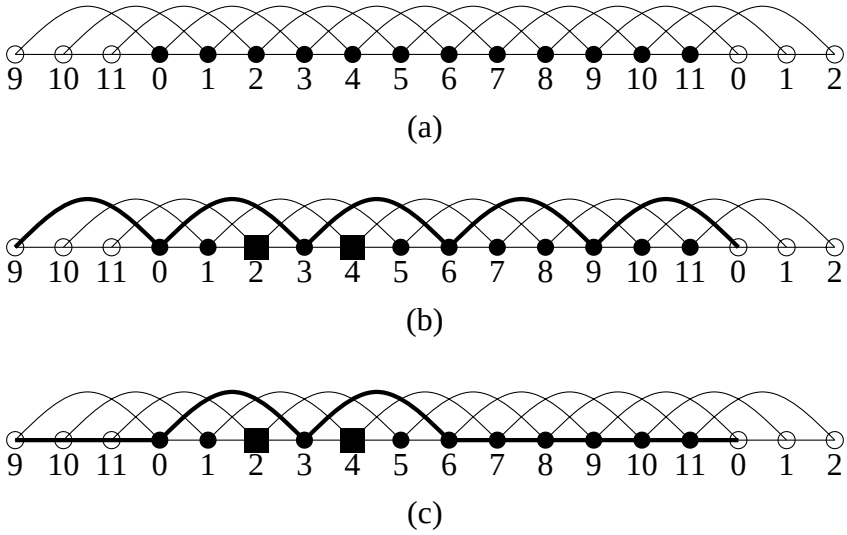


FIG. 2. *A degree-4, 2-fault-tolerant cycle with 4 spare nodes.*

nodes in C'' from 0 through $N-1$. We will now prove that any two nodes numbered a and b in C'', where $(a+c) \bmod N = b$, are connected in M'. Let a' and b' be the labels of a and b in M', and assume without loss of generality that $a' < b'$. We know that it is possible to traverse the cycle C'' from a to b by traversing 1-offset edges and at most k $(k+1)$-offset edges. Therefore, $b' - a' = c + jk$ for some integer j where $0 \leq j \leq k$, which implies that a and b are connected in M'. □

3.2. Hierarchical constructions. In the previous subsection we described a construction of a k-FT cycle with k^2 spare nodes and degree 4 and a construction of a k-FT two-dimensional mesh with k^2 spare nodes and degree $2k+6$. In this subsection we will present techniques for reducing the degree of these FT graphs. The general idea is to replace each node in the original FT graph by a small graph (which we call a supernode). Then, for each edge (a, b) in the original graph, one or more nodes in the supernode corresponding to a is connected to one or more nodes in the supernode corresponding to b. This approach results in a FT graph with lower degree than the original graph, although it does increase the number of spare nodes that are required.

3.2.1. Hierarchical fault-tolerant cycles. We illustrate the concept of a supernode by creating a hierarchical FT cycle.

THEOREM 3.3. *Let k and N be positive integers, where $N \geq k^2+k+1$, and let $\hat{C}$ be the graph with $2N+2k^2$ nodes, numbered 0 through $2N+2k^2-1$, and with edges specified as follows: Each odd numbered node i is connected to nodes $(i+1)$, $(i-1)$, and $i+2k+1$, and each even numbered node i is connected to nodes $(i+1)$, $(i-1)$, and $i-2k-1$, where all of the arithmetic is performed modulo $(2N+2k^2)$. Then $\hat{C}$ is a k-FT C_{2N}.*

Proof. The graph $\hat{C}$ can be obtained from the graph C' of Theorem 3.1 by replacing each node with a supernode consisting of a pair of nodes connected to one another. The edges that correspond to the positive direction connections in C' are connected to odd nodes in $\hat{C}$ while the edges that correspond to negative direction

connections in C' are connected to even nodes in $\hat{C}$. Consider the graph C' in which a node a is faulty iff at least one of the nodes in the supernode corresponding to a in $\hat{C}$ is faulty. It follows from Theorem 3.1 that C' contains a cycle of N healthy nodes. Therefore, $\hat{C}$ must contain a cycle of $2N$ healthy nodes corresponding to the cycle of N healthy nodes in C'. □

Figure 3 shows an example of a 2-FT cycle of degree 3 with $2k^2 = 8$ spares.

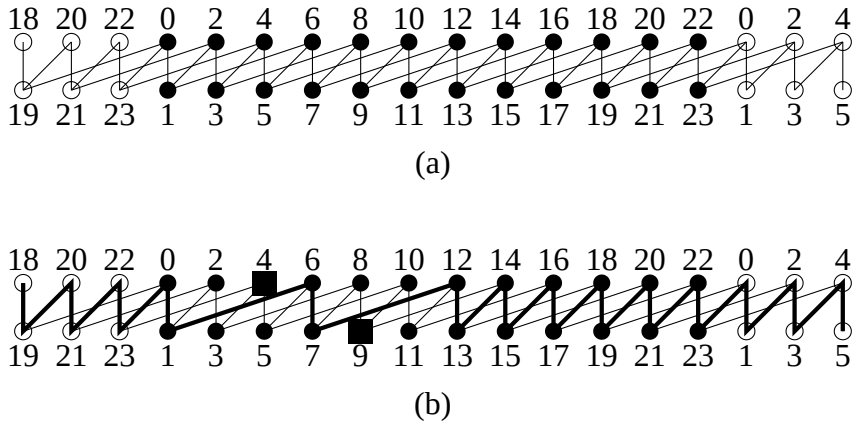


FIG. 3. *A degree-3, 2-fault-tolerant cycle with 8 spare nodes.*

3.2.2. Hierarchical fault-tolerant meshes. We will now show how hierarchical constructions can be used to reduce the degree of the graph M' of Theorem 3.2. We will begin with an approach that reduces the degree to $\Theta(\sqrt{k})$. We will then consider a more powerful technique that reduces the degree to a constant. The first approach uses the following graph as a supernode.

DEFINITION. *Let H_n be a graph with n nodes and degree 3 (if n is even) or degree 4 (if n is odd) such that for every pair of distinct nodes in H_n, there is a Hamiltonian path that has those nodes as endpoints. H_n graphs have been created for all $n \geq 2$* [5]. *See Figure 4 for an example.*

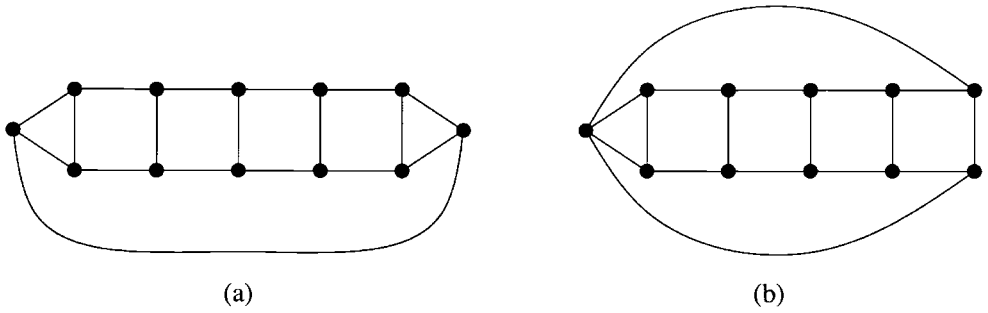


FIG. 4. *Examples of Hamiltonian graphs by Moon* [5] *with minimal degree. The number of nodes is even in* (a) *and is odd in* (b).

CONSTRUCTION 3.1. *Let k, r, c, n, and s be positive integers where $r, c, s \geq 2$, $rc \geq k^2 + k + 1$, and $2rs = c = n$, let $V = \{c + ik | 0 \leq i \leq k\}$, and let the graph $M' = C(rc + k^2, \{1, k+1\} \cup V)$. Let $\hat{M}$ be the hierarchical graph obtained from M' by replacing each node in M' by a supernode H_{2s}. Divide the nodes in each supernode arbitrarily into two halves of s nodes each. Add connections between supernodes as follows:*

1. *Connect each node in each supernode i to every node in supernodes $i-1$, $i+1$, $i-k-1$, and $i+k+1$ (all modulo $rc+k^2$). These edges, called horizontal edges, contribute $8s$ to the degree of each node.*
2. *For each offset $v \in V$ and for every supernode i, connect one of the nodes in the second half of supernode i to one of the nodes in the first half of supernode $(i+v) \bmod (rc+k^2)$. These edges, called vertical edges, should be evenly distributed among the nodes in each half of each supernode so they contribute at most $\lceil (k+1)/s \rceil$ to the degree of each node.*

Note that the degree of $\hat{M}$ is at most $8s + \lceil (k+1)/s \rceil + 3$. Choosing $s = \Theta(\sqrt{k})$ yields a graph $\hat{M}$ with degree $O(\sqrt{k})$ and with $O(k^{5/2})$ spare nodes.

THEOREM 3.4. *The graph $\hat{M}$ defined in Construction* 3.1 *is a k-FT $M_{n,n}$.*

Proof. Consider the graph M' of Theorem 3.2 in which a node a' is faulty iff at least one of the nodes in the supernode corresponding to a' in $\hat{M}$ is faulty. It follows from Theorem 3.2 that M' contains a healthy $M_{r,c}$ subgraph. We will show that this implies that $\hat{M}$ contains a healthy $M_{n,n}$ subgraph.

Let a' be any node in the healthy $M_{r,c}$ subgraph of M', and let $\hat{a}$ be the supernode in $\hat{M}$ corresponding to a'. We will view $\hat{a}$ as a column of $2s$ nodes in $M_{n,n}$. Note that a' has vertical neighbors $a' - v_1 \bmod (rc+k^2)$ and $a' + v_2 \bmod (rc+k^2)$, where v_1 and v_2 are in V. Let t be the node in the first half of $\hat{a}$ that is connected to a node in supernode $\hat{a} - v_1 \bmod (rc+k^2)$, and let b be the node in the second half of $\hat{a}$ that is connected to a node in supernode $\hat{a} + v_2 \bmod (rc+k^2)$. We will view t as being the top node and b as being the bottom node in the column of $2s$ nodes formed by $\hat{a}$. Recall that for every pair of nodes in H_{2s}, there is a Hamiltonian path that has those nodes as endpoints. Therefore, we can use the Hamiltonian path with endpoints t and b as the vertical connections within $\hat{a}$. Furthermore, the connections between the node b in one supernode and the node t in the next supernode provide the vertical connections between supernodes. Finally, note that a' has horizontal neighbors $a' - x_1 \bmod (rc+k^2)$ and $a' + x_2 \bmod (rc+k^2)$ where x_1 and x_2 are in $\{1, k+1\}$. Because each node in $\hat{a}$ is connected to every node in supernodes $\hat{a} - x_1 \bmod (rc+k^2)$ and $\hat{a} + x_2 \bmod (rc+k^2)$, the horizontal connections between supernodes are also present. □

We will now show how the use of a different supernode graph can yield a k-FT mesh with $O(k^3)$ spare nodes and constant degree. The following graph will be used as the supernode graph.

DEFINITION. *The graph P_k consists of $2k+4$ nodes. This graph consists of two parts, denoted S_1 and S_2, each of which is the graph $C(k+2, \{1,2\})$, plus an edge connecting node $k+1$ in S_1 with node $k+1$ in S_2. See Figure* 5 *for an example of P_6.*

Now we describe the construction of a k-FT mesh based on the graph P_k as a supernode.

CONSTRUCTION 3.2. *Let k, r, c, and n be positive integers where $r, c \geq 2$, $rc \geq k^2 + k + 1$, and $(2k+4)r = c = n$, let $V = \{c + ik | 0 \leq i \leq k\}$, and let $M' = C(rc + k^2, \{1, k+1\} \cup V)$.*

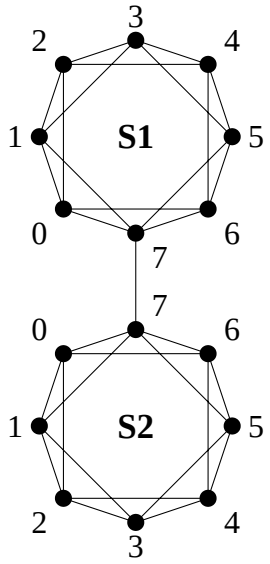


FIG. 5. *An example of the graph* P_6.

Let $\tilde{M}$ *be the hierarchical graph obtained from* M' *by replacing each node in* M' *by the supernode* P_k. *Add connections between supernodes as follows:*

1. *Connect each node* $j \in S_1$ *of supernode* i *to nodes* $\{j-2, j-1, j, j+1, j+2\} \bmod (k+2)$ *in* S_1 *of supernodes* $\{i-1, i+1, i-k-1, i+k+1\} \bmod (rc+k^2)$. *These edges, called horizontal edges, contribute* 20 *to the degree of each node in* S_1.
2. *Connect each node* $j \in S_2$ *of supernode* i *to nodes* $\{j-2, j-1, j, j+1, j+2\} \bmod (k+2)$ *in* S_2 *of supernodes* $\{i-1, i+1, i-k-1, i+k+1\} \bmod (rc+k^2)$. *These edges, also called horizontal edges, contribute* 20 *to the degree of each node in* S_2.
3. *Connect each node* $j \in S_2$ *of supernode* i, *where* $0 \le j \le k$, *to node* $j \in S_1$ *of supernode* $(i + c + jk) \bmod (rc + k^2)$. *These edges, called vertical edges, correspond to the* $k+1$ *offsets in* V *and contribute* 1 *to the degree of each node numbered less than* $k+1$ *in each half of each supernode.*

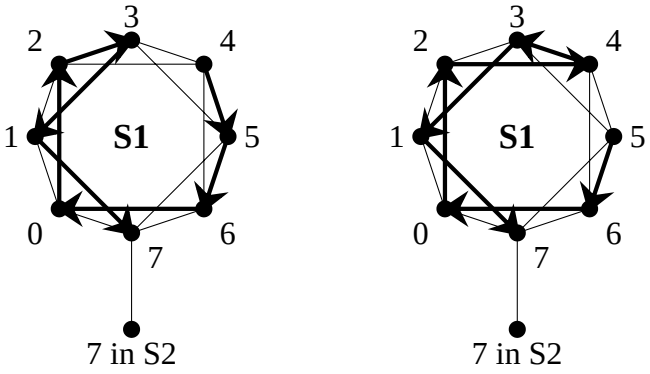
Note that the degree of $\tilde{M}$ is 25. The fact that $\tilde{M}$ is a k-FT mesh relies on the following lemmas.

LEMMA 3.5. *Consider the subgraph* $S = S_1$ (*or equivalently,* $S = S_2$) *of* P_k. *There exists a set of paths* $\{Q_0, Q_1, \ldots, Q_k\}$ *such that for each* i, $0 \le i \le k$, Q_i *is a Hamiltonian path through* S *with endpoints* i *and* $k+1$, *and for each* i, $0 \le i < k$, *and for each* j, $0 \le j \le k+1$, *if* a *is the* j*th node in* Q_i *and* b *is the* j*th node in* Q_{i+1}, *then* $(a-b) \equiv x \pmod{k+2}$ *where* $x \in \{-2, -1, 0, 1, 2\}$.

Proof. For each i, $0 \le i \le k$, define Q_i as follows. Start at i and traverse the 1-offset edges in the positive direction until node k is reached. Then traverse the 2-offset edges in the positive direction until either node $i-1$ or node $i-2$ is reached. If node $i-1$ is reached, traverse the 1-offset edge to node $i-2$ and then traverse the 2-offset edges in the negative direction until node $k+1$ is reached. On the other hand,

if node $i-2$ is reached before node $i-1$, traverse the 1-offset edge to node $i-1$ and then traverse the 2-offset edges in the negative direction until node $k+1$ is reached. See Figure 6 for an example of the paths Q_4 and Q_5 in S_1 of P_6.

If $i \le a \le k-1$, then $b = a+1$. If $a = k$, then $b = 0$. If $a = k+1$, then $b = a$. If $0 \le a \le i-1$, we have the following cases: (i) if a is even and $0 \le a \le i-2$, then $b = a+2$, (ii) if a is even and $a = i-1$, then $b = a+1$, and (iii) if a is odd and $1 \le a \le i-1$, then $b = a$. Therefore, in every case $(a-b) \equiv x \pmod{k+2}$ where $x \in \{-2,-1,0,1,2\}$. □



(a) Path starts from node 4 (b) Path starts from node 5

FIG. 6. *An example of two paths in S_1 of P_6 starting from nodes 4 and 5, respectively.*

LEMMA 3.6. *Let k, r, c, n, V, and M' be as defined in Construction 3.2. Consider any set of k faulty nodes in M' and let M be the healthy mesh $M_{r,c}$ in M' that is obtained by applying Theorem 3.2. Let a, b, a', and b' be any nodes in M' such that a and b are horizontal neighbors in M, a' and b' are horizontal neighbors in M, a and a' are vertical neighbors in M, and b and b' are vertical neighbors in M. If $a' = (a+c+ik) \bmod (rc+k^2)$ and $b' = (b+c+jk) \bmod (rc+k^2)$ where $0 \le i,j \le k$, then $|i-j| \le 1$.*

Proof. Assume without loss of generality that a is to the left of b in M and a' is to the left of b' in M. Note that $(b-a) \equiv x \pmod{rc+k^2}$ where $x \in \{1, k+1\}$ and $(b'-a') \equiv x' \pmod{rc+k^2}$ where $x' \in \{1, k+1\}$. Therefore, $(i-j)k \equiv (c+ik)-(c+jk) \equiv (a'-a)-(b'-b) \equiv x-x' \pmod{rc+k^2}$, which implies that $|i-j| \le 1$. □

THEOREM 3.7. *The graph $\tilde{M}$ defined in Construction 3.2 is a k-FT $M_{n,n}$ and has constant degree and $2k^3+4k^2$ spare nodes.*

Proof. The proof is analogous to that of Theorem 3.4. In particular, as in the proof of Theorem 3.4, we project the faults in $\tilde{M}$ onto M' and use Theorem 3.2 to find a healthy $M_{r,c}$ subgraph of M'.

Let a' be any node in the healthy $M_{r,c}$ subgraph of M' and let $\tilde{a}$ be the corresponding supernode in $\tilde{M}$. We view $\tilde{a}$ as a column of $2k+4$ nodes in $M_{n,n}$. We find top and bottom nodes t and b in $\tilde{a}$ as in the proof of Theorem 3.4, and we use Lemma 3.5 (twice) to create a Hamiltonian path through $\tilde{a}$ with endpoints t and b. Then, let b' be a node that is horizontally adjacent to a' in the healthy $M_{r,c}$ subgraph of M', and let $\tilde{b}$ be the corresponding supernode in $\tilde{M}$. It follows from Lemma 3.6

that the top nodes in $\tilde{a}$ and $\tilde{b}$ have positions within their supernodes that differ by at most one. A similar argument applies to the bottom nodes in $\tilde{a}$ and $\tilde{b}$. Therefore, it follows from Lemma 3.5 that for each i, $0 \leq i < 2k+4$, the ith node in the Hamiltonian path in $\tilde{a}$ has a horizontal connection to the ith node in the Hamiltonian path in $\tilde{b}$, which completes the proof. □

Hence, we have obtained a construction of a k-FT two-dimensional mesh with constant degree and $O(k^3)$ spare nodes. Although the construction given above is for a k-FT $M_{n,n}$ where n is a multiple of $2k+4$, it is straightforward to generalize the construction to arbitrary values of n as follows.

CONSTRUCTION 3.3. *Let k, r, c, and n be positive integers where $r, c \geq 2$, $rc \geq k^2+k+1$, $r = \lfloor n/(2k+4) \rfloor$, and $c = n$, let $V = \{c+ik | 0 \leq i \leq k\}$, and let $M' = C(rc+k^2, \{1, k+1\} \cup V)$.*

Let $n \bmod (2k+4) = \alpha$. If $\alpha = 0$, let $\hat{M}$ be the graph $\tilde{M}$ defined in Construction 3.2. *If $\alpha \neq 0$, first define the graph P'_k from P_k as follows. Add a node, denoted by x, to P_k, connect node x to node $k+1$ of S_1 in P_k, and connect node x to node $k+1$ of S_2 in P_k. Let $\hat{M}$ be the graph obtained by replacing each of the first $\alpha n + k^2$ nodes in M' by the supernode P'_k and replacing each of the remaining nodes in M' by the supernode P_k. Add connections between supernodes as follows:*

1. *Ignore the x nodes in the P'_k supernodes and add connections between supernodes as required by Construction* 3.2.
2. *For each supernode i, where $0 \leq i < \alpha n + k^2$, connect node x in supernode i to node x in supernode j, where $j \in \{i-1, i+1, i-k-1, i+k+1\}$ and $0 \leq j < \alpha n + k^2$.*

The following theorem is immediate from the preceding construction.

THEOREM 3.8. *Let k and n be positive integers, let $r = \lfloor n/(2k+4) \rfloor$, and let $c = n$. If $r, c \geq 2$ and $rc \geq k^2+k+1$, then there exists a k-FT $M_{n,n}$ with constant degree and $2k^3+5k^2$ spare nodes.*

Although the degree of $\hat{M}$ is increased to 26 (as both node $k+1$ of S_1 and node $k+1$ of S_2 have an edge to node x in the same supernode), one can easily reduce the number of horizontal edges of node $k+1$ to 4 (as opposed to 20 in the current definition) so that the degree of $\hat{M}$ remains 25. In fact, we remark that it is possible to reduce the degree still further by using a different graph for each supernode. Specifically, if each supernode is defined to be the product graph of P_k and a 4-node linear array, and if each supernode plays the role of a $(2k+4)$ by 4 submesh, it is possible to obtain a k-FT mesh with degree 12 and $8k^3+16k^2$ spare nodes. The details are omitted.

Finally, we will consider laying out the fault-tolerant graph $\hat{M}$ using Thompson's VLSI model [28]. One of the greatest advantages of two-dimensional mesh networks is that they can be laid out using only short (constant length) wires. The following theorem shows that the fault-tolerant graph $\hat{M}$ may require somewhat longer wires, but the wire lengths are still independent of n.

THEOREM 3.9. *It is possible to lay out the graph $\hat{M}$ defined in Construction* 3.3 *using only wires with length $O(k^3)$.*

Proof. We begin by presenting a mapping from the nodes in M' to the nodes in a torus network which maintains locality. We will then use standard techniques for laying out torus networks to obtain the final layout of $\hat{M}$. First, consider the case where k^2 is a multiple of c. In this case, lay out the nodes in M' in row-major order on an $(rc+k^2)/c$ by c torus. It is straightforward to verify that any pair of nodes that are connected in M' map to nodes that are in columns of the torus that differ by at most $O(k^2)$ and in rows of the torus that differ by at most $O(1)$. This torus can then be

mapped to an $(rc+k^2)/c$ by c grid by using the standard technique of placing the first half of the torus columns (rows) in increasing order in the even numbered columns (rows) of the grid and the remaining torus columns (rows) in decreasing order in the odd numbered columns (rows) of the grid (see, for example, [21, p. 246]). Finally, each node in M' can be laid out using an $O(k)$ by $O(k)$ square. The vertical tracks between grid columns are $O(k)$ wide and the horizontal tracks between grid rows are $O(k^3)$ wide (to accommodate wires that traverse $O(k^2)$ nodes, each of which is $O(k)$ wide). Thus each wire is of length $O(k^3)$.

Now consider the case where c does not evenly divide k^2. In this case, let $\alpha = k^2 \bmod c$ and use a $(rc+k^2)/c$ by $c+1$ torus. The nodes of M' are placed in the torus in row-major order, with the first α rows receiving $c+1$ nodes and all remaining rows receiving only c nodes. Again, it is straightforward to verify that any pair of nodes that are connected in M' map to nodes that are in columns of the torus that differ by at most $O(k^2)$ and in rows of the torus that differ by at most $O(1)$. This torus can then be laid out as described in the case where k^2 is a multiple of c. □

4. Random faults. In this section we consider random fault distributions. More specifically, we will assume that the fault-tolerant graph contains k faults, and that every configuration of k faulty nodes is equally likely. We will focus on the problem of creating fault-tolerant graphs for the mesh $M_{n,n}$. We will present six constructions for fault-tolerant meshes, analyze their asymptotic fault-tolerance, and study their fault-tolerance for realistic values of n.

The first three constructions are simple generalizations of previously known constructions [10] designed to tolerate worst-case fault distributions, while the remaining three constructions are new. In particular, the fourth construction introduces the concept of adding "dummy faults" in order to provide a fairly regular fault pattern. The fifth construction introduces the use of 2 by 2 "submeshes," and the sixth construction combines the use of dummy faults with the use of submeshes.

DEFINITION. *The graph* $T_1(n) = C(n^2, \{n-1, n\})$. *Recall from Theorem* 2.4 *that* $T_1(n)$ *contains* $M_{n,n}$ *as a subgraph.*

DEFINITION. *The graph* $T_2(n) = D(n^2, \{1, n\})$. *Recall from Theorem* 2.3 *that* $T_2(n)$ *contains* $M_{n,n}$ *as a subgraph.*

DEFINITION. *A graph* tolerates $\Theta(f(n))$ random faults *iff* $o(f(n))$ *random faults can be tolerated with a probability that is* $1 - o(1)$ *and* $\omega(f(n))$ *random faults can be tolerated with a probability that is* $o(1)$.

DEFINITION. *Given a circulant graph with* x *nodes, and given integers* y *and* z *where* $0 \le y, z < x$, *the* y*-node window starting at* z, *denoted* $W(y, z)$, *consists of the* y *nodes in the graph numbered* $z, z+1 \bmod x, \ldots, z+y-1 \bmod x$.

DEFINITION. *Given a circulant graph with* x *nodes, and given integers* y *and* z *where* $0 \le y, z < x$, *the distance between* y *and* z, *denoted* $dist(y, z)$, *is the minimum of* $z - y \bmod x$ *and* $y - z \bmod x$, *and nodes* x *and* y *are* consecutive *iff* $dist(y, z) = 1$.

DEFINITION. *Given a circulant graph with* x *nodes, and given integers* y *and* z *where* $1 \le y < x$ *and* $0 \le z < x$, *the* yth healthy node following (respectively, preceding) z *is the healthy node* a *such that there are exactly* y *healthy nodes in the set* $\{z+1 \bmod x, z+2 \bmod x, \ldots, a\}$ (*respectively,* $\{z-1 \bmod x, z-2 \bmod x, \ldots, a\}$).

It will be assumed throughout that $k \le n/2$ and $k = o(n)$. For constructions 1 through 4, we will consider only embeddings of the target graph in which node 0 of the target graph maps to some healthy node h in the fault-tolerant graph, and for each i, node i in the target graph maps to the ith healthy node following node h. For constructions 5 and 6, we will consider only embeddings obtained by viewing 2 by

2 "submeshes" that contain faults as representing faulty nodes in the corresponding fault-tolerant graph.

4.1. Construction 1. The first construction is based on the target graph $T_1(n)$.

DEFINITION. *The graph* $M_1(n,k) = C(n^2+k, \{n-1, n, n+1\})$.

Note that $M_1(n,k)$ has degree 6. The idea behind this construction is that it can tolerate faults by using the $(n+1)$-offset edges to jump over the faults.

LEMMA 4.1. *Assume that* $M_1(n,k)$ *contains* k *faulty nodes.* $M_1(n,k)$ *tolerates the faults iff for each* i, $0 \le i < n^2+k$, $W(n+1,i)$ *contains at most one fault.*

Proof. First, assume that for each i, $0 \le i < n^2+k$, $W(n+1,i)$ contains at most one fault. In this case, given any healthy node i, $W(n+2,i)$ contains at most one fault. Therefore, there is an edge between each healthy node and both the $(n-1)$st healthy node following it and the nth healthy node following it. As a result, $M_1(n,k)$ contains a healthy copy of $T_1(n)$.

Next, assume that there exists an i, $0 \le i < n^2+k$, such that $W(n+1,i)$ contains two or more faults. Let a be the first healthy node preceding i. Let a' be the node in $T_1(n)$ that maps to a, let b' be node $a'+n \bmod n^2$ in $T_1(n)$, and let b be the node to which b' maps. Note that b is the nth healthy node following a, so $b \notin W(n+1,i)$ and a and b are not connected to one another. □

LEMMA 4.2. *Let* M' *be a circulant graph with* $\Theta(n^2)$ *nodes and let* $y = \Theta(n)$. *Assume that* M' *contains* k *randomly located faulty nodes. If* k *is* $o(n^{1/2})$, *the probability that there exists a node* i *such that* $W(y,i)$ *contains two or more faults is* $o(1)$.

Proof. Given any two faults a and b, the probability that there exists a node i such that both a and b lie in $W(y,i)$ is $\Theta(n^{-1})$. There are $o(n)$ distinct pairs of faults, so the probability that there exists a node i such that $W(y,i)$ contains two or more faults is $o(n^{-1}n) = o(1)$. □

LEMMA 4.3. *Let* M' *be a circulant graph with* $\Theta(n^2)$ *nodes and let* $W = W(y,z_1)$, $W(y,z_2)$, $\ldots, W(y,z_q)$ *be a collection of* q *mutually disjoint* y*-node windows in* M', *where* $y = \Theta(n)$ *and* $q = \Theta(n)$. *Assume that* M' *contains* k *randomly located faulty nodes. If* k *is* $\omega(n^{1/2})$, *the probability that there exists a window in* W *that contains two or more faults is* $1 - o(1)$.

Proof. Divide the faults into halves. After the first half of the faults have been placed, if no window in W contains two or more faults then there must be $\omega(n^{3/2})$ healthy locations, each of which lies within a window in W that contains a fault. Therefore, the probability that any given fault in the second half will lie in a window in W that contains another fault is $\omega(n^{-1/2})$. As a result, the probability that no window in W contains two or more faults after all of the faults have been placed is at most $(1-n^{-1/2})^{\omega(n^{1/2})} = (1-n^{-1/2})^{n^{1/2}\omega(1)} = (1/e)^{\omega(1)} = o(1)$. □

THEOREM 4.4. *The graph* $M_1(n,k)$ *tolerates* $\Theta(n^{1/2})$ *random faults.*

Proof. The proof is immediate from Lemmas 4.2, 4.3, and 4.1. □

4.2. Construction 2. The second construction is also based on the target graph $T_1(n)$.

DEFINITION. *The graph* $M_2(n,k) = C(n^2+k, \{n-1, n, n+1, n+2\})$.

Note that $M_2(n,k)$ has degree 8. It is similar to $M_1(n,k)$, except the $(n+2)$-offset edges allows it to jump over more faults. The proof of the following lemma is analogous to that of Lemma 4.1 and is omitted.

LEMMA 4.5. *Assume that* $M_2(n,k)$ *contains* k *faulty nodes.* $M_2(n,k)$ *tolerates the faults iff for each* i, $0 \le i < n^2+k$, $W(n+2,i)$ *contains at most two faults.*

LEMMA 4.6. *Let M' be a circulant graph with $\Theta(n^2)$ nodes and let $y = \Theta(n)$. Assume that M' contains k randomly located faulty nodes. If k is $o(n^{2/3})$, the probability that there exists a node i such that $W(y, i)$ contains three or more faults is $o(1)$.*

Proof. Given any three faults, the probability that there exists a node i such that all three faults lie in $W(y, i)$ is $\Theta(n^{-2})$. There are $o(n^2)$ distinct sets of three faults, so the probability that there exists a node i such that $W(y, i)$ contains three or more faults is $o(n^{-2}n^2) = o(1)$. □

LEMMA 4.7. *Let $f(n)$ be any function such that $1 \leq f(n) \leq n$. Given $\omega(n)$ independent Bernoulli trials, each of which has a probability of success of at least $1/f(n)$, the probability of at least $n/f(n)$ successes is $1 - o(1)$.*

Proof. Divide the trials into $\omega(n/f(n))$ groups, each of which contains at least $\lceil f(n) \rceil$ trials. Given any one group of trials, the probability of at least one success in that group is at least $1/2$. Therefore, given any $2\lceil n/f(n) \rceil$ groups, the probability of at least $n/f(n)$ successes is at least $1/2$. This implies that the probability that the entire set of $\omega(n)$ trials contains at least $n/f(n)$ successes is at least $1-(1/2)^{\omega(1)} = 1-o(1)$. □

LEMMA 4.8. *Let M' be a circulant graph with $\Theta(n^2)$ nodes and let $W = W(y, z_1)$, $W(y, z_2)$, $\ldots, W(y, z_q)$ be a collection of q mutually disjoint y-node windows in M', where $y = \Theta(n)$ and $q = \Theta(n)$. Assume that M' contains k randomly located faulty nodes. If k is $\omega(n^{2/3})$, the probability that there exists a window in W that contains three or more faults is $1 - o(1)$.*

Proof. Divide the faults into three groups, each of which contains $\omega(n^{2/3})$ faults. Consider the three following statements:

Statement 1. At least $n^{2/3}$ windows in W contain at least one fault each.

Statement 2. At least $n^{1/3}$ windows in W contain at least two faults each.

Statement 3. There exists a window in W that contains three or more faults.

For all sufficiently large n, after the first group of faults has been placed, at least one of the three statements above must be true.

First, consider the situation in which Statement 1 is true after the first group of faults has been placed. For each fault in the second group, consider that fault to be a success iff it lies in a window in W that contains a fault from the first group. Given any fault in the second group, the probability that it is a success is $\Omega(n^{-1/3})$. It follows from Lemma 4.7 that, with probability $1 - o(1)$, at least $n^{1/3}$ faults in the second group are successes.

Therefore, regardless of which statement is true after the first group of faults is placed, there is a probability of at least $1 - o(1)$ that, after the second group of faults is placed, either Statement 2 or Statement 3 (or both) is true. Now consider the situation in which, after the second group of faults is placed, Statement 2 is true and Statement 3 is false. For each fault in the third group, consider that fault to be a success iff it lies in a window in W that contains at least two faults from the union of the first and second groups. Given any fault in the third group, the probability that it is a success is $\Omega(n^{-2/3})$. It follows from Lemma 4.7 that with probability $1 - o(1)$ at least one fault in the third group is a success.

As a result, in any case there is a probability of at least $1 - o(1)$ that, after all of the faults have been placed, Statement 3 holds. □

THEOREM 4.9. *The graph $M_2(n, k)$ tolerates $\Theta(n^{2/3})$ random faults.*

Proof. The proof is immediate from Lemmas 4.6, 4.8, and 4.5. □

4.3. Construction 3. All of the remaining constructions are based on the target graph $T_2(n)$.

DEFINITION. *The graph* $M_3(n,k) = C(n^2+k, \{1,2,n,n+1\})$.

Note that $M_3(n,k)$ has degree 8. The 1-offset and 2-offset edges of the fault-tolerant graph implement the 1-offset edges of the target graph, and the n-offset and $(n+1)$-offset edges of the fault-tolerant graph implement the n-offset edges of the target graph.

LEMMA 4.10. *Assume that* $M_3(n,k)$ *contains* k *faulty nodes.* $M_3(n,k)$ *tolerates the faults if for each* i, $0 \le i < n^2+k$, $W(n+1,i)$ *contains at most one fault.*

Proof. Given any healthy node i, $W(n+2,i)$ contains at most one fault. Therefore, there is an edge between each healthy node and both the first healthy node following it and the nth healthy node following it. As a result, $M_3(n,k)$ contains a healthy copy of $T_2(n)$. □

LEMMA 4.11. *Assume that* $M_3(n,k)$ *contains* k *faulty nodes.* $M_3(n,k)$ *does not tolerate the faults if there exist* x *and* y, *where* $0 \le x, y < n^2+k$, $dist(x,y) \ge 2n$, $W(n+1,x)$ *contains at least two faults, and* $W(n+1,y)$ *contains at least two faults.*

Proof. Assume for the sake of contradiction that the faults can be tolerated. Let a be the first healthy node preceding x and let a' be the node in $T_2(n)$ that maps to a. If there exists a node b' in $T_2(n)$ where $b' = a' + n$, let b be the node to which b' maps. Note that b is the nth healthy node following a, so $b \notin W(n+1,x)$ and a and b are not connected to one another. Therefore, no such node b exists, which implies that $a' \ge n^2 - n$.

Let c be the first healthy node preceding y and let c' be the node in $T_2(n)$ that maps to c. A similar argument shows that $c' \ge n^2 - n$. As a result, $|a' - c'| \le n-1$, so $dist(a,c) \le n-1+k$ and $dist(x,y) \le n-1+2k < 2n$, which is a contradiction. □

THEOREM 4.12. *The graph* $M_3(n,k)$ *tolerates* $\Theta(n^{1/2})$ *random faults.*

Proof. If the number of faults is $o(n^{1/2})$, it follows from Lemmas 4.2 and 4.10 that the probability of tolerating the faults is $1 - o(1)$. If the number of faults is $\omega(n^{1/2})$, divide the faults into halves. Let $W_1 = W(y,0), W(y,y), W(y,2y), \ldots, W(y,qy)$ and let $W_2 = W(y,(q+2)y), W(y,(q+3)y), W(y,(q+4)y) \ldots, W(y,2qy)$, where $y = n+1$ and $q = \lfloor n/4 \rfloor$. Apply Lemma 4.3 to the first half of the faults with $W = W_1$, apply Lemma 4.3 to the second half of the faults with $W = W_2$, and apply Lemma 4.11 to complete the proof. □

4.4. Construction 4.

DEFINITION. *The graph* $M_4(n,k) = C(n^2+n+k, \{1,2,n+1,n+2\})$.

Note that $M_4(n,k)$ has degree 8. The 1-offset and 2-offset edges of the fault-tolerant graph implement the 1-offset edges of the target graph, and the $(n+1)$-offset and $(n+2)$-offset edges of the fault-tolerant graph implement the n-offset edges of the target graph. In particular, the $(n+1)$-offset and $(n+2)$-offset edges of $M_4(n,k)$ can implement the n-offset edges of the target graph provided that each window of $n+1$ consecutive nodes contains at least one fault and each window of $n+2$ consecutive nodes contains at most two faults. Although it is very unlikely (or impossible) that each window of $n+1$ consecutive nodes contains at least one fault, we can view up to n healthy nodes as being "dummy faults" (because there are $n+k$ spares) in order to satisfy this requirement.

DEFINITION. *Given a circulant graph with* x *nodes, a* block of healthy nodes *is a window* $W(y,i)$, *where* $1 \le y < x$ *and* $0 \le i < x$, *consisting solely of healthy nodes such that both node* $i-1 \bmod x$ *and node* $i+y \bmod x$ *are faulty.*

Consider the following algorithm for adding dummy faults to $M_4(n,k)$.

ALGORITHM A. Consider separately each block of healthy nodes. Assume a block consists of y healthy nodes. There are three cases based on the value of y.

Case 1. $y \leq n$. In this case, do not add any dummy faults to the block.

Case 2. $n+1 \leq y \leq 2n$. In this case, add one dummy fault to the block. Place the dummy fault in the middle of the block so that it divides the block into two subblocks of healthy nodes, the first of which has $\lceil (y-1)/2 \rceil$ nodes and the second of which has $\lfloor (y-1)/2 \rfloor$ nodes.

Case 3. $2n+1 \leq y$. In this case, add two dummy faults that divide the block into three subblocks of healthy nodes, the first of which has $n-1$ nodes, the second of which has $z = y - 2n$ nodes, and the third of which has $n-1$ nodes. Let a and b denote these two dummy nodes. Then add an additional $x = \lfloor z/(n+1) \rfloor$ dummy faults between a and b. This leaves $w = z - x$ healthy nodes in the block, which are divided into $x+1$ subblocks of healthy nodes by the x dummy faults. Distribute the dummy faults so that each subblock has length $\lfloor w/(x+1) \rfloor$ or $\lceil w/(x+1) \rceil$.

The following lemmas establish properties of Algorithm A.

LEMMA 4.13. *Given w, x, and z in Case* 3 *above, $xn \leq w \leq (x+1)n$.*

Proof. Because $x = \lfloor z/(n+1) \rfloor$, $z \geq x(n+1)$, and $w = z - x \geq xn$. Because $x = \lfloor z/(n+1) \rfloor$, $z \leq xn + n + x$ and $w = z - x \leq xn + n = (x+1)n$. □

LEMMA 4.14. *After applying Algorithm* A*, no block of $n+1$ or more healthy nodes exists.*

Proof. If there is a block of $n+1 \leq y \leq 2n$ healthy nodes prior to applying Algorithm A, the algorithm adds a dummy node that divides the block into subblocks of at most $\lceil (y-1)/2 \rceil \leq n$ healthy nodes each. If there is a block of $2n+1 \leq y$ healthy nodes prior to applying Algorithm A, the algorithm adds dummy nodes a and b that divide the block into subblocks of $n-1$, $z = y - 2n$, and $n-1$ healthy nodes each. Then $x = \lfloor z/(n+1) \rfloor$ dummy faults are added to the subblock of z healthy nodes, leaving $w = z - x$ healthy nodes. These w healthy nodes occur in subblocks of length at most $\lceil w/(x+1) \rceil \leq \lceil (x+1)n/(x+1) \rceil \leq n$. □

LEMMA 4.15. *After applying Algorithm* A*, no dummy fault is consecutive with another (actual or dummy) fault, provided that $n \geq 2$.*

Proof. If there is a block of $n+1 \leq y \leq 2n$ healthy nodes prior to applying Algorithm A, the algorithm adds a dummy node that divides the block into subblocks of at least $\lfloor (y-1)/2 \rfloor \geq \lfloor n/2 \rfloor \geq 1$ healthy nodes each. If there is a block of $2n+1 \leq y$ healthy nodes prior to applying Algorithm A, the algorithm adds dummy nodes a and b that divide the block into subblocks of $n-1$, $z = y - 2n$, and $n-1$ healthy nodes each. Then $x = \lfloor z/(n+1) \rfloor$ dummy faults are added to the subblock of z healthy nodes, leaving $w = z - x$ healthy nodes. If $x = 0$, there are $w = z - 0 = y - 2n \geq 1$ healthy nodes between dummy faults a and b. If $x \geq 1$, the w healthy nodes occur in subblocks of at least $\lfloor w/(x+1) \rfloor \geq \lfloor xn/(x+1) \rfloor \geq \lfloor n/2 \rfloor \geq 1$ nodes each. □

LEMMA 4.16. *Consider any configuration of actual faults such that no two faults are consecutive and there does not exist a node i such that $W(2n+3, i)$ contains three or more faults, where $n \geq 2$. After applying Algorithm* A *to this configuration of faults, no two (actual or dummy) faults will be consecutive and there will not exist a node j such that $W(n+2, j)$ contains three or more (actual or dummy) faults.*

Proof. The fact that no two faults will be consecutive follows immediately from the preceding lemma. Now assume for the sake of contradiction that after applying Algorithm A, there exists a node j such that $W(n+2, j)$ contains three or more faults. Clearly, $W(n+2, j)$ must contain at least one dummy fault. Select one such dummy

fault and denote it as d, and let C denote the block of y originally healthy nodes containing d. Clearly, $y \geq n+1$.

If $n+1 \leq y \leq 2n$, then d is the only dummy fault in C, so either $W(n+2,j)$ contains two actual faults or $W(n+2,j)$ contains some other dummy fault located in some other block of originally healthy nodes. First, consider the case where $W(n+2,j)$ contains two actual faults. Let e and f denote these actual faults. Either y lies between e and f or it does not. If y lies between e and f, $W(n+2,j)$ must contain at least $y+2 \geq n+3$ nodes, which is a contradiction. Thus y does not lie between e and f. Now let y' denote the number of nodes between e and f. Because $W(n+2,j)$ contains only $n+2$ nodes and because there are at least $\lfloor (y-1)/2 \rfloor \geq y/2-1$ nodes between d and every actual fault, it follows that $y'+y/2+2 \leq n+2$, which implies that $y' \leq n-y/2$ and that there were three actual faults within a window of $y+y'+3 \leq n+y/2+3 \leq 2n+3$ nodes, which is a contradiction.

Now, consider the case where $W(n+2,j)$ contains a dummy fault located in another block of originally healthy nodes. Let d' denote such a dummy fault and let C' denote the block of y' originally healthy nodes containing d'. Clearly, $y' \leq 2n$, because otherwise there would be at least $n-1$ healthy nodes between d' and the nearest actual fault. However, note that if C' follows C, then there are at least $\lfloor (y-1)/2 \rfloor \geq \lfloor n/2 \rfloor$ consecutive healthy nodes following d and at least $\lceil (y'-1)/2 \rceil \geq \lceil n/2 \rceil$ consecutive healthy nodes preceding d', which implies that $W(n+2,j)$ contains at least $n+3$ nodes, which is a contradiction. The case in which C' precedes C is analogous.

If $2n+1 \leq y$, then either $W(n+2,j)$ contains at least one actual fault and at least one dummy fault or $W(n+2,j)$ contains three dummy faults and no actual faults. If $W(n+2,j)$ contains at least one actual fault and at least one dummy fault, then it must contain the $n-1$ healthy nodes which separate the dummy faults in C from the actual faults. Furthermore, because no two (actual or dummy) faults are consecutive, $W(n+2,j)$ must contain at least $n+3$ nodes, which is a contradiction. On the other hand, if $W(n+2,j)$ contains three dummy faults and no actual faults, let a, b, w, x, and z be as defined in Case 3 of Algorithm A. It follows that $x \geq 1$ and that $W(n+2,j)$ contains at least two blocks of $\lfloor w/(x+1) \rfloor$ or more healthy nodes in addition to the three dummy faults. However, the fact that $x \geq 1$ implies that $z \geq n+1$. Therefore, the dummy faults designated a and b cannot both be in $W(n+2,j)$, so it follows that $x \geq 2$. Therefore, $\lfloor w/(x+1) \rfloor \geq \lfloor xn/(x+1) \rfloor \geq \lfloor 2n/3 \rfloor \geq n/2$, so $W(n+2,j)$ contains at least n healthy nodes and three dummy faults, which is a contradiction. □

LEMMA 4.17. *After applying Algorithm* A*, at least* n^2 *healthy nodes remain.*

Proof. First, we will show that Algorithm A adds at most one dummy fault per $n+1/3$ originally healthy nodes. In Case 2 of Algorithm A, one dummy fault is added to a block of at least $n+1$ originally healthy nodes. In Case 3 of Algorithm A, if two dummy faults are added, there are at least $2n+1$ originally healthy nodes in the block, so at most one dummy fault is added per $n+1/2$ originally healthy nodes. In Case 3 of Algorithm A, if $i \geq 3$ dummy faults are added, there are at least $in+i-2$ originally healthy nodes in the block, so at most one dummy fault is added per $n+(i-2)/i$ originally healthy nodes. This quantity is minimized when $i=3$, at which point one dummy fault is added per $n+1/3$ originally healthy nodes.

Now consider the case in which exactly k actual faults exist. In this case there must be n^2+n originally healthy nodes, so at most $\lfloor (n^2+n)/(n+1/3) \rfloor \leq n$ dummy faults are added, and at least n^2 healthy nodes remain. Now consider the case in which $k-x$ actual faults exist, where $x \geq 1$. At most $\lfloor (n^2+n+x)/(n+1/3) \rfloor \leq$

$\lfloor (n^2+n)/(n+1/3) \rfloor + \lceil x/(n+1/3) \rceil \leq n+x$ dummy faults are added, and at least n^2 healthy nodes remain. □

The proofs of the following two lemmas are analogous to those of Lemmas 4.10 and 4.11, and are omitted.

LEMMA 4.18. *Assume that $M_4(n,k)$ contains $f \leq n+k$ (actual or dummy) faults. $M_4(n,k)$ tolerates the faults if no two faults are consecutive and for each i, $0 \leq i < n^2+n+k$, $W(n+1,i)$ contains at least one fault and $W(n+2,i)$ contains at most two faults.*

LEMMA 4.19. *Assume that $M_4(n,k)$ contains f faulty nodes. $M_4(n,k)$ does not tolerate the faults if there exist x and y, where $0 \leq x,y < n^2+n+k$, $dist(x,y) \geq 4n$, $W(n+2,x)$ contains at least three faults, and $W(n+2,y)$ contains at least three faults.*

THEOREM 4.20. *The graph $M_4(n,k)$ tolerates $\Theta(n^{2/3})$ random faults.*

Proof. First, consider the case where the number of faults is $o(n^{2/3})$. It follows from Lemma 4.6 that with probability $1-o(1)$ there does not exist a node i such that $W(2n+3,i)$ contains three or more faults. Also, given any two faults, the probability that they are consecutive is $\Theta(n^{-2})$. There are $o(n^2)$ distinct pairs of faults, so the probability that there exists a pair of faults that are consecutive is $o(n^{-2}n^2) = o(1)$. Therefore, it follows from Lemmas 4.17, 4.16, and 4.18 that after applying Algorithm A, the faults can be tolerated with probability $1-o(1)$.

Next, consider the case where the number of faults is $\omega(n^{2/3})$. In this case, divide the faults into halves. Let $W_1 = W(y,0), W(y,y), W(y,2y), \ldots, W(y,qy)$ and let $W_2 = W(y,(q+4)y), W(y,(q+5)y), W(y,(q+6)y), \ldots, W(y,2qy)$, where $y = n+2$ and $q = \lfloor n/4 \rfloor$. Apply Lemma 4.8 to the first half of the faults with $W = W_1$, apply Lemma 4.8 to the second half of the faults with $W = W_2$, and apply Lemma 4.19 to complete the proof. □

4.5. Construction 5. Construction $M_5(n,k)$ is a hierarchical construction based on $M_3(n/2,k)$. It is defined only for even values of n.

DEFINITION. *The graph $M_5(n,k)$ is created from $M_3(n/2,k)$ as follows:*

1. *Create $n' = (n/2)(n/2)+k = n^2/4+k$ squares (that is, cycles of length 4) numbered 0 through $n'-1$.*
2. *For each square i, connect the upper right corner of i to the upper left corners of $(i+1) \bmod n'$ and $(i+2) \bmod n'$, and connect the lower right corner of i to the lower left corners of $(i+1) \bmod n'$ and $(i+2) \bmod n'$.*
3. *For each square i, connect the lower left corner of i to the upper left corners of $(i+n) \bmod n'$ and $(i+n+1) \bmod n'$, and connect the lower right corner of i to the upper right corners of $(i+n) \bmod n'$ and $(i+n+1) \bmod n'$.*

Note that $M_5(n,k)$ has degree 6. The idea behind this construction is that the squares act as 2 by 2 submeshes and the graph can be reconfigured if the corresponding fault-tolerant graph (namely $M_3(n/2,k)$) can tolerate faults located in the positions corresponding to the faulty squares (see the proof of Theorem 3.4 for a description of hierarchical fault-tolerant graphs). The following theorem follows immediately from Theorem 4.12.

THEOREM 4.21. *The graph $M_5(n,k)$ tolerates $\Theta(n^{1/2})$ random faults.*

4.6. Construction 6. Construction $M_6(n,k)$ is a hierarchical construction based on $M_4(n/2,k)$. It is defined only for even values of n.

DEFINITION. *The graph $M_6(n,k)$ is created from $M_4(n/2,k)$ as follows:*

1. *Create $n' = (n/2)(n/2)+(n/2)+k = n^2/4+n/2+k$ squares (that is, cycles of length 4) numbered 0 through $n'-1$.*

2. *For each square i, connect the upper right corner of i to the upper left corners of $(i+1) \bmod n'$ and $(i+2) \bmod n'$, and connect the lower right corner of i to the lower left corners of $(i+1) \bmod n'$ and $(i+2) \bmod n'$.*
3. *For each square i, connect the lower left corner of i to the upper left corners of $(i+n+1) \bmod n'$ and $(i+n+2) \bmod n'$, and connect the lower right corner of i to the upper right corners of $(i+n+1) \bmod n'$ and $(i+n+2) \bmod n'$.*

Note that $M_6(n,k)$ has degree 6. The following theorem follows immediately from Theorem 4.20.

THEOREM 4.22. *The graph $M_6(n,k)$ tolerates $\Theta(n^{2/3})$ random faults.*

4.7. Summary. Table 1 summarizes various characteristics, including the asymptotic fault-tolerance, of the six fault-tolerant constructions.

TABLE 1
Comparison of characteristics of the 6 FT meshes.

Construction	Symbol	Deg.	No. spares	Offsets	Asymp. FT
M_1	circ6	6	k	$\{n-1, n, n+1\}$	$\Theta(n^{1/2})$
M_2	circ8	8	k	$\{n-1, n, n+1, n+2\}$	$\Theta(n^{2/3})$
M_3	diag8	8	k	$\{1, 2, n, n+1\}$	$\Theta(n^{1/2})$
M_4	diag8r	8	$k+n$	$\{1, 2, n+1, n+2\}$	$\Theta(n^{2/3})$
M_5	diag6	6	$4k$	M_3 + submesh	$\Theta(n^{1/2})$
M_6	diag6r	6	$4k+2n$	M_4 + submesh	$\Theta(n^{2/3})$

Notice that both $M_2(n,k)$ and $M_4(n,k)$ have degree 8 and tolerate $\Theta(n^{2/3})$ faults, but $M_4(n,k)$ requires more spares than does $M_2(n,k)$. Thus, the technique of adding dummy faults does not in itself provide a more practical fault-tolerant network. Similarly, notice that both $M_1(n,k)$ and $M_5(n,k)$ have degree 6 and tolerate $\Theta(n^{1/2})$ faults, but $M_5(n,k)$ requires more spares than does $M_1(n,k)$. Thus, the technique of using 2 by 2 submeshes does not in itself provide a more practical fault-tolerant network. However, by combining these two techniques, $M_6(n,k)$ is the only degree 6 network that is capable of tolerating $\Theta(n^{2/3})$ faults.

Finally, we will consider laying out the fault-tolerant graphs presented in this section using Thompson's VLSI model [28]. The following theorem shows that, just like the mesh itself, all of the fault-tolerant constructions can be laid out with constant length wires.

THEOREM 4.23. *It is possible to lay out each of the graphs $M_i(n,k)$ where $1 \leq i \leq 6$ using only wires with length $O(1)$.*

Proof. The layouts for graphs $M_1(n,k)$, $M_2(n,k)$, $M_3(n,k)$, and $M_4(n,k)$ follow immediately from the techniques presented in the proof of Theorem 3.9. The layouts for graphs $M_5(n,k)$ and $M_6(n,k)$ follow from the layouts for $M_3(n/2,k)$ and $M_4(n/2,k)$, respectively, by replacing each node by a square of four nodes. □

4.8. Simulation results. Figures 7 to 9 show the simulation results for the fault tolerance of an $n \times n$ target mesh for $n = 16$, 64, and 256, respectively. The probability given for each construction and each value of k is the result of 10,000 simulation trials.

For each figure, the probability of reconfiguration for each construction of the FT meshes, $M_i(n,k)$ where $1 \leq i \leq 6$, is plotted as a functions of k. Each curve has a name of the form "xyz," where "x" is either "circ" for circulant graph or "diag" for diagonal graph (as the basic target graph), "y" denotes the degree (6 or 8), and "z" is either "r" (designating an extra row of spare nodes or supernodes) or an empty

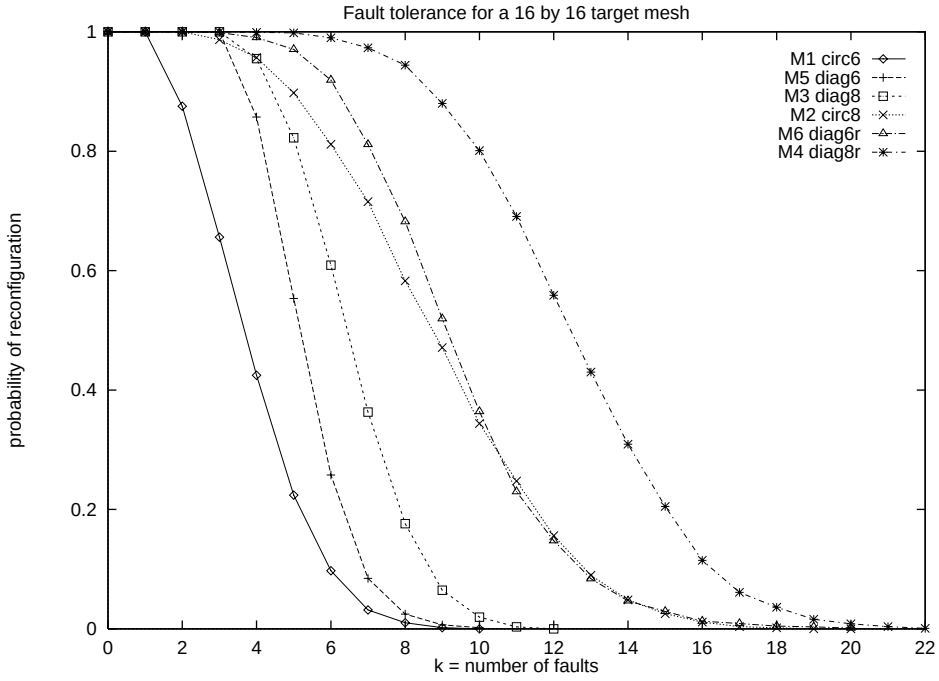


FIG. 7. *Simulation results of fault tolerance for a* 16 *by* 16 *target mesh.*

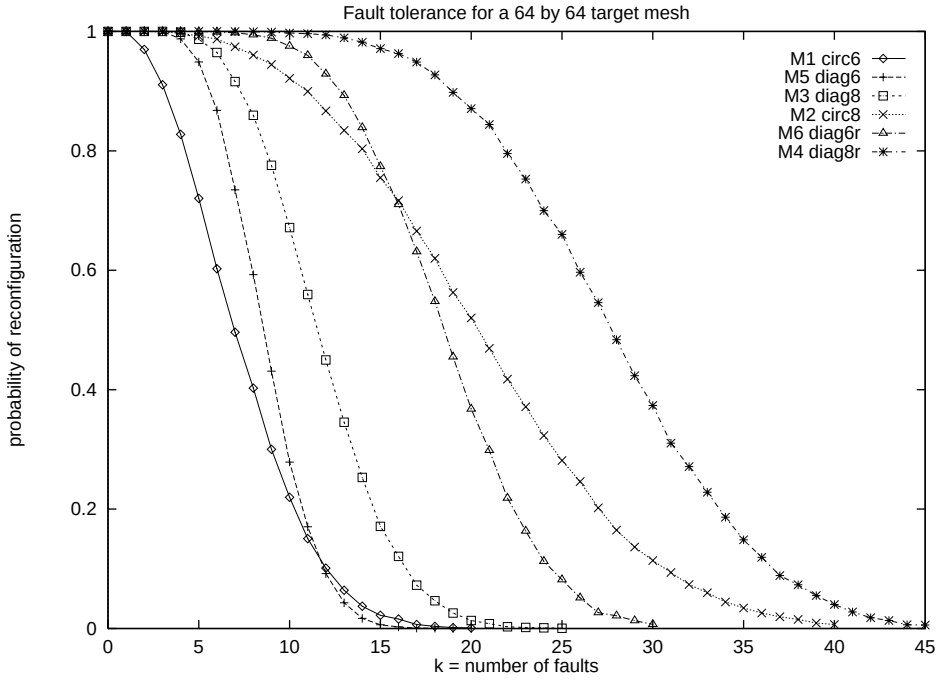


FIG. 8. *Simulation results of fault tolerance for a* 64 *by* 64 *target mesh.*

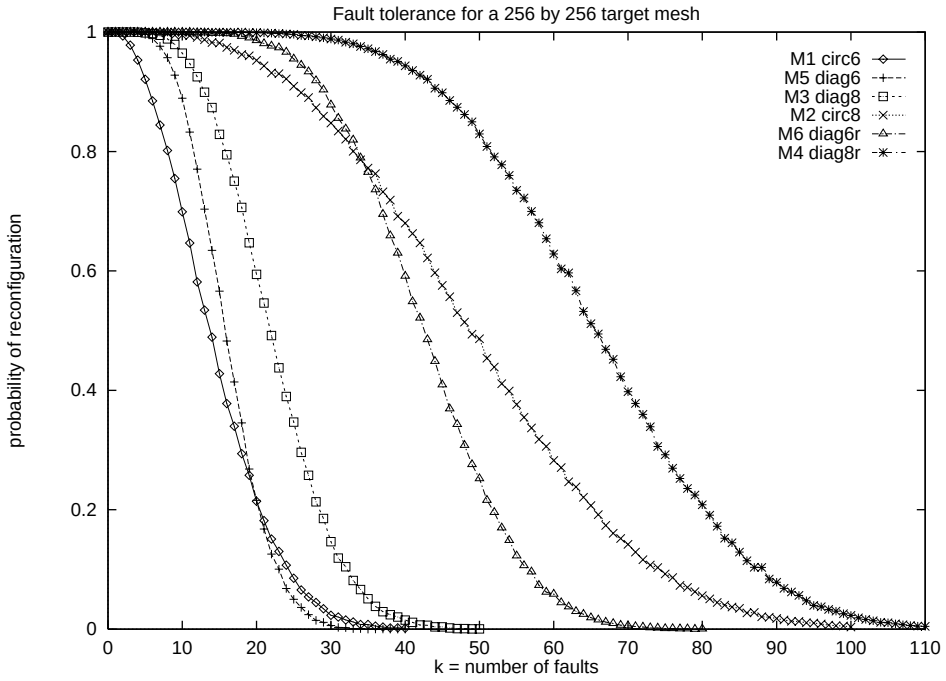


FIG. 9. *Simulation results of fault tolerance for a* 256 *by* 256 *target mesh.*

string. The solid lines denote the degree-6 FT meshes while the dotted lines denote the degree-8 FT meshes.

Note that the FT meshes for the three curves from the left tolerate $\Theta(n^{1/2})$ random faults, while the remaining three curves on the right can tolerate $\Theta(n^{2/3})$ random faults. Thus the asymptotic bounds proven above do appear to describe the behavior of these networks for realistic values of n. Also, note that the graph $M_6(n,k)$ (designated "diag6r" in the figures) performs the best out of the degree-6 networks studied, and that it has over a 90% chance of tolerating 12 faults when $n = 64$.

REFERENCES

[1] M. AJTAI, N. ALON, J. BRUCK, R. CYPHER, C-T. HO, M. NOAR, AND E. SZEMERÉDI, *Fault tolerant graphs, perfect hash functions and disjoint paths*, in Proc. of 33rd Annual IEEE Symp. on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 693–702.

[2] F. ANNEXSTEIN, *Fault tolerance in hypercube-derivative networks*, in Proc. of 1st Annual ACM Symposium on Parallel Algorithms and Architectures, Sante Fe, NM, 1989, ACM, New York, pp. 179–188.

[3] V. BALASUBRAMANIAN AND P. BANERJEE, *A fault tolerant massively parallel processing architecture*, J. Parallel Distrib. Comput., 4 (1987), pp. 363–383.

[4] K. E. BATCHER, *Design of a massively parallel processor*, IEEE Trans. Comput., C-29 (1980), pp. 836–840.

[5] C. BERGE, *Graphs*, North–Holland, Amsterdam, 1985, p. 218, a theorem attributed to Moon.

[6] G. BILARDI AND F. P. PREPARATA, *Horizons of Parallel Computing*, Future Tendencies in Computer Science and Applied Mathematics, A. Bensoussan and J.P. Verjus, eds., Lecture Notes in Comput. Sci. 653, Springer-Verlag, Berlin, 1992, pp. 155–174.

[7] J. BRUCK, R. CYPHER, AND C-T. HO, *Fault-tolerant de Bruijn and shuffle-exchange networks*, IEEE Trans. Parallel Distrib. Systems, 5 (1994), pp. 548–553.

[8] J. Bruck, R. Cypher, and C-T. Ho, *Tolerating faults in a mesh with a row of spare nodes*, Theoret. Comput. Sci., 128 (1994), pp. 241–252.

[9] J. Bruck, R. Cypher, and C-T. Ho, *Wildcard dimensions, coding theory and fault-tolerant meshes and hypercubes*, IEEE Trans. Comput., to appear. Also appeared in Proc. 23rd International Symposium on Fault-Tolerant Computing, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 260–267.

[10] J. Bruck, R. Cypher, and C.-T. Ho, *Fault-tolerant meshes and hypercubes with minimal numbers of spares*, IEEE Trans. Comput., 42 (1993), pp. 1089–1104.

[11] J. Bruck, R. Cypher, and D. Soroker, *Tolerating faults in hypercubes using subcube partitioning*, IEEE Trans. Comput., 41 (1992), pp. 599–605.

[12] R. Cypher, *Theoretical aspects of VLSI pin limitations*, SIAM J. Comput., 22 (1993), pp. 356–378.

[13] S. Dutt and J. P. Hayes, *On designing and reconfiguring k-fault-tolerant tree architectures*, IEEE Trans. Comput., C-39 (1990), pp. 490–503.

[14] S. Dutt and J. P. Hayes, *Designing fault-tolerant systems using automorphisms*, J. Parallel Distrib. Comput., 12 (1991), pp. 249–268.

[15] S. Dutt and J. P. Hayes, *Some practical issues in the design of fault-tolerant multiprocessors*, in Proc. 21st International Symposium on Fault-Tolerant Computing, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 292–299.

[16] B. Elspas and J. Turner, *Graphs with circulant adjacency matrices*, J. Combin. Theory, 9 (1970), pp. 297–307.

[17] J. Håstad, F. T. Leighton, and M. Newman, *Fast computations using faulty hypercubes*, in Proc. 21st Annual ACM Symposium on Theory of Computing, ACM, New York, 1989, pp. 251–284.

[18] J. P. Hayes, *A graph model for fault-tolerant computing systems*, IEEE Trans. Comput., C-25 (1976), pp. 875–884.

[19] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, and A. Tsantilas, *Asymptotically tight bounds for computing with faulty arrays of processors*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 285–296.

[20] S.-Y. Kuo and W. K. Fuchs, *Efficient spare allocation for reconfigurable arrays*, IEEE Design and Test, 4 (1987), pp. 24–31.

[21] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[22] F. T. Leighton and C. E. Leiserson, *Wafer scale integration of systolic arrays*, IEEE Trans. Comput., C-34 (1985), pp. 448–461.

[23] T. Leighton, B. Maggs, and R. Sitaraman, *On the fault tolerance of some popular bounded-degree networks*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 542–552.

[24] M. Paoli, W. W. Wong, and C. K. Wong, *Minimum k-Hamiltonian graphs*, II, J. Graph Theory, 10 (1986), pp. 79–95.

[25] A. L. Rosenberg, *The diogenes approach to testable fault-tolerant VLSI processor arrays*, IEEE Trans. Comput., C-32 (1983), pp. 902–910.

[26] V. P. Roychowdhury, J. Bruck, and T. Kailath, *Efficient algorithms for reconfiguration in VLSI/WSI arrays*, IEEE Trans. Comput., C-39 (1990), pp. 480–489.

[27] H. Tamaki, *Construction of the mesh and the torus tolerating a large number of faults*, in Proc. 6th Annual ACM Symposium on Parallel Algorithms and Architectures, Cape May, NJ, 1994, ACM, New York, pp. 268–277.

[28] C. Thompson, *A Complexity Theory for VLSI*, Ph.D. thesis, Dept. of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, 1980.

[29] W. W. Wong and C. K. Wong, *Minimum k-Hamiltonian graphs*, J. Graph Theory, 8 (1984), pp. 155–165.

SIAM J. COMPUT.
Vol. 26, No. 6, pp. 1785–1803, December 1997

012

ON TRANSLATIONAL MOTION PLANNING OF A CONVEX POLYHEDRON IN 3-SPACE*

BORIS ARONOV† AND MICHA SHARIR‡

Abstract. Let B be a convex polyhedron translating in 3-space amidst k convex polyhedral obstacles $A_1, \ldots, A_k$ with pairwise disjoint interiors. The *free configuration space* (space of all collision-free placements) of B can be represented as the complement of the union of the Minkowski sums $P_i = A_i \oplus (-B)$, for $i = 1, \ldots, k$. We show that the combinatorial complexity of the free configuration space of B is $O(nk \log k)$, and that it can be $\Omega(nk\alpha(k))$ in the worst case, where n is the total complexity of the individual Minkowski sums $P_1, \ldots, P_k$. We also derive an efficient randomized algorithm that constructs this configuration space in expected time $O(nk \log k \log n)$.

Key words. combinatorial geometry, computational geometry, combinatorial complexity, convex polyhedra, geometric algorithms, randomized algorithms, algorithmic motion planning

AMS subject classifications. 52B10, 52B55, 65Y25, 68Q25, 68U05

PII. S0097539794266602

1. Introduction.

Let $A_1, \ldots, A_k$ be k closed convex polyhedra in three dimensions with pairwise disjoint interiors, and let B be another closed convex polyhedron which, without loss of generality, is assumed to contain the origin o. In the context of motion planning, B is a "robot" that can only translate in 3-space, and the A_i's are obstacles which B must avoid. Suppose A_i has q_i faces and B has p faces, and put $q = \sum_{i=1}^{k} q_i$. Let

$$P_i = A_i \oplus (-B) = \{a - b \mid a \in A_i,\, b \in B\}$$

be the *Minkowski sum* of A_i and $-B$, for $i = 1, \ldots, k$. Let $\mathcal{P} = \{P_i\}_{i=1}^{k}$ be the resulting collection of these so-called *expanded obstacles* and let $U = \bigcup_{i=1}^{k} P_i$ be their union. As is well known, the complement $\mathcal{C}$ of U (also called the *common exterior* of $\mathcal{P}$) represents the *free configuration space* FP of B, in the sense that, for each point $z \in \mathcal{C}$, the placement of B, for which the reference point O lies at z, does not intersect any of the obstacles A_i, and all such free placements are represented in this manner.

As is well known, the combinatorial complexity (i.e., the number of vertices, edges, and faces on the boundary) of each P_i is at most $\Theta(pq_i)$, so the sum n of the complexities of the expanded obstacles is $O(pq)$. In typical situations, n is usually much smaller than pq. The bounds derived in this paper depend only on n and k and not on p and q.

Our main result is that the combinatorial complexity of the union U, and thus also of FP, is $O(nk \log k)$ and $\Omega(nk\alpha(k))$ in the worst case. This should be compared

* Received by the editors April 27, 1994; accepted for publication (in revised form) December 19, 1995. A preliminary version of this paper appeared as *On translational motion planning in* 3*-space*, in the Proceedings of the 10th ACM Symposium on Computational Geometry, Stony Brook, NY, 1994, ACM, New York, pp. 21–30.

http://www.siam.org/journals/sicomp/26-6/26660.html

† Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201 (aronov@ziggy.poly.edu). The research of this author was supported by NSF grant CCR-92-11541.

‡ School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel (sharir@math.tau.ac.il) and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012. The research of this author was supported by NSF grant CCR-91-22103, by a Max Planck Research Award, by grants from the U.S.–Israeli Binational Science Foundation, the German–Israeli Foundation for Scientific Research and Development, and the Fund for Basic Research administered by the Israeli Academy of Sciences.

to the recent bound obtained by the authors [4, 5] on the combinatorial complexity of the union of any k convex polyhedra in 3-space with a total of n faces; it is shown [5] that the maximum complexity of such a union is $O(k^3+nk\log k)$ and $\Omega(k^3+nk\alpha(k))$. Thus, the convex polyhedra P_i arising in the context of translational motion planning have special properties that yield the above improved bound, without the cubic term of the general bound.

The problem of obtaining sharp bounds for the combinatorial complexity of the union of Minkowski sums, as above, has been open for the past eight years, and has been studied in several papers during this period. It was raised by Kedem et al. [22] (see also [27]), where the two-dimensional version of the problem was successfully tackled. Related work on the three-dimensional case is described in [2, 3, 4, 5, 15, 20, 26]. Except for [4, 5, 20] (and parts of [2]), these papers studied different (though related) problems involving the complexity of the lower envelope or of a single cell in an arrangement of triangles in 3-space. A recent paper of Halperin and Yap [20] analyzes the complexity of the union of Minkowski expansions, as above, for the case where B is a box, and obtains a bound of $O(n^2\alpha(n))$. Our result is general, and thus almost settles this open problem, except for the sublogarithmic gap between our upper and lower bounds; we conjecture that the correct bound is $\Theta(nk\alpha(k))$.

We also consider the algorithmic problem of efficient construction of FP. We obtain an efficient and rather simple randomized algorithm that computes FP in expected time $O(nk\log k\log n)$. The algorithm and its analysis are adapted from a very similar algorithm given in [4, 5] for constructing the union of arbitrary convex polyhedra in 3-space.

The paper is organized as follows. In section 2 we bound the number of components and local minima of $\mathcal{C}$. The analysis of the topology of U, and of several related structures constructed from a family of convex polyhedra, continues in section 3. Section 4 establishes the main result of the paper, namely, the bounds on the combinatorial complexity of the union of the polyhedra, as stated above. Section 5 describes the randomized algorithm for computing the boundary of the union, and its application in the context of translational motion planning, and section 6 concludes with some remarks and open problems.

2. The number of holes of U and local extrema of $\mathcal{C}$. We first simplify the analysis by assuming that the given polyhedra A_i and B are in *general position*, meaning that the coordinates of the vertices of the A_i's and of B are all transcendentals that are algebraically independent over the rationals. In particular, this implies that (a) no point is common to the boundaries of any four distinct expanded polyhedra P_i; (b) no vertex of one expanded polyhedron lies on the boundary of another expanded polyhedron; (c) no two edges of distinct expanded polyhedra meet; and (d) no edge of an expanded polyhedron meets the polygonal curve of intersection of two other expanded polyhedra. Several other implications of the general position assumption will be needed to simplify the proof of Theorem 2.4. As argued in [4, 5], the general position assumption involves no real loss of generality. This is because one can always slightly perturb the given polyhedra, putting them in general position such that the number of vertices of the union that are incident to three distinct polyhedra does not decrease. The number of all other vertices of the union is only $O(nk)$, as follows from Proposition 2.1 below (which also holds when the polyhedra are not in general position). Also see a remark following Theorem 4.2 that discusses this issue.

To simplify some of the subsequent arguments, we modify the family $\mathcal{P}$ so as to make U bounded. To be more precise, we assume that B is bounded, which will

clearly be the case in our motion planning application. (If B is unbounded, then the resulting set $\mathcal{P}$ of unbounded expanded obstacles has the property that the boundary of its union can be regarded as the upper envelope of k concave polyhedral surfaces with a total of n faces. This problem is much simpler and has already been addressed by Huttenlocher, Kedem, and Sharir [21], where a tight $\Theta(nk\alpha(k))$ bound is obtained.) We then truncate all the unbounded obstacles A_i, by intersecting all obstacles with a sufficiently large tetrahedron, and by slightly perturbing the resulting intersections, so as not to violate the general position assumptions. The tetrahedron is chosen sufficiently large so that the union U of the resulting expanded obstacles P_i does not lose any of the bounded boundary features (vertices, edges, and faces) that it had before the truncation. Thus, the process does not reduce the complexity of the union U. In what follows we will therefore assume that U is bounded, so that $\mathcal{C}$ has a single unbounded component.

To establish our upper bound on the complexity of the union U, it clearly suffices to bound the number of vertices of U. Moreover, it suffices to bound the number of vertices of U that are formed by the intersection of the relative interiors of three faces of three distinct polyhedra P_i, because the number of all other vertices, namely, vertices of the original P_i's and intersections of edges of the original P_i's with faces of other expanded obstacles, is only $O(nk)$. This is a consequence of the following easy proposition (see [4, 5]).

PROPOSITION 2.1. *The number of vertices of $P_i \cap P_j$, summed over all $P_i, P_j \in \mathcal{P}$, is $O(nk)$.*

We first bound the number of connected components of the complement $\mathcal{C}$ of U (the common exterior of $\mathcal{P}$). Under the assumptions made above, $\mathcal{C}$ has a unique unbounded component, so it suffices to estimate the number of bounded components of $\mathcal{C}$, which correspond to "holes" in U.

Let an *intersection edge* be an edge of the intersection of the boundaries of two distinct expanded obstacles. Each such edge e meets the boundary of the union U in at most $k-1$ intervals, as it intersects each of the remaining $k-2$ polyhedra in at most one interval, and the complement (within e) of the union of these $k-2$ intervals consists of at most $k-1$ intervals.

We introduce a real parameter $t \in [0,1]$ (we refer to it as "time"), define $P_i(t) = A_i \oplus (-tB)$, for $i = 1, \ldots, k$, and $U(t) = \bigcup_{i=1}^{k} P_i(t)$, and let $\mathcal{C}(t)$ denote the complement of $U(t)$. Note that the general position assumption does not imply that conditions (a)–(d) hold for $\{P_i(t)\}$ for all values of t. It does imply, however, that, at any given t, we can have at most one degenerate contact between the $P_i(t)$'s. Such a contact can be expressed as a polynomial equality in t and in the coordinates of the vertices of the A_i's and of B. Moreover, there is only a finite number of values of t at which such a degeneracy can occur. For example, if $t_0 > 0$ is the first instant of time at which $P_i(t)$ and $P_j(t)$ meet, then $P_i(t) \cap P_j(t)$ must consist of a single point, for otherwise we would have at least two algebraically independent polynomial equalities holding simultaneously, which is forbidden by the general position assumptions. Moreover, such an initial single-point contact of $P_i(t)$ and $P_j(t)$ must occur either between a vertex of one of these polyhedra and a face of the other, or between two edges, one from each polyhedron. Similarly, it is easy to verify that the first contact between three growing obstacles must occur at a single point.

Before continuing, we recall a well-known fact. Let P and Q be two convex polyhedra. Denote by $\mathcal{N}_P$ the *normal diagram* (also known as the *Gaussian diagram*) of P. This is the decomposition of the Gaussian sphere $\mathcal{S}^2$ of orientations into regions,

each consisting of the outward normals of all planes supporting P at some vertex. Each edge of $\mathcal{N}_P$ is an arc of a great circle consisting of the outward normals of all planes supporting P at some edge. Each vertex of $\mathcal{N}_P$ is the outward normal of some face of P. The normal diagram $\mathcal{N}_Q$ of Q is analogously defined. Then the overlay of $\mathcal{N}_P$ and $\mathcal{N}_Q$ is the normal diagram of the Minkowski sum $P \oplus Q$. This implies that the combinatorial structure (i.e., the number of, and incidence relations between, faces of all dimensions) of $P \oplus Q$ is completely determined by the orientations of the faces of P and Q and by their incidence structures. In particular, the combinatorial structure of $P_i(t) = A_i \oplus (-tB)$ is the same for all $t > 0$. This implies the following property. For $t \in [0, 1]$, let $s(t)$ denote the Minkowski sum of either a fixed edge of A_i and $(-t)$ times a fixed vertex of B, or of a fixed vertex of A_i and $(-t)$ times a fixed edge of B. If $s(t_0)$ is an edge of $P_i(t_0)$, for some $t_0 \in (0, 1]$, then $s(t)$ is an edge of $P_i(t)$ for all $t \in (0, 1]$.

We want to keep track of the number of holes of $U(t)$ as t increases from 0 to 1. At $t = 0$, the number of holes of $U(t)$ is 0. As t increases, $\mathcal{C}(t)$ shrinks, so the number of holes can *increase* only when a component of $\mathcal{C}(t)$ is split into two components.

LEMMA 2.2. *A component of $\mathcal{C}(t)$ can split into two components only at times t when an edge $s(t)$ of some $P_\ell(t)$ meets an intersection edge $e(t)$ of two other polyhedra, $P_i(t)$, $P_j(t)$, at some point v, so that the intersection of $\mathcal{C}(t)$ with a small neighborhood of v is disconnected.*

Proof. We use the observation of de Berg, Matoušek, and Schwarzkopf [6], which states that the number of connected components of $\mathcal{C}(t)$ can increase only at times t, at which the elements of some new triple growing polyhedra intersect for the first time. As noted above, the general position assumptions guarantee that such a triple, $P_i(t)$, $P_j(t)$, $P_\ell(t)$, intersects in a single point. Moreover, the intersection must occur on the boundary of $\mathcal{C}(t)$ and cause a local disconnection of $\mathcal{C}(t)$, or else no change in the topological structure of $\mathcal{C}(t)$ would take place. (Note that the above conditions are only necessary, but not sufficient, for the number of components of $\mathcal{C}(t)$ to increase.) The intersection in question cannot occur at a point in the relative interior of three faces, one on each of these three polyhedra, for then $P_i(t') \cap P_j(t') \cap P_\ell(t')$ would also be nonempty for t' slightly smaller than t. This and the general position assumption are easily seen to imply the lemma. ∎

Suppose that such a configuration does indeed arise at some time t, and suppose that $e(t)$ is formed by the intersection of a face of $P_i(t)$ and a face of $P_j(t)$. The edge $s(t)$ intersects both $P_i(t)$ and $P_j(t)$ at two respective intervals $s(t) \cap P_i(t)$ and $s(t) \cap P_j(t)$, which have a common endpoint and which, by Lemma 2.2, have disjoint relative interiors.

LEMMA 2.3. *If at time t the intervals $s(t) \cap P_i(t)$ and $s(t) \cap P_j(t)$ intersect, then $s(t') \cap P_i(t')$ and $s(t') \cap P_j(t')$ intersect for all $t' > t$.*

Proof. Let $v(t)$ denote some common point of the intervals $s(t) \cap P_i(t)$, $s(t) \cap P_j(t)$. Recall that $s(t)$ is an edge of $A_\ell \oplus (-tB)$. Assume first that $s(t)$ is the Minkowski sum of a vertex a of A_ℓ and $(-t)$ times an edge of B, whose endpoints are b, b'. Since $v(t) \in s(t)$, there exists $\lambda \in [0, 1]$ such that $v(t) = a - t(\lambda b + (1 - \lambda)b')$. We are given that $v(t) \in P_i(t)$, which means that $v(t) = c - tb''$, for some $c \in A_i$ and $b'' \in B$. We claim that the point $v(t') = a - t'(\lambda b + (1 - \lambda)b')$, with the same λ as above, lies in $P_i(t')$, for all $t' > t$ (and, by definition, $v(t') \in s(t')$; see the discussion preceding Lemma 2.2). Indeed, if we put $b_\lambda = \lambda b + (1 - \lambda)b'$, then

$$v(t') = a - tb_\lambda - (t' - t)b_\lambda = c - tb'' - (t' - t)b_\lambda = c - t' \left[\frac{t}{t'} b'' + \frac{t' - t}{t'} b_\lambda \right],$$

which clearly belongs to $A_i \oplus (-t'B)$, whenever $t' \geq t$. An analogous argument shows that $v(t') \in A_j \oplus (-t'B)$, whenever $t' \geq t$. This implies the asserted claim. The case where $s(t)$ is the Minkowski sum of an edge of A_ℓ and $(-t)$ times a vertex of B is handled in a similar manner, as the reader can easily verify. This completes the proof of the lemma. □

Lemmas 2.2 and 2.3 imply that, for each t at which a connected component of $\mathcal{C}$ is split into two components, there exist two polyhedra, $P_i(t)$, $P_j(t)$, and an edge $s(t)$ of another polyhedron $P_\ell(t)$, such that the two intervals $s(t) \cap P_i(t)$ and $s(t) \cap P_j(t)$ meet at a common endpoint that lies on $\partial\mathcal{C}(t)$ and continue to overlap for all $t' > t$. Thus, the number of such "hole-splitting" events, involving the same edge $s(t)$ of any of the expanded polyhedra, is at most $k-2$, so the total number of holes that the union can have is at most $O(nk)$. That is, we have shown the following theorem.

THEOREM 2.4. *The number of connected components of the common exterior $\mathcal{C}$ of Minkowski sums of k polyhedra, as above, is $O(nk)$.*

Remark. We do not know if this bound is tight, as the best lower bound we can construct is $\Omega(k^2)$. Assume for simplicity that $k > 2$ is even. A configuration yielding such a bound consists of $k/2-1$ line obstacles of the form $z = 0$, $x = i$, for $i = 1, \ldots, k/2-1$, of $k/2-1$ line obstacles of the form $z = 0.1$, $y = i$, for $i = 1, \ldots, k/2-1$, and of two plane obstacles $z = \pm 1/2$. The robot is an axis-parallel cube with side length $2/3$.

Let S be any closed polyhedral set in 3-space in general position. (In this paper we consider only polyhedral sets bounded by a finite number of edges, vertices, and faces.) For a point p, let a *neighborhood* of p in S be the intersection of S with a ball centered at p, whose radius is smaller than the distance from p to any edge, face, or vertex of S not incident to p. A point p of S is a *local minimum* if it is a point of smallest z-coordinate in some neighborhood of p in S. (Under the general position assumptions, no face or edge can be parallel to the xy-plane, so p will be the only point in its neighborhood with this z-coordinate.) A *local maximum* is similarly defined. The following theorem is a consequence of Theorem 2.4.

THEOREM 2.5. *The number of local maxima and minima of the closure $\bar{\mathcal{C}}$ of $\mathcal{C}$ is $O(nk)$.*

Proof. We prove the claim only for the number of local minima of $\bar{\mathcal{C}}$, since the treatment of local maxima is fully symmetric. Any bounded convex component K of $\bar{\mathcal{C}}$ has exactly one local minimum (which is also the global minimum of K), because K does not have a lowest horizontal edge, as implied by our general position assumption. Hence, the number of such minima is $O(nk)$ by Theorem 2.4. This also applies to components of $\bar{\mathcal{C}}$, all of whose horizontal cross sections are connected, because such a component has at most one local (and global) minimum as well. So we will proceed to bound the number of local minima of those components that have at least one disconnected horizontal cross section; we refer to these components as *interesting components*. We will in fact prove that the number of such minima is only $O(k^2)$.

Let $\mathcal{C}^*$ denote the union of all interesting components of $\bar{\mathcal{C}}$. Sweep $\mathcal{C}^*$ with a plane π which is parallel to the xy-plane, move upwards, i.e., in the positive z-direction, and keep track of the number I of connected components of $\mathcal{C}^* \cap \pi$. This number is initially 1 (when π is below all vertices of $\mathcal{C}^*$). (Actually, it is possible for the outer component to be uninteresting, in which case the number starts off at 0.) I increases by 1 when π sweeps through a local minimum of $\mathcal{C}^*$ or when a connected component of $\mathcal{C}^* \cap \pi$ splits into two subcomponents; I decreases by 1 when π sweeps through a local maximum of $\mathcal{C}^*$ or when two components of $\mathcal{C}^* \cap \pi$ merge into a single component.

(The general position assumption implies that, at any given z, only two components can merge into a component of $\mathcal{C}^* \cap \pi$, and similarly a component cannot split into more than two subcomponents.) The events at which components may merge or split occur only when π sweeps, repsectively, through the topmost or bottommost vertex of the intersection of some pair P_i, P_j of expanded obstacles, for some $i \neq j$. Indeed, it is easily checked that, under the assumption of general position, splitting always occurs when a vertex of some polygon $\pi \cap P_i$ meets an edge of another polygon $\pi \cap P_j$ in such a way that the two polygons are disjoint before the event and meet in a small triangle after the event. Thus P_i and P_j are disjoint immediately below the critical value of z and meet immediately afterwards, implying our claim that the critical position of π passes through the lowest point of $P_i \cap P_j$. Merges are analyzed in an entirely symmetric fashion. To summarize, the number of splits and merges of cross-sectional components is bounded by $2\binom{k}{2} = O(k^2)$. It remains to prove that this implies that the number of local minima is also $O(k^2)$.

Consider the following dynamic scheme for assigning weights to each component of $\mathcal{C}^* \cap \pi$. When π sweeps through a local minimum point of $\mathcal{C}^*$, a new component of $\mathcal{C}^* \cap \pi$ is created and is assigned weight -1. When two components of $\mathcal{C}^* \cap \pi$ merge, the weight assigned to the new component is 2 plus the sum of the weights of the merged components. When a component shrinks and disappears, its final weight is added to a global count. When a component is split into two subcomponents, each of them is assigned weight $1 + \frac{w}{2}$, where w is the weight of the split component.

We claim that, at any given time during the sweep, the weight of any component of $\mathcal{C}^* \cap \pi$ is always at least -1, and the weight of a component that was formed by a splitting or merging operation is nonnegative. Both claims are easy to prove by induction on the sweep events. Now suppose that, at some point during the sweep, there are s local minima of $\mathcal{C}^*$ below π, and that the number of component splittings and mergings below π is N (which is at most $O(k^2)$). Then the total weight of the components of $\mathcal{C}^* \cap \pi$, plus the value of the current global count (i.e., the sum of the final weights of all cross sections of interesting components of $\mathcal{C}^*$ that have terminated below π), is easily seen, by induction on the sweep events, to be $2N - s$. As argued above, the value of the global count is always nonnegative. When the sweep plane passes the topmost vertex of $\mathcal{C}^*$, the cross section $\mathcal{C}^* \cap \pi$ has just one component, necessarily of nonnegative weight, which thus implies that, at this final stage, $s \leq 2N = O(k^2)$. It follows that the number of local minima of $\mathcal{C}^*$ contained in interesting components is $O(k^2)$. Adding the number of local minima in non-interesting components of $\bar{\mathcal{C}}$, we obtain the bound asserted in the theorem. □

3. On the structure of levels in $\mathcal{A}(\mathcal{P})$. Define the *arrangement* $\mathcal{A}(\mathcal{P})$ of the collection $\mathcal{P}$ of the expanded obstacles to be the decomposition of space into vertices, edges, faces, and three-dimensional cells, induced by the faces of the polyhedra of $\mathcal{P}$; for more details on arrangements, see [2, 14, 29]. For each $s = 1, \ldots, k$, let

$$U^{(s)} = \bigcup \{P_{j_1} \cap \cdots \cap P_{j_s} \,|\, 1 \leq j_1 < j_2 < \cdots < j_s \leq k\}.$$

Furthermore, put $\mathcal{C}^{(s)} = \mathbb{R}^3 \setminus U^{(s)}$. We refer to $\partial\mathcal{C}^{(s+1)} = \partial U^{(s+1)}$ as the *sth level* of the arrangement $\mathcal{A}(\mathcal{P})$. We also denote by $\bar{\mathcal{C}}^{(s)}$ the closure of $\mathcal{C}^{(s)}$. Note that $U^{(1)} = U$ is just the union of the polyhedra of $\mathcal{P}$ and that $\mathcal{C}^{(1)} = \mathcal{C}$ is their common exterior. These definitions are illustrated in the planar case in Figure 1.

In this section we prove some geometric and topological properties of $U^{(2)}$ and $U^{(3)}$ that will be used in the proof of our main theorem (Theorem 4.2 in section 4).

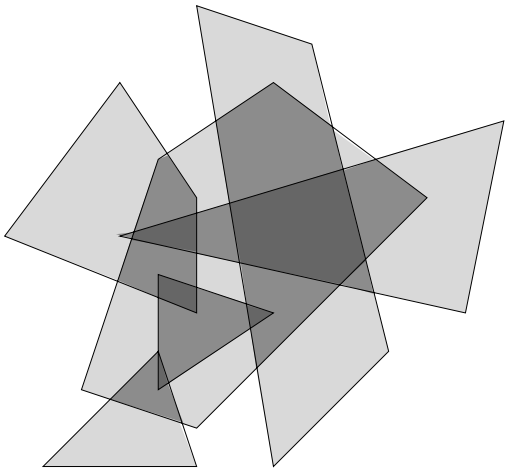


FIG. 1. *A family of convex polygons*; $U^{(1)} \setminus U^{(2)}$, $U^{(2)} \setminus U^{(3)}$, *and* $U^{(3)}$ *are shaded successively darker.*

LEMMA 3.1. *The number of local minima and the number of connected components of* $U^{(3)}$, *for any collection of* k *polyhedra with a total of* n *faces in general position in* 3*-space, are bounded by the number of local maxima of* $\bar{\mathcal{C}}^{(1)}$ *plus* $O(k^2)$.

Proof. $U^{(3)}$ is the union of triple intersections of polyhedra in $\mathcal{P}$. Thus a local minimum of $U^{(3)}$ is necessarily a local minimum of one of these triple intersections. (This immediately bounds the number of local minima by $\binom{k}{3}$.) Since the number of all pairwise intersections of polyhedra in $\mathcal{P}$ is only $\binom{k}{2}$, it suffices to consider only those minima that are formed as intersections of three distinct polyhedron boundaries. By the general position assumption, such an intersection must be the unique common point v of the relative interiors of three faces of three distinct polyhedra in $\mathcal{P}$. Thus, if v is a local minimum in $U^{(3)}$ of this kind, then $U^{(3)}$ near v has the form of an octant O, formed by the three planes containing the faces incident to v, with v at its minimum. Thus, near v, $\bar{\mathcal{C}}^{(1)}$ coincides with the octant opposite to O, which has v as a local maximum. Hence, any local minimum of $U^{(3)}$ that is formed as the triple intersection of three distinct polyhedron boundaries corresponds to a local maximum of $\bar{\mathcal{C}}^{(1)}$. Since any bounded connected component of $U^{(3)}$ has a local minimum, and since we have assumed U and thus also $U^{(s)}$, for any s, to be bounded, it follows that the number of components of $U^{(3)}$ also satisfies the bound asserted in the lemma. □

Applying Theorem 2.5, we obtain the following corollary.

COROLLARY 3.2. *If* $\mathcal{P}$ *is a collection of polyhedra with a total of* n *faces, which arise as Minkowski expansions of a collection of* k *convex polyhedra with pairwise-disjoint interiors by another convex polyhedron, then the number of local minima and the number of connected components of* $U^{(3)}$ *is* $O(nk)$.

Proof. Lemma 3.1 is applicable here even though the expanded polyhedra are not in fully general position. Nevertheless, if we assume that the original A_i's and B are in general position, then the assumptions made in the proof of Lemma 3.1 do hold, as is easily checked, and the corollary thus follows. □

Next we study the topological structure of $U^{(2)}$. We begin by deriving a topological property of general polyhedral sets. Let S be a closed polyhedral set in $\mathbb{R}^3$

in general position; in particular, no two vertices of S have the same z-coordinate. Note that a neighborhood of a point p in S, as defined in the previous section, is star-shaped with respect to p, so its intersection with a horizontal plane through p is also star-shaped and thus connected.

A *merge point* of S is a point $p \in S$ such that, for any neighborhood of p, any horizontal plane lying below p and sufficiently close to it intersects the neighborhood in a disconnected set. (It is easily checked that p must be a vertex of S.) The number of components of such an intersection approaches some limit as the horizontal plane approaches p. We define the *merge number* $m(p)$ of p to be the limit number of connected components in this intersection, less 1. Since every neighborhood of p is a scaled copy of any other neighborhood of p, $m(p)$ is independent of the choice of the neighborhood. We can naturally extend this definition to any nonmerge point p' by putting $m(p') = 0$. We will apply our analysis to $S = U^{(2)}$, so in this case $m(p) \leq 2$ for any point in the set, as is easily implied by the general position assumption on $\mathcal{P}$.

The *first Betti number* of S, denoted $\beta_1(S)$, is the rank of the first singular homology group of S [17]. Informally, it is the number of "linearly independent" homotopy classes of closed cycles in S, where each class consists of all cycles homotopic within S to some given cycle and not contractible to a single point within S.

PROPOSITION 3.3. *The first Betti number $\beta_1(S)$ of any compact polyhedral set S in $\mathbb{R}^3$ in general position does not exceed the sum of the merge numbers of its vertices.*

The proof of this proposition is somewhat technical, and we give it in the appendix.

LEMMA 3.4. *For a collection of k arbitrary convex polyhedra in $\mathbb{R}^3$ in general position, with a total of n faces, $\beta_1(U^{(2)})$ is at most proportional to k^2 plus the number of local minima of $U^{(3)}$. Hence, for collections of Minkowski expansions, as above, this number is $O(nk)$.*[1]

Proof. Applying Proposition 3.3 to $S = U^{(2)}$, we conclude that it suffices to bound the number of merge points of $U^{(2)}$. This is because the merge number of any point of $U^{(2)}$ is at most 2, as follows from the general position assumption (and as already noted above). Let $v \in \partial U^{(2)}$ be a merge point; then v must be a vertex of $\mathcal{A}(\mathcal{P})$. We may assume that v is a vertex formed by the transversal intersection of some three faces F_i, F_j, F_ℓ belonging to three respective distinct polyhedra P_i, P_j, P_ℓ. Indeed, any other vertex is either a vertex of some P_i or of some $P_i \cap P_j$, for $i, j = 1, \ldots, k$. No vertex of the former type can be a merge point of any $U^{(s)}$, as is easily seen. A vertex of the latter type, on the other hand, can be a merge point only if it is the bottommost point of $P_i \cap P_j$, as is readily verified using the general position assumption. Thus, the number of these merge points is at most $\binom{k}{2}$; in fact, the total number of merge points of this form, over all sets $U^{(s)}$, is at most $\binom{k}{2}$.

The planes π_i, π_j, π_ℓ, containing, respectively, the faces F_i, F_j, F_ℓ, partition 3-space into eight octants; in a neighborhood of v, exactly one of these octants, O^+, is contained in all three polyhedra P_i, P_j, P_ℓ, three octants are contained in exactly two of these polyhedra, three other octants are contained in exactly one of these polyhedra, and one octant, O^-, is disjoint from all three polyhedra; the octants O^+ and O^- lie opposite each other. Note that v is not contained in the interior of any polyhedron of $\mathcal{P}$, for otherwise $U^{(2)}$, in a neighborhood of v, contains all octants around v except possibly for O^-, in which case v cannot be a merge point of $U^{(2)}$. We conclude that, near v, $U^{(2)}$ consists of O^+ and of its three adjacent octants.

[1] Again, in the case of Minkowski expansions, the polyhedra are not in fully general position, but they satisfy the assumptions made in the relevant proofs, so the corollary does apply.

We claim that v is the bottommost point of O^+. Indeed, otherwise, by the general position assumption, a horizontal plane π passing just below v would meet O^+. Let O', O'', O''' be the octants adjacent to O^+, each covered, locally near v, by two of the polyhedra P_i, P_j, P_ℓ. Then, near v the cross section $U^{(2)} \cap \pi$ coincides with

$$((O^+ \cup O') \cap \pi) \cup ((O^+ \cup O'') \cap \pi) \cup ((O^+ \cup O''') \cap \pi).$$

Each of $O^+ \cup O'$, $O^+ \cup O''$, $O^+ \cup O'''$ is a convex set, namely, a dihedral wedge. In particular, $U^{(2)} \cap \pi$ near v is the union of three convex sets, all containing the non-empty set $O^+ \cap \pi$, and thus is connected, contradicting the assumption that v is a merge point, and thus establishing the claim. In other words, we have shown that if v is a merge point of $U^{(2)}$ then π must avoid O^+, i.e., O^+ must lie fully above the horizontal plane passing through v, or, equivalently, v must be a local minimum of $U^{(3)}$. This completes the proof of the lemma. □

4. The complexity of the union. Having all this machinery available, we now turn to the analysis of the complexity of U. We apply the analysis technique of [5] (an earlier and somewhat weaker version of that analysis is given in [4]). Here we go quickly through its main steps; we refer the reader to [5] for full details. Let $C(k, n)$ denote the total number of vertices of U incident to three distinct polyhedra of $\mathcal{P}$, maximized over all collections $\mathcal{P}$ of k Minkowski sums, as above, with a total of n faces, in general position. As argued in [4, 5], this number is equal to the total number of vertices, edges, and faces, up to a constant multiplicative factor and an additive term of $O(kn)$. The analysis scheme of [5] yields the following recurrence for $C(k, n)$:

$$(k - 5/3)C(k, n) \leq \sum_{i=1}^{k} C(k - 1, n - n_i) + O(D(k, n)), \tag{1}$$

where n_i is the number of faces of P_i, for $i = 1, \ldots, k$, and where $D(k, n)$ is the maximum, taken over all collections $\mathcal{P}$ as above, of the quantity $\sum_f w(f)$, with the sum ranging over all faces f of $\partial U^{(2)}$; here $w(f) = \max(2e_f - 3, 0)$ and e_f is the number of edges on ∂f that lie on the boundary of the only polyhedron containing f in its interior.

We next estimate $D(k, n)$ exactly as in [4, 5]. The inductive analysis technique used in these papers leads to the recurrence

$$(k - 2)D(k, n) = \sum_{i=1}^{k} D(k - 1, n - n_i) + O(nk + \mathcal{Q}(k, n)), \tag{2}$$

where $\mathcal{Q}(k, n)$ is the number of "special quadrilaterals" of the following form, maximized over all collections $\mathcal{P}$ as above. Such a quadrilateral Q is schematically depicted in Figure 2; it is defined by a triple $(F(Q), P'(Q), P''(Q))$, where $P' = P'(Q)$, $P'' = P''(Q)$ are distinct polyhedra of $\mathcal{P}$, and $F = F(Q)$ is a face of another polyhedron, $P(Q)$, of $\mathcal{P}$; the intersection $P' \cap P'' \cap F$ is the quadrilateral Q, which is assumed to be disjoint from all other polyhedra; the four vertices of Q are vertices of the union U, two opposite edges of Q are contained in $\partial P' \cap F$ and in the interior of only P'', and the other two edges of Q are contained in $\partial P'' \cap F$ and in the interior of only P'. In other words, if we remove P' and P'' from $\mathcal{P}$ and replace them by $P' \cap P''$, then ∂Q appears on the boundary of the union of the modified collection, and the union of all polyhedra in $\mathcal{P}$, except for P' and P'', contains Q on its boundary. In the notation

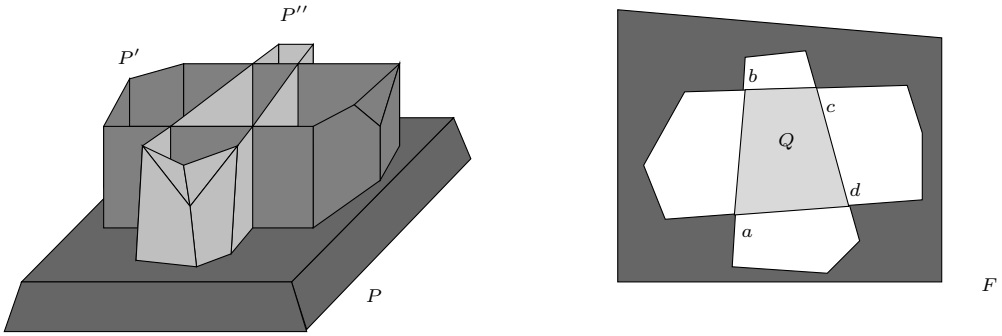


FIG. 2. *A special quadrilateral Q.*

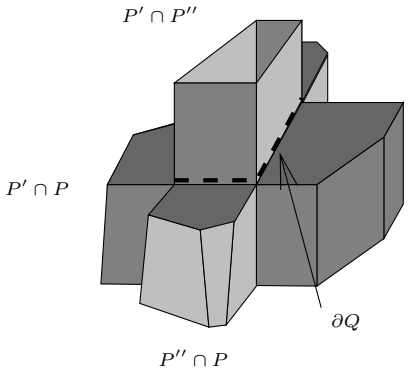


FIG. 3. *$U^{(2)}$ near a special quadrilateral Q.*

of section 3, Q is a special type of face of $\partial U^{(3)}$. The structure of $U^{(2)}$ locally near Q is schematically depicted in Figure 3.

In the case of general convex polyhedra, the authors give in [4, 5] a bound of $O(k^3 + nk)$ for $\mathcal{Q}(k, n)$ and show that it is tight in the worst case. However, we show here that the special properties of $\mathcal{A}(\mathcal{P})$ in the case of Minkowski sums, as established in section 3, lead to an improved bound of only $O(nk)$. Plugging this bound into the recurrences (1) and (2), we obtain, following the analysis of [5], $D(k, n) = O(nk \log k)$ and $C(k, n) = O(nk \log k)$, which thus completes the proof of the upper bound of our main result.

We actually prove the following stronger result.

LEMMA 4.1. *For collections $\mathcal{P}$ of arbitrary convex polyhedra in general position, the number of special quadrilaterals is at most proportional to nk plus the number of local minima of $U^{(3)}$. Hence, for Minkowski expansions this number is $O(nk)$.*

Proof. Let Q be a special quadrilateral defined by $(F(Q), P'(Q), P''(Q))$, as above. The boundary ∂Q of Q is necessarily a bounding cycle of a connected component of $\Sigma(Q) \equiv \partial(P'(Q) \cap P''(Q)) \cap \partial U^{(2)}$. Following an argument similar to that of [4, 5], we observe that the overall number of special quadrilaterals Q whose boundaries are contractible to a point in the corresponding sets $\Sigma(Q)$ is just $O(nk)$, since in the process of contracting ∂Q one has to encounter a vertex of $P'(Q) \cap P''(Q)$, and it is easily checked that no vertex of this form is charged by more than one contractible quadrilateral for fixed P' and P''. The claim now follows from Proposition 2.1. We

thus need to consider only *nontrivial* quadrilaterals, namely, those whose boundaries are not contractible in the above fashion.

Let Q be such a nontrivial special quadrilateral. By definition, Q does not meet the interior of $U^{(3)}$ but is fully contained in $U^{(2)}$. Moreover, $\partial Q \subset \partial U^{(2)}$ (see Figure 3). "Cut" $U^{(2)}$ along Q (or, rather, shift Q slightly away from $P(Q)$ and then cut $U^{(2)}$ along the shifted quadrilateral). We apply this cutting procedure to every nontrivial special quadrilateral, in some arbitrary order. Since, by definition, the relative interiors of any pair of distinct special quadrilaterals are disjoint, it follows that the cuts do not interfere with each other.

Each cut of $U^{(2)}$, performed along some nontrivial special quadrilateral Q, has one of two possible effects: either the first Betti number of $U^{(2)}$ decreases by 1, or the number of connected components of $U^{(2)}$ increases by 1, according to, respectively, whether points of $U^{(2)}$ lying on the two sides of Q sufficiently near Q can or cannot be connected by a path that avoids Q in the current version of $U^{(2)}$. The number of cuts is thus proportional to the increase in the number of connected components of $U^{(2)}$ plus the decrease in the first Betti number of $U^{(2)}$, as effected by the cuts. Since none of the cuts can *increase* the first Betti number of $U^{(2)}$, it follows by Lemma 3.4 that the latter quantity (the decrease in $\beta_1(U^{(2)})$) is bounded by $O(k^2)$ plus the number of local minima of $U^{(3)}$.

To estimate the former quantity, consider a cut performed along a special quadrilateral Q, defined by the triple $(F(Q), P'(Q), P''(Q))$, which increases the number of components of the current version of $U^{(2)}$. Since Q is nontrivial, "dragging" Q along $\partial(P'(Q) \cap P''(Q))$ in either direction encounters a third polyhedron, and thus also a distinct component of $U^{(3)}$. To be more precise, let $U_1^{(2)}$ and $U_2^{(2)}$ be the components obtained by cutting the current version of $U^{(2)}$ along Q. One of the new components of $U^{(2)}$, say $U_1^{(2)}$, contains, near Q, points belonging to $P(Q) \cap P'(Q) \cap P''(Q)$, so $U_1^{(2)}$ contains a component of $U^{(3)}$. The other component of $U^{(2)}$, $U_2^{(2)}$, is bounded near Q by the connected portion K of $\Sigma(Q)$ incident to Q. It is clear that K must also have been incident (before any cuts were made) to some third polyhedron, for otherwise K would have been homeomorphic to a disk whose boundary corresponds to ∂Q, and Q would then have been a trivial quadrilateral, contractible to a point in $K \subset \Sigma(Q)$. There are now two subcases to consider.

(i) The component $U_2^{(2)}$ also contains a component of $U^{(3)}$. (Note that no component of $U^{(3)}$ is ever split by the cuts, because all special quadrilaterals are disjoint from the interior of $U^{(3)}$, so the cuts, performed along slightly shifted copies of the quadrilaterals, are thus disjoint from $U^{(3)}$.) The total increase in the number of components of $U^{(2)}$ formed by cuts of this kind cannot exceed the number of components of $U^{(3)}$. This is because each such cut disconnects in $U^{(2)}$ two components of $U^{(3)}$ that were connected in $U^{(2)}$ before the cut was made. Hence the number of these cuts is bounded by the number of local minima of $U^{(3)}$.

(ii) The component $U_2^{(2)}$ does not contain a component of $U^{(3)}$. The corresponding boundary K, as defined above, was incident, before any cuts were made, to another component of $U^{(3)}$. It follows that this subcase occurs because previous cuts, along other special quadrilaterals incident to K, have separated K from all other adjacent components of $U^{(3)}$. In this case, we charge the current cut to one of these preceding cuts. It is easily checked that no cut is charged in this manner more than once. This implies that the number of special quadrilaterals in this subcase is at most equal to the number of nontrivial special quadrilaterals of the preceding types (those decreasing $\beta_1(U^{(2)})$ and those appearing in case (i) above).

This completes the proof for collections of general convex polyhedra. In the case of Minkowski expansions, as above, Corollary 3.2 implies that the number of special quadrilaterals is $O(nk)$. □

We have thus shown that, in the case of Minkowski expansions, $\mathcal{Q}(k,n) = O(nk)$, which in turn completes the analysis of $C(k,n)$ and yields the upper bound in the following main result of the paper.

THEOREM 4.2. *Let $A_1, \ldots, A_k$ be k convex polyhedra in 3-space with pairwise disjoint interiors, and let B be another convex polyhedron. The combinatorial complexity of the union of the Minkowski sums $A_i \oplus (-B)$, for $i = 1, \ldots, k$, is $O(nk \log k)$, where n is the overall complexity of the individual Minkowski sums. In the worst case, this complexity can be $\Omega(nk\alpha(k))$.*

Remark. The preceding argument assumes general position of the polyhedra A_i and B. Nevertheless, the theorem also holds for collections of polyhedra not in general position, as can be argued using a perturbation scheme [4, 5]. The only delicate part of this reasoning is in the handling of pairs of obstacles whose boundaries overlap. This is done as follows. Shrink each A_i homothetically by a sufficiently small amount. This may cause some features of the union U to disappear. However, each vertex of U that is formed by the transversal intersection of three faces of three distinct P_i's appears as a (slightly perturbed) vertex of the new union. Since the number of all other vertices of U is only $O(nk)$ (see the paragraph preceding Proposition 2.1), it follows that the upper bound of Theorem 4.2 also applies in degenerate configurations of this form.

Proof of the lower bound. We make use of a planar construction, given in [1], of k convex polygons with a total of n edges, such that their union has $\Omega(n\alpha(k))$ edges and vertices. Additionally, the polygons can be arranged so that their union is star-shaped, say with respect to the origin, and at least some fixed fraction of its vertices are visible from the point $(0, +\infty)$, in the strong sense that there exists some fixed angle $\beta > 0$ (independent of k and n), so that, for any such visible vertex v, the wedge whose apex is v, whose bisecting ray is parallel to the positive y-axis, and whose angle is β, does not meet any of the polygon interiors.

Without loss of generality, assume that k is even and $n \geq 4.5k$. We start our three-dimensional construction with a set of $k/2$ convex polygons in xy-plane with $n - 3k$ edges altogether, so that their union U' has $\Omega(n\alpha(k))$ vertices visible from $(0, +\infty, 0)$ in the above strong sense. By scaling, we may assume that the entire construction is contained in the unit disk about the origin in the xy-plane. Now slightly shift and expand the polygons in the z-direction, each by a different amount, to produce pairwise-disjoint flat and thin convex prisms, all contained in, say, the slab $|z| \leq 0.1$; see Figure 4. The second set of $k/2$ polyhedra consists of points $(0, M, i)$, for $i = 1, \ldots, k/2$ (or, rather, tiny tetrahedra centered around these points), where $M \gg k$ is an appropriate parameter. This gives us a collection $\{A_i\}_{i=1}^{k}$ of k pairwise-disjoint convex polyhedra with a total of n faces. The polyhedron B is a tetrahedron with vertices $(0, 0, \pm M')$ and $(\pm M', M, 0)$, where $k \ll M' < M/4$ is another parameter, chosen so that the dihedral angles of B at its horizontal and vertical edges are both equal to some $\beta' < \beta$; note that, by the choice of M', we have $\tan \frac{\beta'}{2} < 1/4$, an inequality that will be needed below. See Figure 4 for an illustration.

Let v be a vertex of U' visible from $(0, +\infty, 0)$ in the above sense. By construction, we can place B so that its vertical edge e_v touches the two prisms corresponding to the two polygons whose boundaries intersect at v; moreover, we can slide B vertically upwards and downwards, by a total distance of close to $2M'$, so that e_v maintains

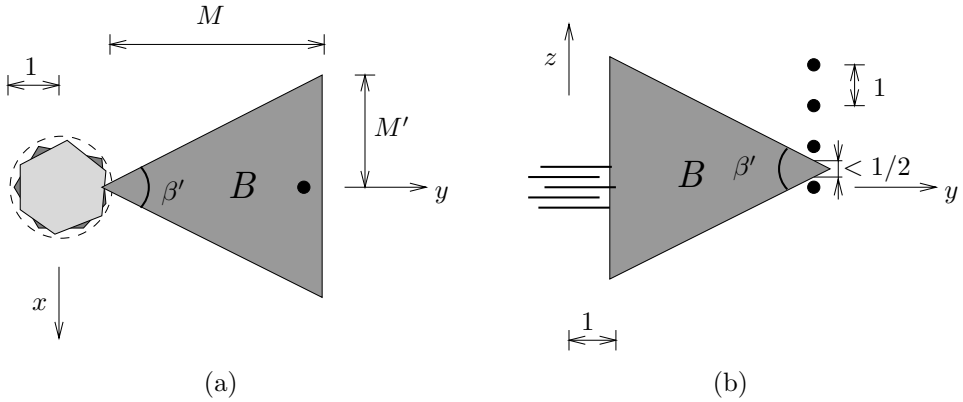


FIG. 4. *The lower bound construction, not to scale:* (a) *a view from above*; (b) *a side view.*

these two contacts, while the interior of B remains disjoint from any of the shifted prisms. It is easily seen that, for an appropriate choice of M, independently of the choice of v, the boundary of B will meet each of the tiny tetrahedra A_i around the points $(0, M, i)$ during the vertical motion. Moreover, our choice of parameters also implies that the intersection of B with the vertical line $x = 0, y = M$ has length less than $2\tan\frac{\beta}{2} < 1/2$, so when B touches one of these A_i's, its interior remains disjoint from all the other polyhedra A_i; see Figure 4(b) for an illustration. In other words, each of the $\Omega(n\alpha(k))$ vertices of U' that is visible from $(0, +\infty, 0)$ gives rise to $\Omega(k)$ placements of B where it makes three contacts with the A_i's, while its interior remains disjoint from all these polyhedra. Since each of the resulting $\Omega(nk\alpha(k))$ placements of B corresponds to a vertex of the union of the expanded polyhedra $\bigcup_{i=1}^{k} A_i \oplus (-B)$, the lower bound of the theorem follows. □

COROLLARY 4.3. *The combinatorial complexity of the free configuration space of a convex polyhedron B, translating in 3-space amidst a collection of k convex obstacles $A_1, \ldots, A_k$ having pairwise disjoint interiors, is $O(nk \log k)$ and can be $\Omega(nk\alpha(k))$ in the worst case, where n is the overall complexity of the Minkowski sums $A_i \oplus (-B)$ for $i = 1, \ldots, k$.*

Remarks. (1) Returning to the parameters p and q that count, respectively, the number of faces of B and of all the A_i's together, we can state the upper bound of Theorem 4.2 and Corollary 4.3 as $O(pqk \log k)$ and the lower bound as $\Omega(qk\alpha(k))$. Another trivial lower bound is $\Omega(pq)$. It remains an open problem to obtain a sharper calibration of the lower bound in terms of the parameters p, q, and k.

(2) Our proof of Theorem 4.2 makes use of the fact that the P_i's are Minkowski sums only in the proof of Theorem 2.4. Hence, our analysis also applies to any collection of k general convex polyhedra with a total of n faces, with the property that, for any subset of r of these polyhedra, the number of components of the complement of their union is $O(rm)$, where m is the total number of faces of those r polyhedra. We pose the open problem of finding other natural examples of collections of convex polyhedra with this property.

5. Efficient construction of the union and its motion planning application. Next we derive an efficient randomized algorithm for constructing the union U of a collection $\mathcal{P}$ of expanded polyhedra, as above. The input to the algorithm consists of the original polyhedra A_i and B, so we first compute the individual Minkowski

sums $P_i = A_i \oplus (-B)$ for $i = 1, \ldots, k$. This computation can be done in several ways, the most efficient of which is by using the technique of Guibas and Seidel [19]. Each P_i can be constructed in time $O(p + q_i + n_i)$, where n_i is the complexity of P_i. Thus the cost of this stage is $O(pk + q + n)$.

The main algorithm is essentially identical to the one given in [4, 5] for the case of general polyhedra; for the sake of completeness, here is a very brief review of the algorithm.

We first compute all the pairwise intersections $P_i \cap P_j$, for $1 \le i < j \le k$, in time $O(nk)$ [9]. In additional $O(nk)$ time, we also extract, for each face F of a polyhedron $P_i \in \mathcal{P}$, the collection $\mathcal{Q}_F$ of the convex polygons $Q_j = F \cap P_j$, for $j \ne i$. The set $U_F = F \setminus \bigcup_{j \ne i} Q_j$ is the portion of F that appears on ∂U, so the algorithm computes the sets U_F, over all faces F, and then glues the sets U_F to each other in an appropriate manner. To construct U_F, for a face F of some $P_i \in \mathcal{P}$, we choose a random order of the polyhedra in $\mathcal{P} \setminus \{P_i\}$ and insert the polygons $Q_j \in \mathcal{Q}_F$, one by one, in the corresponding order, maintaining the complement of their union (within the plane containing F) as we go. For this we use the same technique as in [10, 18, 24], which maintains a vertical decomposition (relative to some direction within F) of the complement into trapezoids. When the incremental procedure ends, we truncate the resulting complement to within F. See [4, 5] for more details.

The analysis of the expected running time of the algorithm is essentially identical to that given in [4, 5]. One only has to plug the improved bound $O(nk \log k)$ on the complexity of the union into the appropriate expressions given in [4, 5]. Omitting these routine calculations, we obtain the following theorem.

THEOREM 5.1. *The union of a collection of k expanded convex polyhedra in 3-space, as above, with a total of n faces, can be computed in randomized expected time $O(nk \log k \log n)$. (In terms of the original parameters p and q, the expected running time is $O(pqk \log k \log (pq))$.)*

Next we apply Theorem 5.1 to the problem of planning a purely translational collision-free motion of the "robot" B amidst the obstacles $A_1, \ldots, A_k$. As observed previously, the complement $\mathcal{C}$ of the union of the expanded obstacles is an adequate representation of the free configuration space of B. However, the representation of $\mathcal{C}$, as computed by the preceding algorithm, needs a few enhancements to facilitate processing of motion planning queries. Here we describe one such method, but alternative (and, hopefully, more efficient and/or simpler) techniques should also be explored.

The first step is to link together all the connected components of the boundary of each connected component of $\mathcal{C}$. This can be done in several ways. For example, we can take the highest point w on each (bounded) boundary component σ and consider the upward-directed ray ρ_w emanating from w. If ρ_w leaves $\mathcal{C}$ near w, then σ is the *outer* boundary of this component of $\mathcal{C}$, and nothing needs to be done. Otherwise, trace ρ_w and find the first polyhedron P_i (if any) met by that ray. The hitting point w' necessarily lies on $\partial\mathcal{C}$ (but not on σ), and the connections between w and w', over all inner boundary components σ, yield the desired links, as is easily verified. (The case when ρ_w meets no polyhedron, which can happen when σ is an unbounded boundary component, requires different, though equally simple, treatment.) To find the point w', we simply find the intersections (if any) of ρ_w with each of the P_i's, by explicit enumeration of all the n faces of the P_i's, and choose the point nearest to w. If w' is found to lie in a face F of some P_i, we use the point location structure for U_F, which is obtained as a byproduct of the incremental construction of the vertical

decomposition of U_F (see [4, 5]), to determine which face of $F \cap \partial U$ contains w'. Since the highest point w on an inner component σ of $\partial\mathcal{C}$ must be the top vertex of one of the P_i's, the whole step takes time $O(nk)$.

We next scan all the faces of $\partial\mathcal{C}$ and assign to each of them the connected component of $\mathcal{C}$ that it bounds. This is easily done by a depth-first search through the adjacency graph of the trapezoids forming $\partial\mathcal{C}$, augmented by the additional adjacencies induced by the new vertical links computed above. This can be done in time linear in the size of $\partial\mathcal{C}$, that is, in time $O(nk \log k)$.

Now, suppose we are given two placements ζ_1, ζ_2 of B and wish to determine whether B can be moved from ζ_1 to ζ_2 without colliding with any A_i. To this end, let ρ_1, ρ_2 be the upward-directed rays emanating from ζ_1, ζ_2, respectively. Apply the procedure described above to these rays, and let ξ_1, ξ_2 be the first points where the respective rays ρ_1, ρ_2 meet $\partial\mathcal{C}$. We then simply check whether the trapezoids on the faces of $\partial\mathcal{C}$ containing ξ_1, ξ_2 bound the same component of $\mathcal{C}$, using the information computed in the preliminary stage. If this is the case, then collision-free translational motion of B from ζ_1 to ζ_2 is possible; otherwise no such motion is possible. This "decision procedure" takes time $O(n)$, if we test explicitly all n faces of the P_i's, as above; this can be improved to $O(k \log n)$ time by preprocessing each polyhedron P_i for efficient line-intersection queries (as in [13]) and then by computing the intersection points of the rays ρ_1, ρ_2 with each polyhedron separately, in logarithmic time per polyhedron. Computing the trapezoids of $\partial\mathcal{C}$ containing ξ_1 and ξ_2, respectively, can be done by point-location queries for ξ_1 and ξ_2 in the respective point-location structures of U_{F_1} and U_{F_2}, computed by the preceding algorithm, where F_i is the face containing ξ_i, for $i = 1, 2$.

It is also rather straightforward to produce a "semifree" motion of B (i.e., a motion during which B does not penetrate, but may touch the obstacles) from ζ_1 to ζ_2, when one exists, as the concatenation of the segments $\zeta_1\xi_1$ and $\zeta_2\xi_2$ with a path that proceeds along the boundary of $\mathcal{C}$, and, if necessary, also along vertical links produced in the preliminary stage. We omit here the rather easy details.

COROLLARY 5.2. *Given a convex polyhedron B, free to translate among k convex polyhedral obstacles with pairwise-disjoint interiors, the entire free configuration space of B can be computed and preprocessed in randomized expected time $O(nk \log k \log n)$, where n is the total number of faces of the Minkowski sums of the obstacles and $-B$. Then, given two placements, ζ_1, ζ_2, of B, we can decide, in $O(k \log n)$ time, whether B can translate in a collision-free manner from ζ_1 to ζ_2.*

Remark. An interesting challenge is to revise the algorithm so that the above reachability queries can be performed in time faster than $O(k \log n)$, perhaps at only polylogarithmic cost.

6. Conclusions. In this paper we have shown that the combinatorial complexity of the union of the Minkowski sums of k convex polyhedra in three dimensions, having pairwise-disjoint interiors, with another convex polyhedron, is $O(nk \log k)$ (and $\Omega(nk\alpha(k))$ in the worst case), where n is the overall complexity of the individual Minkowski sums. We have also presented an efficient and rather simple randomized algorithm for computing the union in expected time $O(nk \log k \log n)$. Both the combinatorial bound and the algorithm have applications to translational motion planning of a convex polyhedral object in a three-dimensional environment amidst polyhedral obstacles, and we have also discussed these applications.

These results almost settle a long-standing open problem but raise a whole collection of new open problems; some of these problems have already been mentioned in

earlier sections. One open problem is to tighten the remaining gap between the lower and upper bounds on the complexity of the union. We conjecture that the correct worst-case bound is $\Theta(nk\alpha(k))$. There are also the problems of designing an efficient deterministic algorithm for computing the union and of improving the performance of the motion planning algorithm described above.

The more challenging and interesting open problems, however, involve generalizations and extensions of our results and techniques. First, what is the combinatorial complexity of the union of Minkowski sums $A_i \oplus B$, where the A_i's are k convex polyhedra with pairwise disjoint interiors, and B is a ball? Even the special case where the A_i's are lines seems to be open; in this case we want to bound the combinatorial complexity of the union of k congruent infinite cylinders, where the conjecture is that this complexity is near-quadratic in k. This problem arises in motion planning, when applying a standard heuristic of enclosing the moving (rigid) object by a ball, and planning the motion of the enclosing ball.

Another open problem involves *generalized Voronoi diagrams* in 3-space. Given $A_1, \ldots, A_k$ and B as above, the B-Voronoi diagram of the A_i's is the partition of 3-space into cells, $V(A_1), \ldots, V(A_k)$, where

$$V(A_i) = \{\xi \mid d_B(\xi, A_i) \le d_B(\xi, A_j) \text{ for all } j \ne i\},$$

where d_B is the *convex distance function* induced by B, namely, $d_B(x, y)$ is the smallest positive λ such that $y \in \{x\} \oplus \lambda B$, and $d_B(x, A) = \min\{d_B(x, y) \mid y \in A\}$. (Here, for the function d_B to be well defined, B must contain the origin in its interior. Also note that if B is a ball, this is the standard Euclidean Voronoi diagram of the "sites" A_i.) See [23] for a study of planar B-Voronoi diagrams. The problem is to bound the combinatorial complexity of the B-Voronoi diagram of the A_i's. If we view the diagram as the lower envelope of graphs of an appropriate collection of trivariate distance functions $w = d_B((x, y, z), A_i)$ in four dimensions, following the standard observation of [16], the results of this paper can be interpreted, as is easily verified, as bounding the complexity of any "horizontal" cross section $w = \text{const}$ of the envelope. The results of [28] imply that the complexity of the B-Voronoi diagram is $O(n^{3+\varepsilon})$, for any $\varepsilon > 0$, where n is the overall complexity of the corresponding Minkowski sums. Considerable progress was made very recently in [11]; it is shown there that the complexity of the Voronoi diagram of n lines in 3-space, under a polyhedral convex distance function, where the underlying polyhedron B has a constant number of edges, is $O(n^2\alpha(n)\log n)$. However, it is still an open problem to extend this result and obtain near-quadratic bounds for the general case where the A_i's and B are arbitrary polyhedra, as above (see [8] for some recent progress on this problem). The main challenge lying ahead is to obtain near-quadratic bounds for the complexity of Euclidean Voronoi diagrams for a set of polyhedral objects. No subcubic bounds are known as yet for this problem, except for the special case where the sites are points (and then the bound is actually quadratic).

Appendix. Proof of Proposition 3.3. In this appendix we give the proof of Proposition 3.3. Recall that this proposition asserts that the first Betti number $\beta_1(S)$ of any compact polyhedral set S in $\mathbb{R}^3$ in general position does not exceed the sum of the merge numbers of its vertices.

Let $z : S \to \mathbb{R}$ be the z-coordinate function. For $a, b \in \mathbb{R}$, let $S[a] = z^{-1}(\{a\})$, $S[a, b] = z^{-1}([a, b])$, and $S^a = z^{-1}((-\infty, a])$.

We begin the proof by stating and sketching a proof of a polyhedral analogue of a basic fact of Morse theory (cf. [25, Theorem 3.1]).

FACT 6.1. *Let $S \subseteq \mathbb{R}^3$ be a triangulated polyhedral set in general position. If $S[a,b]$ contains either no vertex of S, or exactly one vertex v and the z-coordinate of v is a, then $S[a]$ is a strong deformation retract*[2] *of $S[a,b]$.*

Sketch of proof. The proof is an easy and standard exercise in topology, but we include a brief sketch of it here for the sake of completeness. We consider only the case in which there exists a vertex $v \in S[a,b]$ with $z(v) = a$. For each $c \in (a,b]$, the intersection of the triangulation of S with the plane $z = c$ is a convex subdivision of $S[c]$ into triangles and quadrilaterals. It is easy to construct a triangulation, T_c, of this subdivision, in such a way that the combinatorial structure of T_c, for each $c \in (a,b]$, is the same. More formally, if we label each vertex of T_c with the edge of the triangulation of S on which it lies, T_c and $T_{c'}$ are isomorphic labeled planar maps, for any two values $c, c' \in (a,b]$.

For any $c \in (a,b]$ and any $p \in S[c]$, there is a triangle $\triangle(p)$ of T_c containing p, so we can write $p = \alpha_1 u_1 + \alpha_2 u_2 + \alpha_3 u_3$, where u_1, u_2, u_3 are the vertices of $\triangle(p)$, and $\alpha_1, \alpha_2, \alpha_3 \geq 0$, $\sum_i \alpha_i = 1$. For each $\xi \in (a,b]$, define $p(\xi) = \alpha_1 u_1(\xi) + \alpha_2 u_2(\xi) + \alpha_3 u_3(\xi)$, where, for $i = 1,2,3$, $u_i(\xi)$ is the vertex of T_ξ corresponding to u_i (i.e., they both lie on the same edge of the triangulation of S). We have $p(c) = p$, and $p(\xi)$ is a linear function of ξ over $(a,b]$ (for any fixed p), so it is continuous and converges to a limit $p(a)$ as $\xi \to a$. Now define the map $\rho : S[a,b] \times [0,1] \to S[a,b]$, so that, for $p \in S[a,b]$ and $t \in [0,1]$, we have $\rho(p,t) = p\big(z(p)(1-t) + at\big)$ if $z(p) > a$, and $\rho(p,t) = p$ if $z(p) = a$. It is easy to verify that ρ is continuous and satisfies $\rho(p,0) = p$, $\rho(p,1) \in S[a]$, for all $p \in S[a,b]$, and $\rho(p,t) = p$ for all $p \in S[a]$, $t \in [0,1]$. Hence, by definition, $S[a]$ is a strong deformation retract of $S[a,b]$. □

We now return to the proof of Proposition 3.3. We first triangulate S. This may add new vertices to S, but their presence does not affect the statement of the proposition, since they all have merge number 0. Let $z = c$ be a plane below all the vertices of S. Since S is assumed to be bounded, we have $S^c = \emptyset$, so $\beta_1(S^c) = 0$. It suffices to prove that, as t increases from c to $+\infty$, the Betti number $\beta_1(S^t)$ increases at (the z-coordinate of) each vertex of S by at most the merge number of the vertex, and never changes otherwise.

Fact 6.1 implies that $\beta_1(S^t)$ changes only when the plane $z = t$ sweeps through a vertex of S. Indeed, if $t' > t$ is such that no vertex of S has z coordinate in $(t,t']$, then $S[t]$ is a strong deformation retract of $S[t,t']$. However, $S^{t'}$ is the union of S^t and $S[t,t']$, attached along $S[t]$, so S^t is a strong deformation retract of $S^{t'}$ (extending the map provided in Fact 6.1 by the identity map over S^t). Thus S^t and $S^{t'}$ have the same homologies [30, Corollary 1.12], which implies that $\beta_1(S^{t'}) = \beta_1(S^t)$.

Let v be a vertex of S. We want to bound the difference $\beta_1(S^t) - \beta_1(S^{t'})$, where $t = z(v)$ and $t' < t$ is appropriately close to t. One can easily verify that the transformation from $S^{t'}$ to S^t is equivalent, topologically, to gluing a ball D centered at v to $S^{t'}$, choosing the size of D to be such that it intersects all triangles of $T_{t'}$ with at least one vertex approaching v as $t' \to t$, and no other triangles. An easy topological

[2] For a topological space X, a subspace $Y \subseteq X$ is called a *strong deformation retract* of X if there exists a continuous map $\rho : X \times [0,1] \to X$ such that $\rho(x,0) = x$ for all $x \in X$, $\rho(X,1) = Y$, and $\rho(y,t) = y$ for all $y \in Y, t \in [0,1]$; see, for example, [30, p. 56]. Intuitively, it means that X can be deformed into its subspace Y in a continuous fashion while staying in X and not moving a single point of Y.

calculation[3] yields

$$\beta_1(S^t) = \beta_1(S^{t'} \cup D) \leq \beta_1(S^{t'}) + \beta_1(D) + \beta_0(S^{t'} \cap D) - 1.$$

Here $\beta_0(X)$, the 0th Betti number of a topological space X, is the number of connected components of X. Now $\beta_1(D) = 0$ and $\beta_0(S^{t'} \cap D)$ is exactly equal to the number of components of the intersection of a neighborhood of v in S with the plane $z = t'$, lying just below v, namely, it is equal to $m(v) + 1$. Hence $\beta_1(S^t) - \beta_1(S^{t'}) \leq m(v)$, as claimed. This concludes the proof of Proposition 3.3. □

Acknowledgments. We wish to thank Dan Halperin for useful discussions on the problem, and Jiří Matoušek and Igor Rivin for helpful suggestions on matters of topology.

REFERENCES

[1] B. Aronov and M. Sharir, *The common exterior of convex polygons in the plane*, Comput. Geom., 8 (1997), pp. 139–149.
[2] B. Aronov and M. Sharir, *Triangles in space, or building (and analyzing) castles in the air*, Combinatorica, 10 (1990), pp. 137–173.
[3] B. Aronov and M. Sharir, *Castles in the air revisited*, Discrete Comput. Geom., 12 (1994), pp. 119–150.
[4] B. Aronov and M. Sharir, *The union of convex polyhedra in three dimensions*, in Proc. 34th IEEE Symp. on Foundation of Computer Science, Palo Alto, CA, 1993, pp. 518–529.
[5] B. Aronov, M. Sharir, and B. Tagansky, *The union of convex polyhedra in three dimensions*, SIAM J. Comput., 26 (1997), pp. 1670–1688.
[6] M. de Berg, J. Matoušek, and O. Schwarzkopf, *Piecewise linear paths among convex obstacles*, Discrete Comput. Geom., 14 (1995), pp. 9–29.
[7] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec, *Applications of random sampling to on-line algorithms in computational geometry*, Discrete Comput. Geom., 8 (1992), pp. 51–71.
[8] J.-D. Boissonnat, M. Sharir, B. Tagansky, and M. Yvinec, *Voronoi diagrams in higher dimensions under certain polyhedral convex distance functions*, in Proc. 11th ACM Symp. on Computational Geometry, Vancouver, British Columbia, 1995, pp. 79–88; Discrete Comput. Geom., to appear.
[9] B. Chazelle, *An optimal algorithm for intersecting three-dimensional convex polyhedra*, SIAM J. Comput., 21 (1992), pp. 671–696.
[10] B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and J. Snoeyink, *Computing a single face in an arrangement of line segments and related problems*, SIAM J. Comput., 22 (1993), pp. 1286–1302.
[11] L. P. Chew, K. Kedem, M. Sharir, B. Tagansky, and E. Welzl, *Voronoi diagrams of lines in three dimensions under a polyhedral convex distance function*, in Proc. 6th ACM-SIAM Symp. on Discrete Algorithms, San Francisco, CA, 1995, pp. 197–204; J. Algorithms, to appear.
[12] K. Clarkson and P. Shor, *Applications of random sampling in computational geometry* II, Discrete Comput. Geom., 4 (1989), pp. 387–421.
[13] D. P. Dobkin and D. G. Kirkpatrick, *Fast detection of polyhedral intersection*, Theoret. Comput. Sci., 27 (1983), pp. 241–253.
[14] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.
[15] H. Edelsbrunner, L. Guibas, and M. Sharir, *The upper envelope of piecewise linear functions: Algorithms and applications*, Discrete Comput. Geom., 4 (1989), pp. 311–336.
[16] H. Edelsbrunner and R. Seidel, *Voronoi diagrams and arrangements*, Discrete Comput. Geom., 1 (1986), pp. 25–44.

[3] It is easy to show that, for two intersecting path-connected topological spaces X and Y, $\beta_1(X \cup Y) \leq \beta_1(X) + \beta_1(Y) + \beta_0(X \cap Y) - 1$. If X is not path-connected, but Y is, the statement follows by repeated applications of the path-connected version. Note that in our case D is path-connected but $S^{t'}$ need not be. The statement easily follows from arguments based on the Mayer-Vietoris sequence; see, for example, Vick [30, p. 22].

[17] M. Greenberg and J. Harper, *Algebraic Topology: A First Course*, Benjamin-Cummings Pub. Co., Reading, MA, 1981.
[18] L. Guibas, D. Knuth, and M. Sharir, *Randomized incremental construction of Voronoi and Delaunay diagrams*, Algorithmica, 7 (1992), pp. 381–413.
[19] L. Guibas and R. Seidel, *Computing convolutions by reciprocal search*, Discrete Comput. Geom., 2 (1987), pp. 175–193.
[20] D. Halperin and C. K. Yap, *Combinatorial complexity of translating a box in polyhedral 3-space*, in Proc. 9th ACM Symp. on Computational Geometry, San Diego, CA, 1993, pp. 29–37; Comput. Geom. Theory Appl., to appear.
[21] D. Huttenlocher, K. Kedem, and M. Sharir, *The upper envelope of Voronoi surfaces and its applications*, Discrete Comput. Geom., 9 (1993), pp. 267–291.
[22] K. Kedem, R. Livne, J. Pach, and M. Sharir, *On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles*, Discrete Comput. Geom., 1 (1986), pp. 59–71.
[23] D. Leven and M. Sharir, *Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams*, Discrete Comput. Geom., 2 (1987), pp. 9–31.
[24] N. Miller and M. Sharir, *Efficient randomized algorithms for constructing the union of fat triangles and of pseudo-disks*, manuscript, 1991.
[25] J. Milnor, *Morse Theory,* Princeton University Press, Princeton, NJ, 1963.
[26] J. Pach and M. Sharir, *The upper envelope of piecewise linear functions and the boundary of a region enclosed by convex plates: Combinatorial analysis*, Discrete Comput. Geom., 4 (1989), pp. 291–309.
[27] M. Sharir, *Efficient algorithms for planning purely translational collision-free motion in two and three dimensions*, in Proc. IEEE Symp. on Robotics and Automation, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 1326–1331.
[28] M. Sharir, *Almost tight upper bounds for lower envelopes in higher dimensions*, Discrete Comput. Geom., 12 (1994), pp. 327–345.
[29] M. Sharir and P. K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, Cambridge, New York, Melbourne, 1995.
[30] J. W. Vick, *Homology Theory*: *An Introduction to Algebraic Topology*, Academic Press, New York, 1973.